Introducción al

Lenguaje de Programación Pascal

Primera Parte

Apuntes de la Cátedra
Principios de Lenguajes de Programación

Facultad de Informática
Universidad Nacional del Comahue

Marcelo Paulo Amaolo

4 de abril de 2019

Índice

1. Introducción

El presente trabajo se constituye como un apunte de cátedra con el que se introduce al alumno a la programación basada en el paradigma imperativo, particularmente con el uso del lenguaje de programación Pascal.

Los resúmenes incorporados en el presente se construyeron en base a la experiencia de la enseñanza del paradigma y el lenguaje en esta y otras cátedras, y en las habilidades logradas en el mundo laboral con el uso de lenguajes imperativos.

Si bien el apunte refleja cabalmente los conceptos iniciales que se buscó presentar, es sólo base para que los alumnos puedan comprender y utilizar un nuevo lenguaje en proyectos de programación simples que abarquen una buena parte del espectro de las funcionalidades que Pascal puede ofrecer para desarrollar incluso aplicaciones como soluciones informáticas profesionales.

Se abordan conceptos básicos del lenguaje y el uso de estructuras de datos estáticas. Las estructuras de datos dinámicas del lenguaje de programación Pascal se prevé sean incorporadas en una nueva versión del apunte en los próximos cursados.

La versión 2018 del apunte es similar en contenido a las versiones anteriores, habiendo incorporado nuevos ejemplos y corregido los errores detectados. Valga cualquier apunte de la cátedra para avanzar en el aprendizaje del lenguaje de programación Pascal.

1.1. Paradigma Imperativo

En sus inicios los lenguajes de programación imitaron e intentaron abstraer las operaciones de una computadora, por ello, el tipo de computadora para la cual fueron escritos tuvo un efecto significativo sobre su diseño. En la mayoría de los casos la computadora que se utilizó fue el modelo basado en los principios de Von Neumann: una unidad de procesamiento central, única, que en forma secuencial, ejecuta las instrucciones que operan sobre valores almacenados en la memoria.

En la máquina de Von Neumann, tanto los datos como los programas están almacenados en la misma memoria. La Unidad Central de procesamiento (CPU), que ejecuta las instrucciones, está separada de la memoria. Entonces, las instrucciones y los datos deben "transmitirse" desde la memoria a la CPU. Los resultados de las operaciones que realiza la CPU deben devolverse para su almacenamiento en la memoria. La mayoría de los lenguajes de programación desde los años 40s se diseñaron pensando en este funcionamiento, que se esquematiza en la Figura 1.

Ciertamente, el resultado de la completitud de Turing se refiere a este principio: un lenguaje de programación es completo en Turing siempre que tenga variables enteras y aritméticas, y ejecute enunciados en forma secuencial, incluyendo enunciados de *asignación*, *selección* y *ciclo*. Así, los lenguajes basados en este principio, requieren de características típicas de un lenguaje basado en el modelo de arquitectura de Von Neumann: variables, que representan valores de memoria, y asignación que permite que el programa opere sobre dichos valores.

Un lenguaje de programación caracterizado por estas propiedades, ejecución secuencial de instrucciones, el uso de variables en representación de localizaciones de memoria y el uso de la asignación para cambiar el valor de las variables, se conoce como un lenguaje "**imperativo**", puesto que su característica principal es una secuencia de enunciados que representan comandos, u órdenes. Algu-

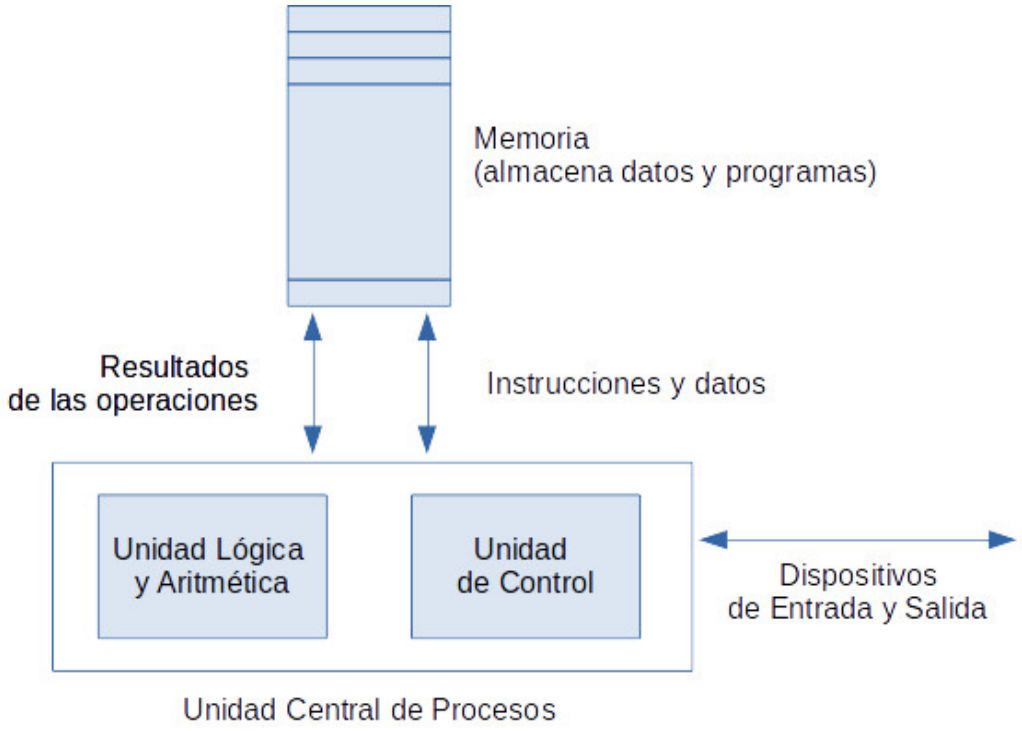


Figura 1: Arquitectura de una computadora

nas veces a este tipo de lenguajes se les llama también "**procedurales**"[1]. Una enorme cantidad de lenguajes de programación son imperativos, y muchos otros lenguajes que operan en base a otros paradigmas, heredan gran cantidad de propiedades de este tipo de lenguajes.

Por ejemplo, en cierto sentido, el paradigma "*orientado a objetos*" es una extensión del paradigma imperativo, ya que se base principalmente en la misma ejecución secuencial con un conjunto cambiante de localizaciones de memoria. La diferencia consiste en que los programas resultantes están formados de un gran número de piezas muy pequeñas, cuyas interacciones están cuidadosamente controladas y al mismo tiempo son fácilmente cambiadas. El paradigma orientado a objetos hoy se ha convertido en el estandar, de la misma manera que en el pasado fue el paradigma imperativo.

1.2. Implementaciones y herramientas

El lenguaje de programación Pascal, así como varias de sus variantes, sigue siendo una alternativa utilizada por muchos desarrolladores de aplicaciones [2], si bien ha perdido mucha popularidad frente a otras opciones como el lenguaje C, C++ y Java.

[1] Los lenguajes procedurales son los lenguajes que soportan la programación estructurada basada en la llamada a subprogramas o procedimientos (rutinas, subrutinas, funciones, etc.) como una cadena serial de pasos computacionales, cualquiera de los cuales puede ser una llamada a otro subprograma o incluso a si mismo.

[2] Si bien ha descendido mucho en los últimos 25 años, permanece en las clasificaciones que se hacen de los lenguajes más empleados en la comunidad de programadores. El índice TIOBE, uno de los más consultados, la variante de Pascal Delphi y Object Pascal, figura en el puesto 13 (con 1,372 por ciento) a agosto de 2018, mientras que el propio lenguaje Pascal aparece en el puesto 17 con un 1,099 por ciento en el año 2015 y fuera de rango (más allá de los 100) en 2018.

El proyecto FreePascal nace en el año 1.996, bajo licencia GNU/GPL, y constituye un excelente entorno para aquellos que quieran conocer el lenguaje, para luego poder construir proyectos más desafiantes utilizando otros entornos visuales, tales como Delphi y Kilyx[3].

1.3. Formatos

Este apunte contiene un gran cantidad de códigos ejemplo, y para ayudar al lector, los códigos y las salidas están adecuadamente indicadas. Por ejemplo, para diferenciar el código fuente del resto del texto del apunte, se distingue de esta manera:

```
El código fuente tiene esta apariencia
y ésta también.
```

```
Las interacciones tienen esta apariencia.
y ésta también
```

1.4. Cómo comenzar

Existen varias herramientas, accesibles y la mayoría de ellos libres y gratis, para trabajar con el lenguaje de programación Pascal. En este curso y en este apunte vamos a hacer referencia al la herramienta "Free Pascal", disponible para varios sistemas operativos. La herramienta incluye un compilador, que se puede invocar en la línea de comandos como "fpc", y un entorno de desarrollo (fp), que incluye un editor, permite la compilación y también un debugger[4] para hacer el seguimiento de un programa.

Para los usuarios de la mayoría de los sistemas operativos Linux, pueden instalarlo directamente de sus repositorios. En el caso de FIDebian[5] la instalación puede realizarse como superusuario escribiendo en la línea de comandos

```
sudo apt-get install fpc
```

o a través de un GUI, tal como el Administrador de Paquetes o el Administrador de Aplicaciones.

Para otros sistemas operativas, los usuarios deberían conectarse por internet con el sitio oficial del proyecto "http://www.free-pascal.org" para bajar el instalador en la pestaña "Download".

Para su uso, en Linux, tiene dos opciones. Trabajar con el entorno que permite editar los archivos, compilarlos y ejecutarlos e incluso depurarlos (debugging), utilizando en la línea de comandos (terminal de linux) el comando "fp". De otra forma, trabajar con un editor de su elección, compilarlo utilizando el comando "fpc", para posteriormente ejecutarlo conveniente.

En la próxima sección se explicará el uso con un ejemplo simple, para posteriormente concentrarse en las estructuras y constructores propios del lenguaje.

[3] El FreePascal es un compilador del lenguaje Pascal portable, libre y de código abierto, que funciona en formato texto y que cuenta con un Entorno de Desarrollo Interactivo -IDE- y refiere a otros proyectos. Ver http://www.freepascal.org/)

[4] El debugger está en desarrollo en la herramienta. Puede utilizarse como reemplazo el software libre que liberó Borland, que pude descargarse en **turbopascal.org** opción **download**

[5] El sistema operativo de la Facultad de Informática basado en Debian

2. Escribiendo programas

Para escribir un programa en el lenguaje Pascal, es necesario respetar un conjunto de reglas y símbolos, que pueden escribirse en cualquier editor de texto, y que deben guardarse en un archivo, generalmente con la extensión ".pas" o ".pp", que al momento de compilarlos se transformarán en un archivo ejecutable.

2.1. Mi primer programa

Tomemos como ejemplo el siguiente programa:

```
PROGRAM ejemplo; { Cabecera }
   CONST max=100; { Constantes }
   VAR a:integer; { Variables }
BEGIN { <== inicia el programa principal }
   write ('Escriba un número: ');
   read(a);
   a := a+max;
   writeln('El resultado de A + 100 es: ',a);
END. { <== finaliza el programa principal }
```

que suma el valor constante 100 a un entero ingresado por teclado.

Para poder escribirlo, puede utilizarse un editor habitual[6] para grabarlo con el nombre "*ejemplo.pas*". Para poder compilarlo, desde una terminal, en el mismo directorio en el que fue almacenado el archivo fuente, en la línea de comandos se escribe:

```
$ fpc ejemplo
```

y el sistema mostrará un mensaje informando que fue compilado, y que fueron creados dos nuevos archivos, los archivos "*ejemplo.o*" y "*ejemplo*"[7], el primero de ellos el objeto del fuente, y el segundo, un programa ejecutable. Para ejecutarlo simplemente debo invocar al programa. Su ejecución puede verse, cuando el operador ingresa el número 5 por teclado, como:

```
$ ./ejemplo
Escriba un número: 5
El resultado de A + 100 es: 105
$
```

2.2. Algunas consideraciones

En general, las implementaciones del lenguaje Pascal incorporan bastante flexibilidad para que los programadores puedan escribir programas más fácilmente. Por ejemplo, es obligatorio el bloque de programas, y las otras declaraciones pueden evitarse.

[6] Tal como el editor "*pluma*" para FIDebian
[7] En Windows el ejecutable se llamaría "*ejemplo.exe*"

Por otro lado, el lenguaje no es sensible a letras mayúsculas, por lo que una variable denominada "variable" puede invocarse como "Variable", "variable", "VARIable" o "VaRiAbLe".

Se denominan **palabras reservadas** a las palabras que no pueden ser utilizadas como identificadores en el programa. Las palabras reservadas las utiliza el compilador para fines propios, tales como para saber dónde empieza el programa, donde termina un subprograma, etc.[8]. Un listado no exhaustivo de las palabras reservadas del lenguaje Pascal se detallan en el apéndice 1, página 52.

En general todos los identificadores deben estar declarados. Cuando el compilador encuentra un identificador que no ha sido declarado (identificador de Constante, de Variable, de Tipo, nombre de Subprograma, etc.) señala un error y no continúa la compilación [9].

Por ejemplo, si el programa anterior hubiera sido escrito, modificando el nombre de la constante (antes era *max* y ahora es *maximo*) como:

```
PROGRAM ejemplo;
   CONST maximo=100;
   VAR a:integer;
BEGIN
   write ('Escriba un número: ');
   read(a);
   a := a+max;
   writeln('El resultado de A + 100 es: ',a);
END.
```

durante la compilación, el compilador hubiera mostrado un mensaje de error como "*identificador max no encontrado*", con la siguiente salida:

```
$ fpc prueba8.pas
Free Pascal Compiler version 3.0.0+dfsg-2 [2016/01/28] for x86_64
Copyright (c) 1993-2015 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling prueba8.pas
prueba8.pas(7,8) Error: Identifier not found "max"
prueba8.pas(10) Fatal: There were 1 errors compiling module, stopping
Fatal: Compilation aborted
$
```

.

3. Programas en Pascal

Un programa Pascal es un conjunto de instrucciones que siguen la sintaxis y la estructura del lenguaje Pascal[10]. La estructura genérica es:

[8] Para mayor claridad, en este apunte, las palabras reservadas se escriben en mayúsculas en todos los ejemplos.
[9] Pascal implementa un compilador de una sola pasada
[10] Si bien existen varios algunos dialectos locales de Pascal, en este apunte se intentará trabajar con la sintaxis estándar según normas ISO 7185 y se señalará explícitamente cuando se trabaje en otros estándares tales como Borland Pascal.

```
PROGRAM nombre (ficheros);
. . .
declaraciones
. . .
BEGIN
. . .
sentencias
. . .
END.
```

Todos los programas empiezan con la palabra reservada PROGRAM, seguida de un nombre que elige el programador para identificar el programa. A continuación y en paréntesis se pueden indicar los archivos de datos de entrada y salida respectivamente. Los archivos *input* y *output* se utilizan para indicar la entrada desde el *teclado* y la salida a la *terminal o pantalla*. Así, la primer línea del programa podría ser:

```
PROGRAM nombre (Input , Output );
```

que es equivalente a

```
PROGRAM nombre;
```

Notar que al finalizar la sentencia se escribe el símbolo punto y coma (;) que en Pascal significa que termina la instrucción.

Después de la identificación del programa se han de situar las instrucciones declarativas del programa que sirven para especificar sin ambigüedad el significado de los términos que se utilizaran en el programa. Posteriormente deben escribirse las instrucciones correspondientes al programa que se quiere realizar. Esta instrucciones están encabezadas por BEGIN y terminan con END y un punto.

El programa mas pequeño y más inútil que cumple las reglas de estructuración de PASCAL es:

```
PROGRAM nulo;
   { Programa ejemplo de la estructura
     más simple de un programa PASCAL }
BEGIN
   (* No hace falta ninguna instrucción
     para no hacer nada *)
END.
```

En la parte reservada a *declaraciones* no se incluye nada pues nada se necesita declarar. Todos los símbolos que se encuentren entre los paréntesis "`{ }`" son comentarios que sirven para hacer más legible el programa. También los símbolos compuestos "`(*`" y "`*)`" sirven para delimitar el principio y fin de un comentario. Al existir dos tipos de delimitadores de comentarios, es posible realizar anidación de comentarios. Por ejemplo,

```
   { Este es un comentario
     (* sintácticamente *)
   correcto en PASCAL }
```

3.1. Estructura de un Programa Pascal

Palabra Clave	Componentes de Programa	Ejemplo
`Program`	Cabecera	`PROGRAM Ejemplo;`
`Const`	Declaración de Costantes	`CONST Maximo = 234 ;`
`Type`	Declaración de Tipos	`TYPE Entero = integer;`
`Var`	Declaración de Variables	`VAR Anio: Real;`
`Procedure`	Declaración de Procedimientos	`PROCEDURE P1(A:T2; B,C:T1);` `...` `BEGIN` `...` `END;`
`Function`	Declaración de Funciones	`FUNCTION F1(N:T2):integer;` `...` `BEGIN` `...` `END;`
`Begin` *sentencias* `End`	Bloque de Programa	`BEGIN` *bloque del programa;* `END.`

Nota: En el cuadro se muestran las declaraciones de subprogramas con un orden particular: primero los procedimientos y posteriormente las funciones. En realidad la declaración de subprogramas (funciones o procedimientos) no requiere de un ordenamiento[11]. En el cuadro se presenta en este orden para simplificar la exposición.

4. Declaraciones

4.1. Constantes

La característica principal de una constante es que su valor **no** puede ser cambiado a lo largo del programa. Para utilizar una constante, debe dársele un nombre (un identificador).

La declaración de una constante asocia el identificador deseado con una constante. A partir de ese momento, utilizar el identificador asociado a la constante, o utilizar su valor directamente es indiferente.

[11]Es decir, un programa puede declarar sólo procedimientos, otro sólo funciones, y un tercero ambos en cualquier orden

La palabra reservada CONST debe encabezar la instrucción, seguido por una lista de declaraciones de constantes. Cada declaración de constante consiste de un *identificador* seguido por un signo de *igual* y un valor constante. Un valor constante puede consistir de un número (entero o real), o de una constante de caracteres. La constante de caracteres consiste de una secuencia de caracteres encerrada entre apóstrofes (')[12], si los valores numéricos van precedidos de $ serán valores hexadecimales, si empiezan con % son valores binarios. Por ejemplo:

```
CONST
   Val_Max = 255;
   precision = 0.0001;
   identificador_clave = 'COPERnico';
   DiasDeLaSemana = 7;
   base16 = $16;
   encabezado = ' NOMBRE DIRECCION TELEFONO ';
```

Por ejemplo, el resultado de ejecutar uno de los programas siguientes es el mismo :

```
PROGRAM EjConstantes;
BEGIN
   write(3.1415);
END.
```

```
PROGRAM EjConstantes2;
CONST
   NumeroPi = 3.1415;
BEGIN
   write(NumeroPi);
END.
```

En este ejemplo, NumeroPi es el nombre de una constante, y 3.1415 es su valor.

```
$ ./ejconstantes
3.1415
$ ./ejconstantes2
3.1415
$
```

Pascal proporciona las siguientes constantes predefinidas[13]:

Nombre	Tipo	Valor
FALSE	boolean	
TRUE	boolean	
MAXINT	integer	32767
PI	real	3.1415926536

[12] Código ASCII *39*, o entidad HTML *'*. Por las características de edición del procesador de texto con el que se escribió el apunte, puede aparecer en otro caracter.

[13] Algunas implementaciones definen el valor de MAXINT dependiendo del tamaño del tipo entero en la arquitectura de la computadora, tales como "$2^{16-1} - 1$", "$2^{32-1} - 1$", etc. Otras no definen el valor de PI.

4.2. Variables

Una variable, al igual que una constante se almacena en una posición de memoria, pero al contrario de las constantes cuyo valor no varia, el valor de una variable cambiará durante el transcurso de la ejecución de un programa.

La forma de definir las variables en PASCAL es como sigue:

```
VAR
   contador : integer;
   nombre : String;
```

Cada variable pertenece a un tipo determinado y hay que indicar el tipo en el bloque de programa VAR, si esto no ocurre, el compilador informará un error. Existen tres tipos básicos de variables, las variables numéricas, las alfanuméricas y las booleanas. El tipo de una variable (es extensible a otros constructores del lenguaje) define los valores posibles que puede tener una variable, y las operaciones con las que se puede operar esa variable.

Las variables numéricas son enteros y reales. Las variables alfanuméricas son caracteres. El tipo booleano pueden tener 2 valores (verdadero y falso).

En todo momento de un programa, cada variable contiene un valor, incluso al comienzo del programa. Por tanto, si escribimos el siguiente programa obtendremos un resultado inesperado:

```
PROGRAM EjemploVariables2;
VAR
   Edad : integer;
BEGIN
   write(Edad);
END.
```

4.2.1. Tipos numéricos

Enteros

Tipo	Rango
byte	0 .. 255
shortint	-128 .. 127
integer	-32768 .. 32767
word	0 .. 65535
longint	-214783648 .. 214783647

Reales

Tipo	Rango
single	$1{,}5 \times 10^{-45}$ .. $3{,}41 \times 10^{38}$
real[14]	5×10^{-324} .. $1{,}7^{308}$
double	$5{,}01 \times 10^{-324}$ .. $1{,}71 \times 10^{308}$
extended	$1{,}9 \times 10^{-4932}$ .. $1{,}1 \times 10^{4932}$
comp	$-2 \times 10^{63} + 1$ .. $2 \times 10^{63} - 1$

4.2.2. Tipos alfanuméricos

Tipo	Rango
char	1 caracter
shortstring	255 caracteres
string(n)	N caracteres

4.2.3. Tipos Booleanos

Tipo	Rango
boolean	TRUE / FALSE

4.3. Operaciones asociadas a los tipos

Los operadores u operaciones sirven para combinar términos de las expresiones que se construyen a partir de elementos del lenguaje. En Pascal se gestiones tres grupos de operadores:

- Aritméticos
- Relacionales
- Lógicos

4.3.1. Operaciones y funciones Aritméticas

Las operaciones aritméticas siven para operar términos numéricos. Estos operadores están clasificados a su vez en:

- Unarios, operadores que trabajan con un único operando.
  - "$-$" Denota la negación del operando aritmético. Por ejemplo, si x tiene el valor 100, $-x$ tendrá el valor -100.
  - "abs", para calcular el valor absoluto, para Enteros y Reales.
  - "sqr", para el cuadrado, para Enteros y Reales.
  - "sqrt", para calcular la raíz cuadrada de un número, , para Reales.
  - "pred", predecesor de un entero.
  - "succ", sucesor de un número entero.
  - funciones trigonométricas, "sin", "cos" y "arctan", para Reales.
  - "log", calcula el logaritmo neperiano, para Reales.
  - "exp", la exponencial con base e, para Reales.

- Binarios, que combinan dos operandos, dando como resultado un valor numérico cuyo tipo corresponderá al tipo más grande de los tipos que tengan los operandos:
  - "+", operación de suma, para Enteros y Reales. Ejemplo: `a + b`.
  - "−", operación de resta, para Enteros y Reales. Ejemplo: `a - b`.
  - "*", operación de producto o multiplicación, para Enteros y Reales. Ejemplo: `a * b`.
  - "/", operación de división, para Enteros y Reales. Ejemplo: `a / b` da como resultado el cociente de a por b.
  - "div", operación de división entera, sólo para Enteros. Ejemplo: `a div b`, da como resultado el cociente entero de a por b.
  - "mod", operación de módulo, sólo para Enteros. Ejemplo: `a mod b` da como resultado el resto de la divisió de a por b.

Es importante considerar que Pascal realizará el cambio del tipo del resultado en forma automática, según los tipos de los operandos:

1. dos operandos **enteros**, el resultado será de tipo **entero**.
2. dos operandos **enteros**, con el operador "/" y cuyo resultado no es exacto, el resultado será de tipo **real**.
3. si alguno de los operandos (o ambos) son **reales**, el resultado será de tipo **real**.
4. para la operación "div", si ambos operadores tienen el mismo signo, el resultado es positivo, si difieren en signo, el resultado es negativo, con truncamiento hacia el cero.
5. las precedencia de los operadores son similares a las de la matemáticas, ver Anexo 2.

A continuación se muestran algunos ejemplos de cálculo de expresiones, con el cálculo en etapas:

```
= 2 + 3 * Sqr(2)
= 2 + 3 * 4
= 2 + 12
= 14
```

```
= 7 div 2 * 2
= 3 * 2
= 6
```

```
= 6 mod 3 + (-5) mod 3
= 0 + (-5) mod 3
= -2
```

Es importante recalcar que en las operaciones aritméticas, el propio programador debe asegurarse que el resultado de aplicar la suma, resta o multiplicación de dos valores, no produzca un resultado fuera de los rangos definidos por la implementación para los diferentes tipos.

Se incluyen también dos funciones que vinculan el tipo Real con el tipo Entero:

- "trunc", que trunca el valor decimal de un número Real para dar como salida un número Entero.
- "round", que redondea el valor decimal de un número Real al Entero más próximo[15].

Por ejemplo, siguiente la misma metodología anterior:

```
= Round(-3.6)
= - 4
```

```
= Trunc(-99.9)
= - 99
```

```
= - Round(99.9)
= - 100
```

```
= - Round(-99.9)
= 100
```

4.3.2. Operadores Relacionales

Una *relación* consiste de dos operandos separados por un operador relacional. Si la relación es satisfecha, el resultado tendrá un valor booleano verdadero (TRUE) ; si la relación no se satisface, el resultado tendrá un valor valso (FALSE). Los operadores deben ser del mismo tipo, aunque los valores de tipo *real* e *integer* pueden combinarse como operandos en las relaciones. A continuación se describen los operadores relacionales utilizados en Pascal:

Símbolo	Significado	Condición
=	igual	$a = b$
<>	distinto	$a \neq b$
<	menor	$a < b$
<=	menor o igual	$a \leq b$
>	mayor	$a > b$
>=	mayor o igual	$a \geq b$

Notar que estas operaciones se encuentran sobrecargadas, es decir, permiten la comparación entre valores de cualquiera de los tipos básicos, resultando de ello un valor lógico.

Con respecto a la comparación de números reales, se deduce la escasa fiabilidad de estas operaciones cuando los argumentos están muy próximos, como consecuencia de las limitaciones de precisión en los sistemas de representación de estos números en la arquitectura subyacente.

[15] Inferior o superior

En cuanto a la comparación de datos no numéricos, el símbolo < significa "anterior" y no "menor" que, obviamente, no tiene sentido en tales casos. En el caso concreto de `char`, esta anterioridad viene dada por sus posiciones en la tabla adoptada[16], y en el de boolean, se considera FALSE anterior a TRUE.

Por ejemplo:

```
= 20 = 11
= FALSE
```

```
= 15 < 20
= TRUE
```

```
= PI > 3.14
= TRUE
```

```
= 'A' < 20
= FALSE
```

```
= 'A' = 65
TRUE
```

Las operaciones relacionales tienen menor precedencia que las aditivas:

```
= 3 + 1 >= 7 - 2 * 3 { ops. multiplicativos }
= 3 + 1 >= 7 - 6 { ops. aditivos }
= 4 >= 1 { ops. relacionales }
FALSE
```

4.3.3. Operaciones Lógicas

Al igual que las relaciones, en las operaciones con operadores lógicos se obtienen resultados cuyo valor de verdad toma uno de los valores booleanos verdadero (TRUE) o falso (FALSE). Asociadas al tipo booleano, se encuentran:

Símbolo	Significado
`NOT`	Negación lógica, con la precedencia más alta.
`AND`	Conjunción lógica, con precedencia multiplicativa
`OR`	Disyunción lógica, con precedencia aditiva

que funciona respondiendo a una tabla de verdad habitual:

[16] Por ejemplo ASCII o la que el sistema implemente

A	B	A and B	A or B	not A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

con operaciones definidas con los siguientes valores:

```
= succ(FALSE)
= TRUE
```

```
= pred(TRUE)
= FALSE
```

```
= ord(FALSE)
= 0
```

```
= ord(TRUE)
= 1
```

4.3.4. Operadores del tipo Char

No existen operaciones internas entre caracteres definidas por el lenguaje. Las funciones predefinidas “pred” y “succ” tienen valor definido como fuera expresado anteriormente.

Existen funciones que permiten convertir entre los valores de los tipos “char” y “entero”:

Símbolo	Significado
`ord`	número de orden del caracter en el juego de caracteres adoptado.
`chr`	caracter asociado a un número de orden dado

En este esquema, por ejemplo:

```
= pred('Z')
= 'Y'
```

```
= pred('z')
= 'y'
```

```
= succ('7')
= '8'
```

```
= chr(ord('a') + 4))
= 'e'
```

5. Instrucciones

Aunque un programa en Pascal puede contar con una sola instrucción[17] (también llamada *enunciado*, *sentencia*, etc.), normalmente incluye una cantidad considerable para realizar alguna computación según la intención del programador que escribió el programa. Existen varios tipos de instrucciones, si bien con un conjunto limitado de ellas es posible escribir cualquier tipo de programa.

5.1. Asignación

La instrucción de *asignación* permite almacenar un dato en una variable y se trata de la instrucción utilizada con mayor frecuencia en el lenguaje Pascal. Con ella se le da el valor inicial a las variables o se modifica el valor que ya tienen.

En algunos compiladores, una variable declarada presenta un valor indefinido al iniciarse el programa, o un valor basuro, representado por el contenido de la memoria reservado cuando se declaró la variable. Lógicamente, un programa que depende de valores indefinidos tienen un comportamiento indeterminado, por ello es necesario evitar el operador con tales variables, asignándoles valores iniciales.

La asignación graba un valor en la memoria y destruye su valor previo, tanto si es un valor concreto como si es indeterminado. Por ejemplo:

```
x1 := (-b + sqrt(sqr(b) - 4 * a * c)) / (2 * a)
```

consta de un identificador de variable (x1), el símbolo de asignación ":=" (dos puntos igual) y una expresión. El proceso de asignación se produce de la siguiente manera: primero se evalúa la expresión, calculándose el valor final, y a continuación se almacena ese valor en la memoria.

La expresión puede contar con valores de constantes, variables, funciones[18], etc., y su combinación.

Por ejemplo la instrucción de asignación

```
contador := contador + 1
```

es una instrucción que tiene por objeto incrementar en una unidad el valor de la variable `contador` mientras que la igualdad

```
contador = contador + 1
```

es una expresión booleana de una relación que, es falsa cualquiera sea el valor de `contador`.

5.2. Bloque de instrucciones

En todo lugar en el que sea válido utilizar una instrucción simple, es posible utilizar una instrucción compuesta o *bloque de instrucciones*, que se forma agrupando varias instrucciones simples por medio de las palabras reservadas BEGIN y END. Por ejemplo:

[17] O ninguna, según la implementación
[18] Se presentarán más adelante en el apunte

```
BEGIN
   suma := 1000.0;
   incr := 20.0;
   total := suma + incr
END
```

No es necesario escribir el punto y coma antes de END ya que el *punto y coma* se usa para *separar* instrucciones, no para *terminarlas*. BEGIN y END son *delimitadores de bloque*.

5.3. Instrucciones de Entrada y Salida

Las instrucciones de Entrada y Salida se especifican dentro del cuerpo principal o bloque de instrucciones de un programa Pascal, y permiten que el programa se comunique con un periférico de manera que se pueda transmitir información desde el exterior a la memoria de la computadora o viceversa. El resultado de una instrucción de Entrada o de Salida depende del contenido de la memoria y del tratamiento de las expresiones presentes en esas instrucciones. Las instrucciones de *entrada estándar* sirven para leer caracteres del teclado, las instrucciones de *salida estándar* despliegan información en la pantalla.

En Pascal todas las operaciones de Entrada/Salida (E/S o I/O) se realizan ejecutando procedimientos de E/S que forman parte del lenguaje, con nombres estándares:

- Procedimientos de Entrada: READ y READLN
- Procedimientos de Salida: WRITE y WRITELN

5.3.1. READ y READLN

READ y READLN (contracción de READ LINE) constituyen las instrucciones básicas para las operaciones de *entrada estándar*. Con estas instrucciones el programa puede recibir información del exterior, con la siguiente forma:

```
read ( lista de variables );
readLn ( lista de variables );
```

en el que *lista de variables* puede el identificador de una variable o una lista de identificadores de variables separados por comas. Las variables pueden ser de tipos enteros, reales, caracteres o cadenas. **No se pueden leer variables de tipo boolean**[19].

Como ejemplo práctico podemos encontrar:

[19] Así como de otro tipo de variables estructuradas que se presentarán próximamente.

```
PROGRAM EjWriteRead1;
VAR
   Largo, Ancho, Alto, Volumen : integer;
Nombre : String[10] ;
Estatura : Real ;
BEGIN
   write('Escribe el largo, ancho y alto : ');
   read(Largo, Ancho, Alto);
   Volumen := Largo * Ancho * Alto ;
   write('El volumen es ');
   write(Volumen);
   read(Nombre);
   read(Estatura)
END.
```

5.3.2. WRITE y WRITELN

WRITE y WRITELN (contracción de WRITE LINE) constituyen las instrucciones básicas para las operaciones de *salida estándar*. Con estas instrucciones el programa puede entregar información del exterior, con la siguiente forma:

```
write ( lista de variables );
writeLn ( lista de variables );
```

en el que *lista de variables* puede ser el resultado de evaluar una expresión (que incluye constantes, variables, etc.), o los resultados de evaluar diferentes expresiones, separándolas mediante comas. El procedimiento WRITE permite que la siguiente instrucción se realice en la misma línea , mientras que WRITELN alimenta además una nueva línea, antes de finalizar.

Por ejemplo:

```
PROGRAM EjWrite;
VAR   Lado, Area : integer;
BEGIN
   write(5);
   write(5,3,2);
   writeLn;
   writeLn(4.53);
   Lado := 10;
   Area := Lado * Lado;
   write('Si el lado de un cuadrado es ', Lado);
   write(', el area es ', Area);
END.
```

que producirá la siguiente salida:

```
$ EjWrite
5532
4.53
Si el lado de un cuadrado es 10, el area es 100
```

Un valor booleano desplegará cualquiera de las cadenas : TRUE o FALSE, así:

```
writeLn('20 + 30 = ', 20 + 30 ,' ES ', 20 + 30 = 50);
writeLn(3.0);
```

producirá :

```
20 + 30 = 50 ES TRUE
3.0000000000E+00
```

dejando el cursor en la misma línea, al final del 3.0. Cuando un valor de salida se escribe sin una especificación de longitud de campo, se utilizará la especificación de campo por omisión. La especificación de longitud de campo por omisión dependerá del tipo de valor de salida y de la implementación de Pascal.

6. Estructuras de Control

6.1. Secuencia

En este caso, las instrucciones se ejecutan una después de la otra sin omitir ninguna de ellas. Tal como es el caso en muchos lenguajes de programación, la sintaxis para las instrucciones ejecutadas en secuencia es muy simple. Debe incorporarse un punto y coma y escribir la siguiente instrucción, en un esquema como el siguiente:

```
...
...
...
<instrucción 1>;
<instrucción 2>;
...
<instrucción n>;
...
...
```

6.2. Alternativa

Usualmente cuando escribimos un programa o un procedimiento, necesitamos elegir entre uno o dos cursos de acción, casos posible, en función de un valor particular de un dato. Las sentencias alternativas en el lenguaje Pascal se realiza con alguna de las dos siguientes formas:

- Sentencia IF, para el Condicional Simple.
- Sentencia CASE, para el Condicional Múltiple.

6.2.1. Sentencia IF

La sentencia de control **simple** tiene la siguiente sintaxis:

```
IF datoBoolean
THEN
  accion1
```

En este caso, si la condición booleana `datoBoolean` es verdadera, se ejecuta la acción `acción1`. Si la condición no es verdadera (es falsa), no hay ejecución de acciones en esta sentencia.

En esta instrucción *primero se evalua la condición booleana*, que puede tratarse de una variable booleana o una expresión con operaciones cuyo resultado es un dato boolean. Si la condición `datoBoolean` no brinda como resultado un valor del tipo boolean, el compilador indicará un error de sintaxis.

La *acción* es una instrucción en Pascal, que puede ser tanto simple como compuesta, en la que una *acción compuesta*, tendría el siguiente formato:

```
IF <expr. booleana>
THEN BEGIN
  <instrucción-1>;
  <instrucción-2>;
  ...
END;
```

También se destaca la instrucción **If simple** en la que se permite ejecutar una acción para el caso de que la condición sea falsa, que responde al siguiente formato:

```
IF datoBoolean
THEN
  accion1
ELSE
  accion2
```

Tal como en el caso anterior, las acciones (accion1 y/o accion2) pueden ser una instrucción o una secuencia de instrucciones como sentencia compuesta. Por ejemplo:

```
IF x > y THEN BEGIN
  max:= x;
  writeLn ('El máximo es ', x)
END
ELSE BEGIN
  max:= y;
  WriteLn('El máximo es ', y)
END;
```

para X con valor 1 y Y con valor 5, producirá :

```
El máximo es 5
```

Con esta estructura **If ... Then ... Else**[20] también se pueden realizar decisiones múltiples. Un modo muy frecuente de utilización es el encadenamiento de estructuras de estas estructuras de la forma:

```
IF datoBoolean1
THEN
  accion1
ELSE
  IF datoBoolean2
  THEN
    accion2
  ELSE
    IF datoBoolean3
    THEN
      accion3
    ...
```

El anidamiento de instrucciones if puede dar lugar a expresiones del tipo

```
IF cond1 THEN IF cond2 THEN I2 ELSE I3
```

que pueden ser de interpretación ambigua en el siguiente sentido: *¿a cuál de las dos instrucciones if pertenece la rama **else**?* En realidad la ambigüedad sólo existe en la interpretación humana, ya que la semántica de Pascal es clara: debe emparejarse la rama *else* con el *then* más próximo que no esté emparejado. Con ello, la interpretación anterior sin ambigüedad sería:

```
if C1 then begin if C2 then I2 else I3 end
```

6.2.2. Sentencia CASE

Es posible concatenar las construcciones **IF** de forma tal de permitir una selección a partir de una cantidad variable de posibilidades, es decir, generar una secuencia de selecciones que se comporta como una selección múltiple. Para ello el lenguaje Pascal cuenta también con la instrucción **CASE**, que permite generar secuencias múltiples, que posibilita elegir la instrucción a ejecutar en función del los distintos resultados de una expresión utilizada, que debe ser del tipo Entero o Caracter.

La instrucción cuenta con una expresión (llamada *selector*) y una lista de sentencias para cada una de las constantes del mismo tipo del selector, de forma que cuando se evalúa la expresión, se ejecuta la instrucción que contenga el o los valores correspondientes:

```
CASE <expresión> OF
  <valor-1>: <instrucción-1>;
  <valor-2>: <instrucción-2>;
  ...
  <valor-k>: <instrucción-k>;
END;
```

[20] Es importante tener en cuenta que después de toda instrucción debe incorporarse un punto y coma, pero no debe haber un punto y coma antes de un ELSE.

o bien

```
CASE <expresión> OF
  <valor-1>: <instrucción-1>;
  <valor-2>: <instrucción-2>;
  ...
  <valor-k>: <instrucción-k>;
ELSE
  <instrucción-n>;
END;
```

de forma que en caso de que la expresión tenga el *valor-i* se ejecutará la *instrucción-i*. Cuando está presente la sentencia *ELSE* sus instrucciones asociadas se ejecutan en caso de que la expresión se evalúa a un valor distinto de todos los valores listados.

Es posible además que la *instrucción-j* sea un bloque de la forma BEGIN-END y que incorpore más de una instrucción (en forma de lista o de subconjunto), y además, la ejecución de una instrucción coincida con más de un valor, que respondería al siguiente formato:

```
CASE <expresión> OF
  <valor1>, <valor2>: <instrucción-1>;
  <valor3>, <valor4>, <valor5>:
    BEGIN
      <instrucción-2a>;
      <instrucción-2b>;
    END;
  <valor6>: <instrucción-3>;
END
```

Por ejemplo, para un programa que escribe cuál es el último día según el número de mes que se ingresa como entrada, se podría escribir un código como el siguiente:

```
PROGRAM EjemploCase;
VAR
  Mes, CantDias : integer ;
BEGIN
  writeln('Ingresa el mes (De 1 a 12): ' );
  readln(Mes);
  CASE Mes OF :
    1, 3, 5, 7, 8, 10, 12 : CantDias := 31;
    4, 6, 9, 11 : CantDias := 30;
    2 : CantDias := 28;
  ELSE
    writeln('Error en la entrada);
END;
  write('El mes número ', Mes:2);
  writeln(' tiene ', CantDias:2 , ' días');
END.
```

y considerando la entrada como valor 6, la salida sería:

```
Ingresa el mes (De 1 a 12): 6
El mes número 6 tiene 30 días
```

Para el caso que se incorpore la posibilidad de que el año sea bisiesto, debería realizar modificaciones para el caso con valor Mes igual a 2, considerando el valor de año.

6.3. Iteración

Las instrucciones del tipo iterativas permiten especificar acciones que se ejecutan repetidamente, llamado usualmente *bucle*. El lenguaje PASCAL tiene tres instrucciones iterativas **WHILE**, **REPEAT UNTIL** y **FOR**, si bien todas ellas pueden ser especificadas utilizando sólo la instrucción *WHILE*.

6.3.1. Sentencia WHILE

La instrucción **WHILE** se utiliza para especificar una acción que se repite mientras es verdadera una determinada condición. La instrucción se ejecuta en dos pasos:

- en el primer paso se evalúa la condición
- en el segundo paso si la condición evaluada es verdadera se ejecutan las acciones del cuerpo del bucle y luego se vuelve al primer paso. Si la condición es falsa, se termina la instrucción.

Se espera que alguna de las instrucciones correspondientes al cuerpo del bucle modifiquen la condición evaluada, de lo contrario, si la condición siempre evalúa a verdadero, nos encontramos con un *bucle infinito* y el programa no pararía nunca.

La instrucción responde al siguiente formato:

```
WHILE <expresión booleana> DO
BEGIN
  <instrucción-1>;
  <instrucción-2>;
  ...
  <instrucción-K>
END;
```

Por ejemplo, el siguiente código de programa permite realizar el cálculo de la suma de los primeros n números, utilizando la instrucción iterativa WHILE:

```
...
ReadLn(n);
Suma:= 0;
Contador:= 1;
WHILE Contador <= n DO BEGIN
   Suma:= Suma + Contador;
   Contador:= Contador + 1
END; {while}
WriteLn(suma);
...
```

Como todas las instrucciones del lenguaje Pascal, la instrucción **WHILE** se puede anidar, de manera de tener dos bucles, uno interno y otro externo, de la siguiente forma:

```
WHILE <condición-1> DO BEGIN
   <instrucción-1>;
   ...
   WHILE <condición-2> DO BEGIN
      <instrucción-interna-1>;
      ...
      <instrucción-interna-k>;
   END;
   ...
   <instrucción-n>;
END;
```

Los bucles anidados son muy útiles cuando se escriben programas, especialmente cuando se trabaja con estructuras complejas, tales como archivos o matrices, para su manipulación en más de una dimensión.

6.3.2. Sentencia REPEAT

La instrucción **REPEAT** también se utiliza para construir bucles a partir de una determinada condición, y se ejecuta en tres pasos:

- en el primer paso se ejecutan las acciones del cuerpo del bucle
- en el segundo paso, se evalúa la condición
- en el tercer paso, si la condición evaluada es verdadera se vuelve al primer paso. Si la condición es falsa, se termina la instrucción.

La instrucción responde al siguiente formato:

```
REPEAT
   <instrucción-1>;
   ...
   <instrucción-k>
UNTIL <condición>;
```

Obsérvese que no es necesario las palabras reservadas BEGIN-END, ya que se asume un conjunto de instrucciones antes del UNTIL.

Una comparación del momento en el que se realiza la evaluación de la condición para las instrucciones iterativas, establece que la instrucción **WHILE** realiza el control *antes de ejecutar* las acciones, por lo que es posible que el bucle tenga 0 iteraciones. La instrucción **REPEAT** realiza el control *después de ejecutar* las acciones, por lo que esta instrucción permite bucles con al menos 1 iteración.

Por ejemplo, para el código que permite sumar los primeros n números, la versión con la instrucción **REPEAT** sería:

```
ReadLn(n);
suma:= 0;
contador:= 0;
REPEAT
   contador:= contador + 1;
   suma:= suma + contador
UNTIL contador = n;
```

Compare ambos códigos, y analice qué ocurre en el momento de realizar la inicialización de las variables, y cuáles deberían las condiciones necesarias para que el valor n no genere un bucle infinito.

6.3.3. Sentencia FOR

La instrucción **FOR** se utiliza para especificar un número predeterminado de repeticiones para una acción.

Como la instrucción iterativa más compleja, tiene el siguiente formato:

```
FOR <contador> := <expresión.1> TO <expresión.2> DO
   <instrucción>;
```

en el cual, el <contador> es una variable de tipo ordinal, y <expresión.1> y <expresión.2> expresiones del mismo tipo, que actúan como valor inicial y valor final, considerando que se ejecutarán las instrucciones del cuerpo del bucle **n** veces, con **n** la diferencia entre el valor inicial y el final.

La ejecución tiene el siguiente comportamiento en tres pasos:

- Paso 1: el <contador> se inicializa con el valor de la expresión <expresión.1>.
- Paso 2: se verifica si <contador> es mayor que el valor de la expresión <expresión.2>. Si es mayor, finaliza el ciclo. Si es menor, sigue en el Paso 3.

- Paso 3: se ejecuta la instrucción (simple o compuesta) <instrucción>. Se *incrementa* <contador> calculando su valor siguiente (por ejemplo, en una unidad si es entero, siguiente letra si es caracter, etc.). Se vuelve al Paso 2.

Si el objetivo del programa es *decrementar* el valor del contador, se utiliza la palabra reservada, en el siguiente formato (con instrucción compleja):

```
FOR <contador> := <expresión.1> DOWNTO <expresión.2> DO
BEGIN      <instrucción.1>;
  ...
  <instrucción.k>
END ;
```

en el que el valor de <expresión.2> debe ser menor o igual que el valor de <expresión.1> para que se ejecute alguna vez las instrucciones del bucle del ciclo.

Como ejemplo, el siguiente código de programa permite calcular el factorial de los números entre 1 y un número ingresado como entrada:

```
PROGRAM factorial;
VAR
  f, n, i: Integer;
BEGIN
  Read(n);
  f := 1;
  FOR i := 1 TO n DO
  BEGIN
    f := f * i;
    Writeln('El factorial de ',i,'es: ', f)
  END;
END.
```

que para la entrada 7 tendría la siguiente salida:

```
7
El factorial de 1 es: 1
El factorial de 2 es: 2
El factorial de 3 es: 6
El factorial de 4 es: 24
El factorial de 5 es: 120
El factorial de 6 es: 720
El factorial de 7 es: 5040
```

Por ejemplo, el código del siguiente programa permite escribir las letras del alfabeto en orden descendente.

```
PROGRAM alReves;
VAR
   c: Char;
BEGIN
   FOR c := 'Z' DOWNTO 'A' DO
      write(' ', c);
END.
```

con la salida

```
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

Si bien sería posible modificar el valor de la variable contador dentro del código del bucle, el objetivo de esta instrucción es permitir que el ciclo se ejecute una cantidad de veces conocidas en su inicio. De hecho algunas implementaciones del lenguaje no permiten asignar valores a la variable contador dentro del código del bucle.

7. Modularidad

A pesar de la corta historia de la computación, el modo de programar una computadora ha tenido grandes cambios, desde un arte, a una serie de principios generales que permiten conformar núcleos de conocimiento de una metodología de programación. Este objetivo de construir programas de calidad, refleja diversas características, entre ellas :

- la corrección, que el programa haga lo que tiene que hacer
- la comprensibilidad, que pueda ser comprendido por otras personas para mayor facilidad y comodidad en el mantenimiento.
- la eficiencia, adaptado a requerimientos de memoria y tiempo de ejecución.
- la flexibilidad, o capacidad de adaptación
- la portabilidad, como capacidad de usar el mismo programa en distintos sistemas sin grandes cambios.

El lenguaje Pascal se ha desarrollado con el sentido de abrazar las características antes enumeradas para lograr programas que sean *comprensibles, correctos, flexibles* y *transportables*, basadas en la **programación estructurada** y en la **programación con subprogramas**, útiles y con posibilidad de utilizarse en múltiples lenguajes.

La idea de la programación estructura, que fueron expuestos por Dijkstra en 1965, es una técnica de programación basada e refinamientos sucesivos que permite construir programas estructurados, generando nuevas estructuras paso a paso, con detalle de las acciones de sus componentes.

A través de la descomposición modular es posible escribir un programa en el lenguaje, concentrándose inicialmente en todo el problema, analizando el programa en acciones mayores, y detallando cada una de estas acciones en estructuras independientes una de las otras.

Estas acciones pueden ser especificadas mediante módulos, que a su ves pueden estar constituidos por otros módulos, sucesivamente, se denominan en forma general **subprogramas**.

7.1. Concepto, ventajas y desventajas

Un subprograma, en líneas generales, puede ser considerado como una caja negra, que recibe unos datos de entrada, los procesa, modificando algunos de ellos si es preciso, y genera unos datos de salida.

Entre sus ventajas se puede enunciar

- organización más clara del programa que facilita la comprensión del lector del código. Los subprogramas permiten eliminar detalles de la lógica global del programa, localizando dichos detalles en unidades independientes.
- se reduce la repetición de secuencias de instrucciones ya que se reutiliza el código de los subprogramas en distintas ocasiones
- se puede descomponer grandes programas en una conjunto de subprogramas más pequeños y manejables
- manejar unidades más pequeñas mejora la detección de errores.
- aumentar la legibilidad y modularidad permite el trabajo entre varios programadores

En el lenguaje Pascal se distinguen dos tipos de subprogramas

- **Procedimientos**, como módulos que se utilizan como nuevas instrucciones y no devuelven valores
- **Funciones**, como módulos que se utilizan en expresiones y que devuelven valores de distintos tipos.

Ambas tipos de módulos se escriben utilizan las construcciones que responden a las palabras claves *PROCEDURE* y *FUNCTION*, de manera tal que cada subprograma realizará una acción determinada.

Para su invocación, se utilizarán *parámetros* que resultan la base de su comunicación de entrada y, si bien no se constituye como una buena práctica, algunas veces también de salida.

Los parámetros (o argumentos) son identificadores que proveen un mecanismo de pase de información, y en su declaración se llaman *parámetros formales*. Al momento de su invocación, la instrucción referencia al subprograma escrito, con una lista de expresiones o valores llamados *parámetros actuales*.

7.2. Procedimientos

Un procedimiento en el lenguaje Pascal es un subprograma que realiza alguna tarea del programa, y no devuelve valor al subprograma que lo haya invocado. La sintaxis con la que se especifican

estos procedimientos es similar a la de un programa PASCAL pero se usa la palabra reservada **PROCEDURE** para especificar su *identificador o nombre* e indicar que se trata de un procedimiento parcial:

```
PROCEDURE <nombre del procedimiento> ( <parámetros formales> ) ;
  Parte declarativa
BEGIN
  Bloque de sentencias del procedimiento
END;
```

o como

```
PROCEDURE <nombre del procedimiento>;
  Parte declarativa
BEGIN
  Bloque de sentencias del procedimiento
END;
```

La *parte declarativa* del procedimiento tiene el mismo fin que en un programa y en ella pueden definirse tipos de datos, variables y constantes[21] como en un programa y también, si es necesarios, sus propios subprogramas en forma de procedimientos y funciones que sean necesarios para la realización de este PROCEDURE.

En el *bloque principal del procedimiento* se escriben las sentencias del lenguaje que constituyen las acciones a realizar, utilizando datos definidos en su parte declarativa, en la parte declarativa del programa principal o bien que se presentan como argumentos del procedimiento.

Dependiendo de la naturaleza del problema, la presencia de argumentos o parámetros será establecido por el programador. Nótese que *si no hay parámetros presentes, no deben indicarse los paréntesis.*

En la declaración del procedimiento los parámetros se especifican de un modo muy parecido a como se especifica el tipo de datos en la declaración de las variables de un programa. Los datos de un mismo tipo van separados por coma y se utiliza el punto y coma para separar datos de distinto tipo.

Por ejemplo, para mostrar junto a un mensaje el valor de resultados, puede escribirse el subprograma siguiente:

```
PROCEDURE imprimeResultados ( Resultado1, Resultado2: Real ) ;
BEGIN
  writeln('El resultado de la primera operación es: ', Resultado1);
  writeln('El resultado de la segunda operación es: ', Resultado2);
  writeln('El resultado de su suma es: ', Resultado1 + Resultado2);
END;
```

Al momento de realizar la invocación desde otro subprograma o el programa principal, se utilizará la siguiente sintaxis:

```
<nombre del procedimiento> ( <parámetros actuales> );
```

[21] Algunas implementaciones del lenguaje establecen limitaciones en cuanto a qué puede definirse en un subprograma

y para el caso del subprograma escrito, se invocaría como:

```
...
VAR a, b: Real;    ...
a := <cálculo del primer resultado>
...
b := <cálculo del primer resultado>
...
imprimeResultados (a*2, b/2);
...
```

y asumiendo que el valor de la variable a es 10.4, y el de la variable b es 15, la salida sería:

```
El resultado de la primera operacion es: 2.0800000000000001E+001
El resultado de la segunda operacion es: 7.5000000000000000E+000
El resultado de su suma es: 2.8300000000000001E+001
```

Pascal permite el uso de la palabra reservada VAR en la definición de los parámetros formales, para que los valores de los parámetros resulten modificados si se realizan asignaciones en el cuerpo de un subprograma.

Por ejemplo, se define el subprograma *intercambio* que permite el cambio de los valores de dos parámetros, pasados al momento de la invocación:

```
PROCEDURE intercambio (VAR x,y:integer);
  VAR
    aux : integer;
BEGIN
  aux := x;
  x := y;
  y := aux;
END;
```

y en el código siguiente de parte de un programa:

```
...
VAR a, b: integer;    ...
writeln('Valores antes del intercambio: ', a, ' y ', b);
intercambio(a,b);
writeln('Valores después del intercambio :', a, ' y ', b);
...
```

resultaría en una salida parcial como:

```
  ...
Valores antes del intercambio: 4 y 6
Valores después del intercambio: 6 y 4
  ...
```

En el código de *intercambio* se distingue la definición de una variable local al procedimiento, la variable *aux* que sólo puede ser utilizada dentro del procedimiento. Esta variable se *crea* cada vez

que se invoca el subprograma y se *destruye* cada vez que termina su ejecución.

7.3. Funciones

Se utilizan de una manera similar a los procedimientos, si bien su principal diferencia es que el nombre o identificador de la función asume un valor, y cuando se ejecutan las acciones de una función, se devuelve el valor de salida al subprograma o módulo que lo invocó.

Las funciones se utilizan en una expresión, no pueden ser invocadas como sentencias, y como deben devolver un valor, es necesario especificarle un tipo de dato asociado. Las declaraciones de una función responden a la siguiente forma:

```
FUNCTION <nombre de la función> ( <parámetros formales> ): <tipo> ;
  Parte declarativa
BEGIN
  Bloque de sentencias de la función
  <nombre de la función> := resultado de la función
END;
```

o como

```
FUNCTION <nombre de la función> : <tipo> ;
  Parte declarativa
BEGIN
  Bloque de sentencias de la función
  <nombre de la función> := resultado de la función
END;
```

El lenguaje Pascal utiliza el propio nombre de la función del lado izquierdo de una asignación para establecer el valor de salida o de retorno de la misma.

Como primer ejemplo, el cálculo de la función factorial resultaría en el siguiente código:

```
FUNCTION factorial (n:integer) : integer ;
  VAR
    i, auxiliar : integer;
BEGIN
  auxiliar := 1;
  FOR i:= 1 TO n DO
    auxiliar := i * auxiliar;
  factorial := auxiliar;
END;
```

Por ejemplo la siguiente función permite calcular el valor de la potencia entera de una número real:

```
FUNCTION potencia( base: real; exponente: integer): Real;
  VAR
    i : integer;
    r : real;
BEGIN
  IF exponente <= 0
    THEN potencia := 1.0
  ELSE IF exponente = 1
       THEN potencia := base
    ELSE BEGIN
      r := base
      FOR i:= 2 TO exponente DO
        r := base * r;
  END;
  THEN potencia := r
END;
```

Para estos casos, sería sentencias válidas en las que se utilizan las funciones definidas las siguientes:

```
a := 3 * factorial(6);
writeln( 'Factorial de 3 = ', factorial(3) );
f := factorial(n) / factorial(m) / factorial(m-n);
valor := potencia(3.65, 6) / factorial(n);
```

Notar que es necesario utilizar las funciones como un operando más en una expresión, con un tratamiento similar a una variable del tipo con el que fue definida la Función, que siempre devuelve un valor. En el caso de que la función no tenga parámetros, no deben utilizarse paréntesis.

7.4. Ámbito de las variables y de los identificadores

Se denominan *variables globales* a las variables que están definidas en el programa principal. Es decir, las declaradas en la zona **VAR** del Programa Principal.

Se denominan *variables locales* a las variables que sólo tienen validez en el interior de un subprograma. Es decir las variables locales declaradas explícitamente en cada procedimiento o función.

Definimos el *ámbito de una variable* el espacio dentro del código de un programa en el que se puede utilizar una determinada variable. El ámbito de las variables globales es el programa entero. El ámbito de una variable determinada es la del bloque en la que está declarada, así como en todos los subbloques que contiene.

Estos conceptos son *extensibles* a otros elementos presentes en el código de un lenguaje que se identifican con *identificadores o nombres*, tales como *constantes*, *tipos*, *parámetros*, *subprogramas*, etc.

Para el caso de que una variable local de un subprograma comparta el nombre con una variable o parámetro de un subprograma que la contenga, la variable del subprograma actual *esconderá* el acceso a los otros datos en otros módulos del programa.

Por ejemplo, sea el siguiente código de programa en el que se comparten nombres de variables:

```
PROGRAM verAlcance;
  VAR a: integer;
PROCEDURE cambia ( a : integer);
  BEGIN
    a := 10;
    writeln( ' Dentro: ', a)
  END;
BEGIN
  a := 3;
  writeln( ' Antes: ', a)
  cambia(a);
  writeln( ' Después: ', a)
END.
```

el identificador *a* refiere a dos variables distintas: la variable global, propia del programa *verAlcance*, y el parámetro formal (local) del procedimiento *cambia*.

Durante la ejecución del programa principal, el uso del nombre *a* referencia a la variable global, pero cuando se invoca al procedimiento *cambia*, en su código el nombre *a* referencia al único parámetro formal, y la variable global con el mismo nombre queda *oculta* para el subprograma, ya que no es posible, en el lenguaje Pascal, referenciar a dos elementos con el mismo nombre.

Este programa tendría la siguiente salida:

```
Antes: 3
Dentro: 10
Despues: 3
```

donde claramente se vislumbra que el valor 3 corresponde a la variable global *a*, mientras que el valor 10 pertenece al nombre del elemento local, en este caso un parámetro, del procedimiento *cambia*.

Obsérvese que si se utilizara el nombre, por ejemplo de una variable que no haya sido definida en forma local, se accederá a ese elemento.

Por ejemplo, si se modifica el código del programa anterior, de forma de incorporar una nueva variable global, con nombre *b*, resultaría como:

```
PROGRAM verAlcance;
  VAR a, b: integer;
PROCEDURE cambia ( a : integer);
  BEGIN
    a := 10;
    writeln( ' Dentro a: ', a)
    b := 8;
    writeln( ' Dentro b: ', b)
  END;
BEGIN
  a := 3;
  b := 5;
  writeln( ' Antes a: ', a)
  writeln( ' Antes b: ', b)
  cambia(a);
  writeln( ' Después a: ', a)
  writeln( ' Después b: ', b)
END.
```

tendrá un comportamiento según la siguiente salida:

```
Antes a: 3
Antes b: 5
Dentro a: 10
Dentro b: 8
Despues a: 3
Despues b: 8
```

ya que la modificación de la variable en el código de *cambia* ha accedido al mismo elemento definido en el programa principal.

7.5. Recursividad

Se denomina *recursión* o *recursividad* a la forma en la cual se especifica un proceso o un dato basado en su propia definición. El lenguaje Pascal soporta recursividad, es decir, que los subprogramas, tanto procedimientos como funciones, además de invocar a otros subprogramas, pueden invocarse a si mismos. Una invocación de este tipo se llama *invocación recursiva*.

Un ejemplo muy utilizado para definir una función recursiva es el cálculo del factorial de un número entero no negativo, partiendo de las definiciones de la matemática en las siguientes expresiones:

$$0! = 1$$

$$1! = 1$$

$$n! = n \times (n-1)!$$

de manera tal que consideraríamos una nueva versión de la función factorial definida en forma recursiva como:

```
FUNCTION factorial2 (n:integer) : integer ;
  VAR
    i, auxiliar : integer;
BEGIN
  IF n = 0
    THEN factorial2 := 1;
    ELSE factorial2 := n * factorial2(n-1);
END;
```

Vale considerar las dos *versiones* del identificador *factorial2* presentes en el código, considerando la línea del ELSE.

El identificador *factorial2* del **lado izquierdo** de la asignación cumple el cometido de servir para establecer el valor de salida o retorno de la función. Mientras que el identificador del **lado derecho** de la asignación se utiliza para realizar la invocación recursiva de la función *factorial2*. Nótese la diferencia en el uso con la presencia de los paréntesis para invocar la función con parámetros. De todas maneras, si la función no hubiera requerido parámetros, no se utilizarían los paréntesis, con una presentación exactamente igual, y sólo puede distinguirse su uso según el lado de la asignación en la que se encuentra.

De la misma manera que con las funciones, es posible utilizar procedimientos con invocación recursiva. Veamos el siguiente ejemplo que permite escribir el abecedario utiizando un procedimiento recursivo.

```
PROGRAM usoRecursivo;
  VAR a: char;
PROCEDURE abeceRecursivo (c: char) ;
  BEGIN
    write( c, ' ');
    IF c <= 'Z'
      THEN abeceRecursivo( succ(c))
  END;
BEGIN
  a := 'A';
  abeceRecursivo(a)
END.
```

que resultará en una salida muy similar a la del programa definido utilizando estructuras iterativas.

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

8. Tipos de Datos definidos por el usuario

En el lenguaje Pascal se pueden definir dos tipos de datos a partir de datos simples: los *enumerados* y los *subrangos*.

8.1. Enumerados

Los enumerados son tipos de datos que se definen como un conjunto de valores válidos para un determinado tipo. Para declarar un tipo de datos enumerado, simplemente se enumeran los posibles valores que puede tener una variable para un tipo de datos.

Como ejemplo podemos considerar los colores del arco iris, los ingredientes para poder preparar una pizza, los días de la semana, etc.

Por ejemplo, para definir un tipo que incorpore los valores de los días de la semana, se puede definir como:

```
TYPE
     tipoDiasSemana = ( lunes, martes, miercoles, jueves, viernes, sabado,
                        domingo)
```

Tenga en cuenta que cada uno de los valores posibles del tipo **tipoDiasSemana** es un identificador. Luego, se pueden definir variables cuyos valores puedan pertenecer al conjunto de días de la semana y usarlos dentro del código de un programa o subprograma:

```
  VAR
    ayer, hoy, laboral : tipoDiasSemana;
    contador: integer; ...
  BEGIN
    ...
    ayer := miercoles;
    hoy := succ(ayer);
    ...
  END.
```

Como son tipos ordinales, tales como los *enteros*, *booleanos* y *caracteres* también heredan sus operadores, por lo que pueden generarse expresiones válidas para incorporar a código de programas, tales como los siguientes ejemplos:

```
  succ(miercoles)          { Resultado jueves }
  ( martes < lunes )       { Resultado FALSE }
  ( pred(lunes) )          { Resultado ERROR }
  ( ord(miercoles) )       { Resultado 2 }
```

No se pueden repetir valores en distintas descripciones de tipos enumerados ya que, en tal caso, su tipo sería ambiguo.

8.2. Subrango

Los datos de tipo subrango se definen tomando como límites, el valor mínimo y el valor máximo del rango, como dos constantes. Las constantes puede ser del tipo *entero*, *caracter* o *enumerado*.

Por ejemplo, pueden definirse:

```
CONST
  MaximoM = 10 ;
VAR
  valores : 1 .. MaximoM
  ...
BEGIN
  ...
  valores := 1;
  write(valores);      ...
END.
```

Los valores de la constantes que vienen dadas pertenecen al mismo tipo ordinal que delimitan el intervalo, y siempre tienen que respetar un orden creciente, es decir que para que el intervalo tenga sentido es necesario que $Ord(ConstanteInicial) \leq Ord(ConstanteFinal)$ donde *Constante Inicial* es el extremo inferior del subrango, y *Constante Final* el extremo superior.

```
TYPE
  enumMes = (ene, feb, mar, abr, may, jun, jul, ago, sep, oct,
             nov, dic);
VAR
  mesesInvierno : jun .. sep ;
  contadorMes : enumMes;
  ...
FUNCTION escribeMes ( mes: enumMes ): String;
BEGIN
  CASE mes OF
    ene : escribeMes := 'Enero';
    feb : escribeMes := 'Febrero';
    mar : escribeMes := 'Marzo';
    abr : escribeMes := 'Abril';
    dic : escribeMes := 'Diciembre';
    ELSE escribeMes := 'Otro mes';
  END;
END;
  ...
BEGIN
  FOR contadorMes := ene TO dic DO
    writeln(escribeMes(contadorMes));
END.
```

8.3. Conjuntos

El tipo de datos definido por el usuario *conjunto* permite agrupar los los elementos de un conjunto de valores, que pueden pertenecer a cualquier tipo ordinal, incluido los enumerados.

Aunque se puede utilizar notación de tipo subrango para especificar secuencia de valores que pertenezcan a un conjunto, los elementos del conjunto no tienen una ordenación interna particular. La única relación entre los miembros de un conjunto es: *existe* o *no existe* en el conjunto.

Por ejemplo los siguientes son conjuntos en su notación extensiva y comprensiva:

- $[1, 3, 5]$ es el conjunto de los tres primeros enteros impares.
- [1.,3, 8.,10] es el conjunto de seis elementos constituidos por los números mayores o iguales que 1 y menores o iguales que 3 y los números mayores o iguales que 8 y menores o iguales que 10

Considerando el ejemplo anterior de los meses con el enumerado definido *enumMeses*, puede definirse:

```
Estacion = Set Of Meses;
VAR
   n : Integer;
   mes : Meses;
   otogno, invierno, primavera, verano, cambio: Estacion;
BEGIN
   verano := [dic, ene .. mar ];
   otogno := [mar .. jun ];
   invierno := [jun .. sep ];
   primavera := [sep .. dic ];
   ...
```

El tipo de datos *conjunto* posee los siguientes operadores sobre los objetos de datos del tipo:

Operación	**Notación Pascal**	**Resultado**
Pertenece	IN	*a IN Conjunto* es verdadero si el valor del elemento *a* está presente entre los valores del conjunto *Conjunto*
Intersección	*	*A*B* es el conjunto cuyos elementos pertenecen a A y B simultáneamente.
Unión	+	*A+B* es el conjunto que contiene todos los elementos que están en A, en B o en ambos.
Diferencia	–	*A – B* es el conjunto cuyos elementos son de A pero no de B.

Por ejemplo, se puede escribir un programa que permita agrupar en el conjunto **setcod** los códigos válidos ingresados por teclado:

```
TYPE
   codigos = SET OF 1..50;
VAR
   setcod : codigos;
   codigo : 0..50;
   ...
BEGIN
```

```
  ...
  setcod := [];
  read(codigo);
  WHILE codigo <> 0 DO
  BEGIN
    read(codigo);
    setcod := setcod + [codigo]
  END;
  ...
  FOR codigo:= 1 TO 50 DO
    IF codigo IN setcod THEN
      ...
  ...
END.
```

Otro ejemplo[22] para el manejo de conjunto de caracteres

```
PROGRAM OtroEjemploSet;
VAR
  minusculasValidas, mayusculasValidas, letrasValidas: SET OF char;
  letra: char;

BEGIN
  minusculasValidas := ['a', 'b', 'c', 'd'];
  mayusculasValidas := ['F', 'H', 'K', 'M'];
  letrasValidas := minusculasValidas + mayusculasValidas;
  REPEAT
    write ( 'Introduce una letra...' );
    readLn( letra );
    IF NOT (letra IN letrasValidas) THEN
      writeLn('No aceptada!');
  UNTIL letra IN letrasValidas;
END.
```

8.4. Arreglos

Cuando se utiliza un conjunto de elementos de un mismo tipo y es necesario poder acceder a cada uno de ellos en forma ordenada, se puede utilizar el tipo de datos *arreglo*.

Un arreglo está formado por un número fijo de elementos contíguos de un mismo tipo. Al tipo se le llama *tipo base* del arreglo. Los datos individuales se llaman *elementos* del arreglo. Para definir un tipo estructurado arreglo, se debe especificar el *tipo base* y el *número de elementos*[23].

Para el manejo de los arreglos, Pascal clasifica a los *arrays* en dos tipos:

[22] Tomado de la red

[23] Si bien algunos dialectos del lenguaje de programación Pascal permiten la manipulación de arreglos cuyos tamaños pueden cambiar dinámicamente, el lenguaje Pascal presente a los arreglos como tipos de datos estructurados de carácter estático. De esta forma sólo pueden manipularse estructuras con un número de elementos conocido *a-priori*.

- **arreglos unidimensionales**, llamados también *vectores*, constituidos por un conjunto ordenado de elementos del tipo base.
- **arreglos multidimensionales**, que según la cantidad de dimensiones pueden llamarse *matrices*, *cubos*, *hipercubos*, etc, y que poseen un ordenamiento en varias dimensiones.

Para caracterizar cada una de las dimensiones de los arreglos, el lenguaje Pascal utiliza sendos *rangos* o *subrangos*, que definen la cantidad y el número de componentes para cada dimensión. Para separar cada dimensión, deben separarse los rangos entre comas al momento de su definición.

Por ejemplo:

```
TYPE
   tipoArreglo = ARRAY [ 1 .. MaxElementos] OF real;
   tipoMatriz = ARRAY [ 1 .. 10 , ene .. abr ] OF integer
...
VAR
   arrM, arrN : tipoArreglo;
   matA, matB : tipoMatriz;
   cubo : ARRAY [1 .. 3 , 1 .. 4 , -2 .. 2] OF char;
```

Técnicamente puede definirse a un arreglo del tipo base como el producto cartesiano de los dominios de los tipos de los índices. Así, los arreglos multidimensionales pueden definirse de varias formas:

- Como un vector de vectores:

```
TYPE
   tMatriz = ARRAY [1 .. 31] of ARRAY [ene .. dic] of real;
```

- Como un vector de un tipo definido antes:

```
TYPE
   tVector = ARRAY [1 .. 31] of real;
   tMatriz = ARRAY [ene .. dic] of tVector;
```

- Con rangos separados por comas

```
TYPE
   tVector = ARRAY [1 .. 31, ene .. dic] of real;
```

8.4.1. Operaciones de Arreglos

La *selección* (o referenciación) de cada elemento del arreglo se realiza con una *expresión* para referenciar cada índice de cada dimensión, con un formato del estilo *arreglo[expresión1, expresión2, .. expresiónN]* para un arreglo de N dimensiones. Por ejemplo:

```
v[3]
m[2, 3+i]
c[succ(enero), a+b, 'b']
```

La *asignación* se puede realizar por un lado para cada uno de los elementos, y por otro para la estructura completa, siempre que los arreglos de la operación tengan exactamente la misma estructura.

```
m[2,5] := a+1;
vector1 := vector5;
```

Para las operaciones de *comparación* en el lenguaje Pascal es necesario realizar la comparación elemento a elemento, comprobando la igualdad, desigualdad, etc., entre ellos.

Si bien los arreglos pueden pasarse como parámetros en procedimientos y funciones, el tipo de datos que devuelve una función no puede ser un arreglo, ya que un arreglo es un tipo estructurado. Usualmente para resolver este problema, puede utilizarse un parámetro adicional por variable para recuperar el resultado que se pretendía devolver con la función.

Presentamos a continuación el código de un programa en el que se utiliza el tipo estructurado *arreglo*:

```
PROGRAM productoEscalar;
CONST
  tamanio = 10;
VAR
  vector1, vector2: ARRAY [1..tamanio] of real;
  i: integer;
  resultado: real;
BEGIN
  writeLn('Ingrese primer vector: ');
  FOR i:= 1 TO tamanio DO
    readLn(vector1[i]);
  writeLn('Ingrese segundo vector: ');
  FOR i:= 1 TO tamanio DO
    readLn(vector2[i]);
  resultado := 0;
  FOR i:= 1 TO tamanio DO
    resultado := resultado + vector1[i]*vector2[i];
  writeLn ('Resultado ', resultado)
END.
```

8.5. Registros

Un *registro* es una estructura de datos que consiste de un número *fijo* de componentes llamados *campos*. Los campos pueden pertenecer a diferentes tipos y requieren de un *identificador de campo*.

Para definir un tipo *registro* se utiliza la palabra reservada *RECORD* seguida de una lista de campos y terminada por la palabra reservada *END*:

```
TYPE
  tipoRegistro = RECORD
    listaId1 : tipo1;
    listaId2 : tipo2;
    ...
    listaIdN : tipoN;
END;
```

en el cual, *tipoRegistro* es el nombre de la estructura del nuevo tipo creado, y *listaId(i)* es una lista de uno o más campos, separados por comas, que pueden pertenecer a un tipo prefedefinido o un tipo definido por el usuario, para cada *tipo(i)*.

Por ejemplo, para gestionar los números complejos, se puede definir el tipo *numeroComplejo* como

```
TYPE
  TipoComplejo = RECORD
      ParteReal: real;
     ParteImaginaria: real;
  END;
VAR
  Numero1 , Numero2 : TipoComplejo;
  Numero3 : real;
BEGIN
  Numero1.ParteReal := 5.5;
  Numero1.ParteImag := 2.0;
  Numero2.ParteReal := 0.0;
  Numero2.ParteImag := Numero3 * 5;
  ...
END.
```

o

```
PROGRAM registrosFecha;
  TYPE
    Fecha = RECORD
      mes : 0 .. 12; { 0 indica que no se conoce la fecha}
      dia : 1 .. 31;
      agno : Integer ;
    END;
  VAR
    alta , baja : Fecha;
```

8.5.1. Registros Variantes

El lenguaje Pascal provee además la alternativa de trabajar con *registros con variantes*, que constan de dos partes: la primera, llamada *parte fija*, está formada por aquellos campos del registro que forman parte de todos los ejemplares; la segunda parte, llamada *parte variable*, está formada por aquellos campos que sólo forman parte de algunos ejemplares.

La programación utilizando los registros variantes está más allá del alcance de este apunte y se deja como material para desarrollar en una futura segunda parte.

8.5.2. Operaciones de Registros

La *selección* (o referenciación) de cada elemento del registro se realiza con el *identificador de campo* para referenciar el campo especificado, como fue indicado en los ejemplos anteriores.

Para las operaciones de *asignación* el razonamiento es similar a los arreglos, elemento a elemento o directamente toda la estructura, siempre que los elementos a asignar compartan exactamente la misma estructura.

Por ejemplo, para la definición de tipos de los números complejos, ver:

```
Numero1 := Numero2 ;
Numero2.ParteReal := Numero3;
```

8.6. Combinación de los datos arreglos y registros

Tomando como ventaja la ortogonalidad del lenguaje de Programación Pascal, de la misma manera que es posible anidar instrucciones estructuradas, también se pueden anidar tipos de datos estructurados.

Por ejemplo, se podría hacer la declaración de 100 números complejos, tomando la declaración de *TipoComplejo* ya presentada en 44, como:

```
TYPE
  vectorComplejo = ARRAY [1..100] of TipoComplejo;
```

y realizar operaciones complejas sobre esta estructura.

Por ejemplo, la inicialización de una variable del tipo *vectorComplejo* se podría realizar con el siguiente código de programa:

```
FOR i:= 1 TO 100 DO
BEGIN
  Vector1[i].ParteReal := 0;
  Vector1[i].ParteImag := 0;
END;
```

8.7. Archivos

Un *archivo* o fichero en Pascal se estructura como una secuencia homogénea de datos, de tamaño no fijado de antemano, la cual se puede representar como una fila de celdas en las que se almacenan los datos componentes del archivo. Una marca especial llamada *fin de archivo* señala el fin de esta secuencia.

Usualmente, un *archivo* como conjunto de datos reside en un dispositivo de almacenamiento externos, tales como un disco duro, una memoria flash, etc. Por residir justamente en un dispositivo

externo, una vez que el programa ha concluido su ejecución o se ha apagado la computadora, los datos del archivo permanecen almacenados y es posible recuperarlos posteriormente.

Los archivos tienen dos formas de acceso:

- **acceso secuencial**: los datos se leen en forma consecutiva, comenzando por el primer elementos del archivo.
- **acceso directo** (a veces llamado acceso aleatorio o random) de forma tal que el programador y el programa determinan el orden en el que los datos se acceden.

Por otro lado, los archivos tienen dos tipos:

- **archivos de texto**: archivos que pueden crearse con editores de texto y también modificar directamente su contenido.
- **archivos binarios**: pueden almacenar gran cantidad de datos, con lectura y escritura rápida de datos, solo pueden accederse desde un programa con la programación adecuada.

8.7.1. Declaración de Archivos

Es posible crear un archivo de valores enteros, declarándose como:

```
VAR
  FichPrueba : FILE OF integer;
  DatoEntero : integer;
```

o un archivo de datos del tipo registro, muy habitual para el manejo de información, tales como[24]:

```
TYPE
  TipoPersona = RECORD
    Dni : ARRAY [1..8] of integer;
    Nombre : ARRAY [1..16] of char;
    Apellido : ARRAY [1..16] of char;
    Edad : 0..125;
  END;
VAR
  FichEmpleados : FILE OF TipoPersona;
  Persona : TipoPersona;
```

8.7.2. Operaciones con Archivos

Las operaciones más habituales e importantes que pueden hacerse con los archivos son la *escritura* y la *lectura* de sus componentes, que se realizan de una manera muy similar a la lectura y escritura usual, con las instrucciones `read` y `write`), pero redireccionadas al archivo adecuado.

Tipos de Operaciones sobre archivos:

[24]Para las cadenas de caracteres puede utilizar el tipo STRING en vez de arreglos de caracteres.

- **Asignación a un archivo externo**: establece la vía de comunicación entre el programa y el archivo almacenado en el dispositivo externo. se realiza con la instrucción *assign*, con los parámetros del nombre interno y el nombre del archivo (en el disco). Por ejemplo las asignaciones válidas:

```
ASSIGN (ArchivoClientes, 'CLIENTES.DAT');
ASSIGN (ArchivoPrueba, '/home/usuario/Documentos/datos.dat');
```

- **Apertura de un archivo**: permite al acceso a un archivo para su lectura y/o escritura. Se puede abrir un archivo de tres maneras:

  1. de sólo lectura: se permite que los datos del archivo sean leídos, siempre que el archivo exista, con la instrucción *reset* por ejemplo:

```
RESET (ArchivoClientes);
RESET (ArchivoPrueba);
```

  2. de sólo escritura: se abre así si el archivo no existe y es necesario crearlo, o si en caso de existir, la intención es sobreescribirlo con nuevo datos. Se usa la instrucción *rewrite* como:

```
REWRITE (ArchivoClientes);
REWRITE (ArchivoPrueba);
```

  3. de agregado: se abre así un archivo existente sobre el que se pretende agregar datos al final del fichero, con la instrucción *append* como:

```
APPEND (ArchivoClientes);
APPEND (ArchivoPrueba);
```

- **Lectura de un registro de archivo**: permite traer el dato desde el archivo al programa, que sólo es posible si se realizó la apertura de *modo sólo lectura*. Por ejemplo:

```
read (ArchivoClientes, Persona);
read (ArchivoPrueba, DatoEntero);
```

- **Escritura de un registro de archivo**: permite escribir el dato desde el programa al archivo, que funciona si se realizó la apertura de *modo sólo escritura o de agregado*. Por ejemplo:

```
write (ArchivoClientes, Persona);
write (ArchivoPrueba, DatoEntero);
```

- **Cerrar un archivo**: no permite operaciones con el archivo, si bien no deshace la asignación ya generada, por ejemplo:

```
CLOSE (ArchivoClientes);
CLOSE (ArchivoPrueba);
```

Así, uno de los principales problemas del manejo de archivo en el lenguaje Pascal es que no se pueden alternar las operaciones de lectura y escritura en un archivo, por lo que en caso de que se desee modificar un archivo, es necesario leerlo completo para escribirlo nuevamente modificado. Algunos dialectos de Pascal, tales como el *Turbo Pascal*, *Free Pascal* u otros, incorporan otras instrucciones que permiten manipular las operaciones de lectura y escritura en forma combinada, usualmente el mismo tipo de sólo lectura como operación combinada utilizando también la instrucción *reset*.

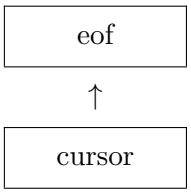
A continuación se presenta el código de un programa sencillo, en el que se muestra la creación y escritura de datos enteros en un archivo binario. El archivo de salida se llamará *UNO.SAL* según se

asigna, que se creará con la sentencia *rewrite*, se escribirán 4 números enteros, y se cerrará antes de finalizar la ejecución.

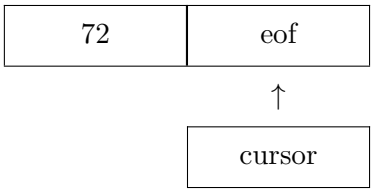
```
PROGRAM Uno;
VAR
  resguardo : FILE OF integer;
  num1, num2, num3, num4 : integer;
BEGIN
  num1 := 72; num2 := 79;
  num3 := 76; num4 := 65;
  ASSIGN (almacen,'UNO.SAL');
  REWRITE (resguardo);
  write (resguardo, num1);
  write (resguardo, num2);
  write (resguardo, num3);
  write (resguardo, num4);
  CLOSE (resguardo)
END.
```

Gráficamente, para analizar el comportamiento del programa Uno, se puede informar:

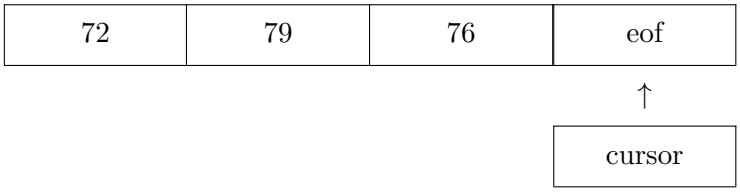
- Posición del cursor antes de realizar alguna escritura



- Posición del cursor después de realizar la escritura de la variable num1



- Posición del cursor después de realizar la escritura de la variable num3



Para leer el mismo archivo e imprimir todos sus números, se puede escribir el siguiente programa:

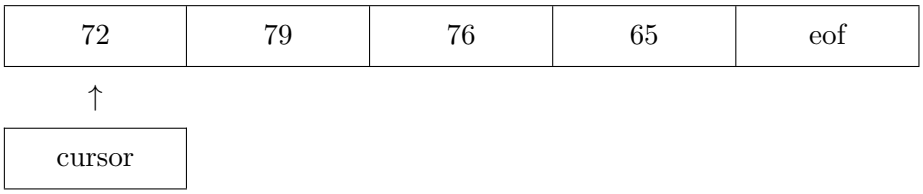
```
PROGRAM Dos;
VAR
   resguardo : FILE OF integer;
   num : integer;
BEGIN
   ASSIGN (almacen,'UNO.SAL');
   RESET (resguardo);
   WHILE not(EOF(resguardo)) DO
   BEGIN
      read (resguardo, num);
      write (num); (* salida por pantalla *)
   END;
   CLOSE (resguardo)
END.
```

Por convención se asume que el último elemento del archivo es un símbolo especial llamado *fin de archivo* que puede verificarse con la función *eof*, que devuelve verdadero si se alcanzó el fin de archivo y falso en caso contrario.

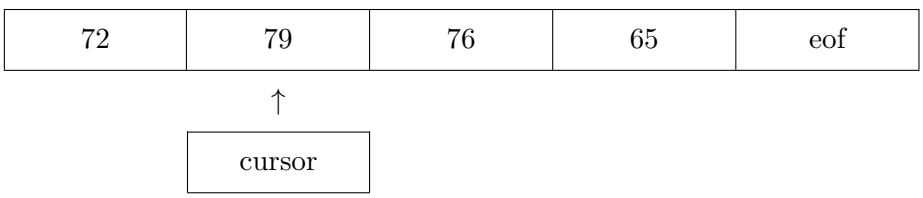
Para el programa **Dos**, la apertura del archivo en modo sólo lectura posiciona un cursor en el primer elemento del archivo, y en la medida que se realizan operaciones de lectura, el cursor se va moviendo de una unidad, hasta alcanzar el último elemento del archivo. Este último elemento puede ser desconocido, teniendo en cuenta que el archivo puede haber sido trabajado por otro programa o en otra ocasión y puede resultar difícil de determinar el número de elementos. Caso contrario, si supiera con precisión que el archivo tiene 4 elementos, podría haber leído los datos con 4 variables, o utilizar un FOR con un contador en un rango entre 1 y 4.

Gráficamente, para analizar el comportamiento del programa Dos, se puede informar:

- Posición del cursor antes de realizar alguna lectura



- Posición del cursor después de realizar la primera lectura



Se presenta un programa que permite generar un archivo ordenado como la mezcla de dos archivos ordenado de lectura.

```
PROGRAM Mezcla;
(* ************************************************
  Este programa mezcla dos ficheros de numeros
  enteros en un tercer fichero.
    Archivos de entrada : 'Arch1.DAT' y 'Arch2.DAT'
    Archivo de salida : 'Salida.DAT'
  ************************************************ *)
  VAR
    ArchivoEnt1, ArchivoEnt2 : FILE OF integer;
    ArchivoSalida : FILE OF integer;
    Dato1, Dato2 : integer;
  BEGIN   (* Asignación de los archivos de entrada y salida *)
    ASSIGN (ArchivoEnt1,'Arch1.DAT');
    ASSIGN (ArchivoEnt2,'Arch12.DAT');
    ASSIGN (ArchivoSalida,'Salida.DAT');
  (* Apertura de los archivos *)
    RESET (ArchivoEnt1);
    RESET (ArchivoEnt2);
    REWRITE (ArchivoSalida);
  (* Se mezclan los archivos hasta que quede uno vacío *)
    WHILE (not eof(ArchivoEnt1)) AND (not eof(ArchivoEnt1)) DO BEGIN
      read (ArchivoEnt1, Dato1);
      read (ArchivoEnt2, Dato2);
      IF (Dato1 < Dato2) THEN BEGIN
        write (ArchivoSalida, Dato1);
        read(ArchivoEnt1, Dato1);
      END
      ELSE BEGIN
        write(ArchivoSalida,Dato2);
        read(ArchivoEnt2, Dato2);
      END;
    END;
  (* Se copia el resto de ArchivoEnt1 si quedan datos *)
    WHILE not eof(ArchivoEnt1) DO BEGIN
      write(ArchivoSalida,Dato1);
      read(ArchivoEnt1, Dato1);
    END;
  (* Se copia el resto de ArchivoEnt2 si quedan datos *)
    WHILE not eof(ArchivoEnt2) DO BEGIN
      write(ArchivoSalida, Dato2);
      read(ArchivoEnt2, Dato2);
    END;
  END.
```

8.7.3. Otras Operaciones

Para el acceso directo, existen otras operaciones para los archivos binarios, algunas de las cuales se presentan a continuación:

Operación	Acción
SEEK(f,n)	Permite posicionarse en el componente determinado ubicado en la posición *n* del archivo *f*.
FILEPOS(f)	Brinda el número del componente actual del archivo *f*
FILESIZE(f)	Informa la cantidad de componentes del archivo *f*

8.8. Otros tipos de Datos

El lenguaje Pascal posee otros tipos de datos, entre los que se destacan los *punteros* para la manipulación de estructuras dinámicas. Un puntero es un tipo de datos cuyo valor es directamente una dirección de memoria.

En el presente apunte no se avanzará sobre las estructuras dinámicas del lenguaje, que se trabajará en un próximo apunte.

9. Apéndices

9.1. Palabras Reservadas

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

9.2. Precedencia de Operadores

Precedencia	**Operaciones**
Primera	NOT (lógico)
	− (aritmético, unario)
Segunda	∗, /, div, mod (aritmético)
	AND (lógico)
Tercera	+, − (aritmético)
	OR (lógico)
Cuarta	<, <=, >, >=, =, <>(relacional)