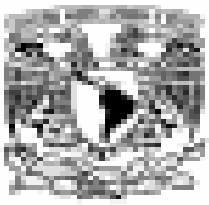


UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN

DIVISIÓN DEL SISTEMA UNIVERSIDAD ABIERTA

TUTORIAL

PARA LA ASIGNATURA

INTRODUCCIÓN A LA PROGRAMACIÓN

Fondo Editorial FCA

2003

DIRECTOR

C. P. C. y Mtro. Arturo Díaz Alonso

SECRETARIO GENERAL

L. A. E. Félix Patiño Gómez

JEFE DE LA DIVISIÓN-SUA

L. A. y Mtra. Gabriela Montero Montiel

COORDINACIÓN DE OPERACIÓN ACADÉMICA

L. A. Ramón Arcos González

COORDINACIÓN DE PROYECTOS EDUCATIVOS

L. E. Arturo Morales Castro

COORDINACIÓN DE MATERIAL DIDÁCTICO

L. A. Francisco Hernández Mendoza

COORDINACIÓN DE ADMINISTRACIÓN ESCOLAR

L. C. Virginia Hidalgo Vaca

COORDINACIÓN ADMINISTRATIVA

L. C. Danelia C. Usó Nava

TUTORIAL

PARA LA ASIGNATURA

INTRODUCCIÓN A LA PROGRAMACIÓN

2003

Fondo Editorial
F C A

Colaboradores

Diseño y coordinación general
L. A. Francisco Hernández Mendoza

Coordinación operativa
L. A. Francisco Hernández Mendoza

Asesoría pedagógica

Corrección de estilo
José Alfredo Escobar Mellado

Edición
L. C. Aline Gómez Angel

PRÓLOGO

En una labor editorial más de la Facultad de Contaduría y Administración, los Tutoriales del Sistema Universidad Abierta, representan un esfuerzo dirigido principalmente a ayudar a los estudiantes de este Sistema a que avancen en el logro de sus objetivos de aprendizaje.

Al poner estos Tutoriales a disposición tanto de alumnos como de asesores, esperamos que les sirvan como punto de referencia; a los asesores para que dispongan de materiales que les permitan orientar de mejor manera, y con mayor sencillez, a sus estudiantes y a éstos para que dispongan de elementos que les permitan organizar su programa de trabajo, para que le facilite comprender cuáles son los objetivos que se persiguen en cada materia y para que se sirvan de los apoyos educativos que contienen.

Por lo anterior y después de haberlos utilizado en un periodo experimental para probar su utilidad y para evaluarlos en un ambiente real, los ponemos ahora a disposición de nuestra comunidad, esperando que cumplan con sus propósitos.

ATENTAMENTE
Cd. Universitaria D.F., mayo de 2003.

C. P. C. Y MAESTRO ARTURO DÍAZ ALONSO,
DIRECTOR.

Primera edición mayo de 2003
DR © 2001 Universidad Nacional Autónoma de México
Facultad de Contaduría y Administración
Fondo editorial FCA
Circuito Exterior de Cd. Universitaria, México D.F., 04510
Delegación Coyoacán

Impreso y hecho en México
ISBN

Contenido

Introducción

El principal propósito de este tutorial es orientar a los estudiantes que cursan sus estudios en el sistema abierto, que se caracteriza, entre otras cosas, porque ellos son los principales responsables de su propio aprendizaje.

Como en este sistema cada alumno debe estudiar por su cuenta, en los tiempos y lugares que más le convengan, se vuelve necesaria un material que le ayude a lograr los objetivos de aprendizaje y que le facilite el acceso a los materiales didácticos (libros, publicaciones, audiovisuales, etcétera) que requiere. Por estas razones, se han estructurado estos tutoriales básicamente en cuatro grandes partes:

1. Información general de la asignatura
2. Panorama de la asignatura
3. Desarrollo de cada una de las unidades
4. Bibliografía

A su vez, estas cuatro partes contienen las siguientes secciones:

La información general de la asignatura que incluye: portada, características oficiales de la materia, índice de contenido del tutorial y los nombres de las personas que han participado en la elaboración del material.

El panorama de la asignatura contiene el objetivo general del curso, el temario oficial (que incluye solamente el título de cada unidad), y el temario detallado de todas las unidades

Por su parte, el desarrollo de cada unidad que está estructurado en los siguientes apartados:

1. Temario detallado de la unidad que es, simplemente, la parte del temario detallado global que corresponde a cada unidad.
2. Desarrollo de cada uno de los puntos de cada unidad.
3. Bibliografía general sugerida. Como no se pretende imponer ninguna bibliografía a los profesores, es importante observar que se trata de una **sugerencia**, **ya que cada profesor**

está en entera libertad de sugerir a sus alumnos la bibliografía que le parezca más conveniente.

Esperamos que este tutorial cumpla con su cometido y, en todo caso, deseamos invitar a los lectores, tanto profesores como alumnos, a que nos hagan llegar todo comentario o sugerencia que permita mejorarla.

A t e n t a m e n t e

L. A. y Mtra. Gabriela Montero Montiel

Jefe de la División del Sistema Universidad Abierta

Mayo de 2003.

Características de la asignatura

Introducción a la programación	Clave: 1137
Plan: 98	Créditos: 8
Licenciatura: Informática	Semestre: 1º
Área: Informática	Horas de asesoría: 2
Requisitos: Ninguno	Horas por semana: 4
Tipo de asignatura: Obligatoria (x)	Optativa ( )

Objetivo general de la asignatura

El alumno implantará algoritmos y estructuras de datos fundamentales en un lenguaje de programación imperativa.

Temario oficial (68 horas sugeridas)

1. Plataforma teórico conceptual (6 horas)
2. Datos constantes, variables, tipos, expresiones y asignaciones (6 horas)
3. Control de flujo en el lenguaje de programación Pascal (12 horas)
4. Procedimientos y funciones (14 horas)
5. Definición de tipos (8 horas)
6. Manejo dinámico de memoria e introducción a la implantación de estructuras de datos (10 horas)
7. Archivos (6 horas)
8. Compilación separada y diseño de programas (6 horas)

Temario detallado

1. Plataforma teórico-conceptual
   - 1.1 Antecedentes históricos del lenguaje de programación Pascal
   - 1.2 Conceptos de programación estructurada
   - 1.3 Conceptos de compilador, compilación y ejecución
   - 1.4 Estructura de un programa en el lenguaje de programación Pascal

- 5.3 Arreglos
  - 5.3.1 Unidimensionales
  - 5.3.2 Multidimensionales
- 5.4 Registros.

6. Manejo dinámico de memoria e introducción a la implantación de estructuras de datos
   - 6.1 Apuntadores
   - 6.2 Listas ligadas
   - 6.3 Pilas
   - 6.4 Colas
   - 6.5 Árboles binarios
7. Archivos
   - 7.1 Secuenciales
   - 7.2 De texto
   - 7.3 De entrada-salida
8. Compilación separada y diseño de programas
   - 8.1 Diseño descendente y ascendente de programas
   - 8.2 Encapsulamiento de módulos
   - 8.3 Uso de módulos en otros programas
   - 8.4 Las secciones de interfase e implementación
   - 8.5 Uso de los comandos make y build de Turbo Pascal

Unidad 1. Plataforma teórico-conceptual

Temario detallado

1. Plataforma teórico-conceptual
   - 1.1 Antecedentes históricos del lenguaje de programación Pascal
   - 1.2 Conceptos de programación estructurada
   - 1.3 Conceptos de compilador, compilación y ejecución
   - 1.4 Estructura de un programa en el lenguaje de programación Pascal
   - 1.5 Definición formal de un lenguaje de programación mediante diagramas. sintácticos de tren y ecuaciones de Backus-Naur

1.1. Antecedentes históricos del lenguaje de programación Pascal

Pascal es un lenguaje de programación de computadoras de alto nivel con propósito general. Fue desarrollado entre 1967 y 1971 por el Profesor Niklaus Wirth, en la Universidad Técnica de Zurich, Suiza. El nombre del programa fue elegido en honor de Blaise Pascal (1623-1662), brillante científico y matemático francés, entre cuyos logros se encuentra la primera máquina calculadora mecánica en el mundo.

El propósito de Wirth fue producir un lenguaje apropiado para enseñar, clara y sistemáticamente, las nuevas técnicas de desarrollo de la programación estructurada. La motivación inicial fue exclusivamente académica.

La amplia detección automática de errores ofrecida por los compiladores de Pascal es de gran ayuda para los principiantes, cosa que los hace bastante adecuados como introducción en la programación de computadoras. Sus características permiten escribir programas avanzados para diferentes áreas de aplicación.

Pascal ha sido bautizado como "el FORTRAN de los 80" y ha tenido gran influencia en lenguajes como ADA. Además, es descendiente de ALGOL 60, con quien comparte muchas características: estructuración en bloques (con

declaraciones de procedimientos anidados), fuerte tipificación de datos, soporte directo de recursividad… Por otra parte, incluye otras estructuras de datos (como registros y conjuntos) y permite la definición de tipos y el uso de punteros por parte del programador. A pesar de lo anterior, el profesor Wirth intentó que Pascal fuera lo más simple posible, a diferencia de ALGOL 68 (otro sucesor de ALGOL 60). Aunque hay que señalar que esta simplicidad no implica carencia de estructuras y limitaciones en el lenguaje, sino transparencia, claridad e integridad. Sin embargo, no debemos excluir algunas características costosas de esta programación, como las matrices de tamaño dinámico.

El antecesor directo de Pascal es ALGOL W, de quien toma los enunciados *while* y *case*, así como la definición de registros. Una innovación importante de Pascal es la definición de tipos de datos por el usuario, que no existía en lenguajes anteriores derivados de ALGOL. Por otra parte, Pascal es guía importante para la formalización de lenguajes de programación (semántica axiomática, utilizada para describir formalmente la de Pascal) y para la verificación formal de programas.

La primera versión de Pascal, de 1973, con errores de diseño detectados gracias a los trabajos de Hoare en verificación formal de programas, fue corregida y formalizada. En 1974, se presentó una segunda versión, el Pascal actual, por Niklaus Wirth y Kathleen Jensen: Pascal User Manual and Report.

Una versión muy conocida de Pascal fue desarrollada en la Universidad de California en San Diego: UCSD Pascal. Entre los lenguajes derivados de Pascal, están Pascal Concurrente, MODULA-2 y Euclid.

El éxito de Pascal se debió, además, a que sus versiones de lenguaje trabajan en un rango muy amplio de computadoras. Razón por la cual, la aceptación de Pascal aumentó considerablemente en 1984, cuando la compañía Borland International introdujo Turbo Pascal (compilador de alta velocidad y bajo costo para sistemas MS-DOS, del que se vendieron más de un millón de copias en diferentes versiones).

1.2. Conceptos de programación estructurada

Edgser W. Dijsktra, de la Universidad de Hainover, es considerado como el padre de la programación estructurada. En 1965, propuso sus aportes en un volumen titulado *Notas de programación estructurada*. Otras ideas recogidas en las obras *Notas sobre datos estructurados*, de Hoare, y *Programación estructurada*, de Dijsktra y Dahl (1972, Academic Press) influyeron en el desarrollo de Pascal.

La idea básica de la programación estructurada consiste en que la estructura del texto del programa debe auxiliarnos para entender la función del programa. Precisamente Pascal fue el primer lenguaje que utilizó plenamente los principios de la programación estructurada. Pero hasta mediados de los setenta comenzó a popularizarse esta filosofía.

La técnica de Pascal aumenta considerablemente la productividad del programa reduciendo en mucho el tiempo requerido para escribir, verificar, depurar y mantener los programas. Sin embargo, aún sigue mal interpretándose el verdadero objetivo de la programación estructurada. Algunos la reducen simplemente a la eliminación del salto incondicional de los programas como estructura de control.

La Programación Estructurada está enfocada a las estructuras de control de un programa. Su técnica primaria consiste en la eliminación del salto incondicional y su reemplazo por sentencias bien estructuradas de bifurcación y control.

La programación estructurada es un caso especial de la programación modular. El diseño de un programa estructurado se realiza construyendo bloques tan pequeños que puedan ser codificados fácilmente, lo que se logra hasta que se alcanza el nivel de *módulos atómicos*, es decir, sentencias individuales (si- entonces, haz-mientras, etcétera).

La definición de la programación estructurada exige sistemas de control bien organizados, definidos por los siguientes principios:

Teorema de la estructura: establece que se requiere tres bloques básicos para construir cualquier programa:

- Caja de proceso

- Decisión binaria
- Mecanismo de repetición

Programa propio: debe cubrir los siguientes requisitos:

- Tener un sólo punto de entrada hasta arriba.
- Leerse de arriba hacia abajo.
- Poseer un solo punto de salida hasta abajo.

Con el teorema de la estructura y el concepto de programa propio se construyen todas las estructuras sintácticamente válidas para esta filosofía de programación. Por una parte, el salto incondicional debe ser eliminado como una forma de programación; por otra, cualquier estructura que deseemos elaborar debe estar construida sólo por los tres elementos básicos mencionados en el teorema de la estructura, que debe cumplir con el concepto de programa propio.

Las estructuras que cumplen con las condiciones anteriores son:

- Bloque de proceso
- Decisión binaria
- Estructuras de repetición

  HAZ - MIENTRAS

  REPITE - HASTA
- Selección múltiple

1.3. Conceptos de compilador, compilación y ejecución

Compilador

Pascal es un lenguaje de alto nivel, ya que sus instrucciones se asemejan al lenguaje del ser humano. Antes de que una computadora ejecute un programa escrito en Pascal, primero debe traducirlo al lenguaje máquina por medio de un compilador. En esta fase, las instrucciones del programa se convierten en códigos numéricos binarios para operaciones y direcciones numéricas de datos y se almacenan en la

memoria RAM (memoria de acceso aleatorio) de la computadora para ejecutarse. El compilador asigna una dirección de memoria a cada variable utilizada en el programa.

Compilación

La compilación de un programa es el paso mediante el cual traducimos dicho programa al lenguaje máquina entendible por la computadora (código binario), para después ejecutarlo en la memoria.

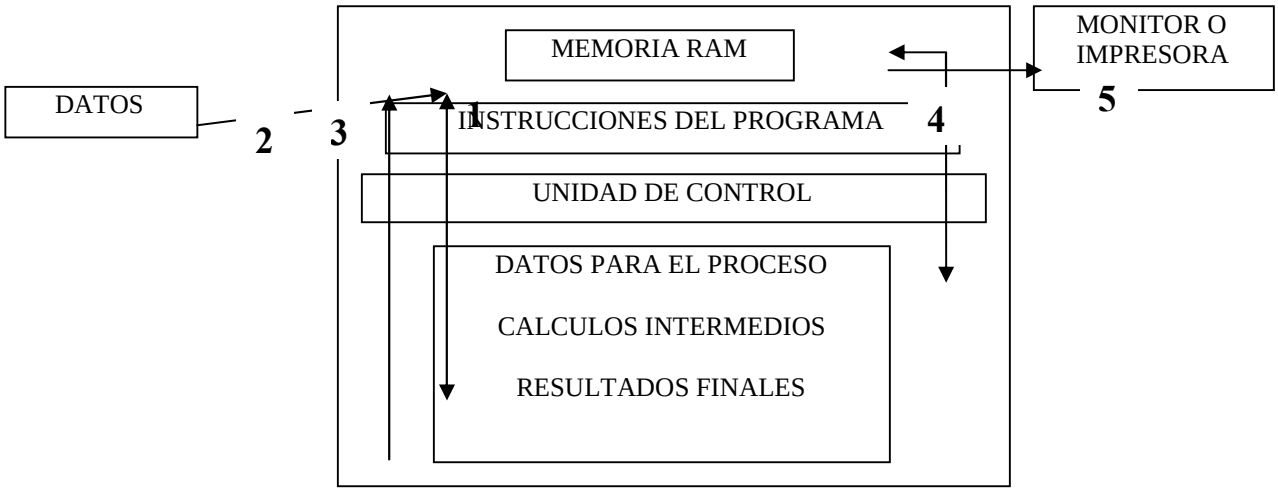
El compilador o traductor analiza todo el programa fuente (conjunto de instrucciones escritas en texto simple) y detecta limitaciones de sintaxis ocasionadas por fallas en la codificación o la transcripción (los errores de lógica que pueda tener nuestro programa fuente no son detectadas por el compilador). Cuando no hay fallas graves en la compilación, el compilador traduce cada orden del programa fuente a instrucciones propias de la máquina, creando el programa objeto.

Algunas computadoras utilizan interpretadores (ordinariamente, para el lenguaje Basic) para reemplazar los compiladores. La limitación del interpretador es que recibe, desde una terminal, sólo una instrucción a la vez, la analiza y, si está bien, la convierte al formato propio de la máquina. Si la instrucción tiene algún error, el interpretador lo indica al usuario para que haga la corrección.

Como resultado de la corrida del compilador, podemos obtener, entre otros, listados del programa fuente, de los errores detectados y de campos utilizados. Debemos corregir las fallas sobre el mismo programa fuente, ya sea en el disco flexible o en el duro. Este paso de la compilación lo debemos repetir hasta eliminar todos los errores y obtener el programa ejecutable.

Ejecución. Para ejecutar un programa, la unidad de control de la computadora (encargada de llevar a cabo las instrucciones del sistema operativo) examina cada una de las instrucciones del programa almacenadas en la memoria y envía señales de mando para realizarlas. Los datos entran a la memoria y se realizan cálculos sobre éstos (los resultados intermedios pueden guardarse en la memoria; los finales,

de acuerdo con las instrucciones dadas, pueden permanecer en la memoria o darles salida a algún dispositivo periférico como el monitor o a una impresora).



1.4. Estructura de un programa en el lenguaje de programación Pascal

Un programa en Pascal se divide en tres partes fundamentales: cabecera (nombre y lista de parámetros del programa), zona de declaraciones y definiciones (etiquetas, constantes, tipos, variables, procedimientos y funciones [que pueden anidarse]) y cuerpo (equivalente a la función *main* de C).

La forma general de un programa en Pascal es la siguiente:

PROGRAM nombre del programa (INPUT, OUTPUT); (*encabezado*)
CONST
    Definición de constante
    Definición de constante } Parte declarativa
VAR
    Definición de variable
    Definición de variable
BEGIN
    Proposición del programa
    Proposición del programa } Cuerpo del programa
END.

Cada identificador usado en el programa debe ser registrado en la parte de declaración. No puede utilizarse el mismo identificador para una constante y una variable. Todas las definiciones de constantes y variables se colocan después de las palabras reservadas CONST y VAR, respectivamente. Cada proposición, ya sea en la parte de declaración o en el cuerpo, debe separarse de la siguiente mediante punto y coma, exceptuando la que antecede a la palabra reservada END (que indica el final del programa).

A cada variable debe asignársele un valor inicial en el cuerpo del programa antes de ser manejada en una expresión o de que vaya a ser impresa.

Ejemplo de un programa de Pascal:

```
PROGRAM INVERSO (INPUT, OUTPUT);
CONST
        MARCO=”|”;
VAR
        A, B, C: CHAR;
BEGIN
        READ(A,B,C);
        WRITELN (MARCO,C,B,A,MARCO)
END.
```

Opcionalmente, pueden usarse comentarios en el programa (no forman parte de él) y colocarse en cualquier parte del programa. Deben ir entre paréntesis y asterisco (*comentario*) o entre llaves.

- Sintaxis: reglas que deben seguirse en la escritura del programa.
- Estatutos: comandos de Pascal diseñados para un propósito específico.
- Instrucción: indicación a la computadora sobre lo que ha de realizar. Las instrucciones se forman con los correspondientes estatutos de Pascal, siguiendo las reglas de sintaxis que dicho lenguaje determine.

- Programa: conjunto de instrucciones que indican a la computadora qué llevar a cabo. Es necesario que se especifiquen de acuerdo con la sintaxis de Pascal y en el orden lógico apropiado.
- Palabras reservadas: términos que sólo pueden ser usados para un propósito específico, ya que tienen un significado estándar predefinido (en la estructura básica, las negritas indican palabras reservadas en Pascal).
- Identificadores creados por el programador: palabras creadas por el programador (identificador para dar nombre al programa, nombres de variables, por ejemplo).

Algunas consideraciones

La computadora es una herramienta poderosa. Los datos de entrada pueden almacenarse en la memoria y manejarse a velocidades excepcionalmente altas para producir resultados (salida del programa).

Podemos describirle a la computadora una tarea de manejo de datos presentándole una lista de instrucciones (programa) que debe llevar a cabo.

Programación es elaborar una lista de instrucciones, es decir, escribir un programa. Diseñar un programa de computadora es muy similar a describir a alguien las reglas de un juego que nunca ha jugado. En ambos casos se requiere de un lenguaje de descripción que entiendan las partes involucradas en el fenómeno comunicativo.

Los lenguajes utilizados para la comunicación entre el hombre y la computadora son llamados de programación. Las instrucciones dadas a la computadora deben ser representadas y combinadas de acuerdo con las reglas de sintaxis del lenguaje de programación. Las reglas de un lenguaje de programación son muy precisas y no permiten excepciones o ambigüedades.

El éxito del lenguaje de programación Pascal se debe a su acertada selección del programa de control de estructuras que provee soporte efectivo para

programación estructurada, y a su variedad de datos que permite que éstos sean descritos detalladamente. Asimismo, este lenguaje ha sido diseñado para implementarse fácilmente, y su disponibilidad de modelos hace que se expanda rápidamente.

La combinación de registros variantes, arreglos y punteros permite que sean definidas, creadas y manipuladas gran variedad de estructuras de datos.

Pascal es estáticamente tipeado y la verificación de tipos en tiempo de compilación permite eliminar muchos errores de programación (indexar un arreglo con valor fuera del rango o asignar un número a una variable declarada como de cadena, por ejemplo).

En este lenguaje podemos definir un subrango de tipo primitivo discreto, elegir cualquier tipo primitivo discreto para el índice de un arreglo, escoger libremente el tipo de los componentes de un arreglo o emplear conjuntos como elementos de un tipo primitivo discreto.

El comando For puede ser usado para iterar sobre los índices de cualquier arreglo o rango de elementos potenciales de cualquier conjunto.

Pascal es una estructura de datos medianamente complicada, pero las definiciones de tipos la hacen clara. Muchos tipos punteros son usados en este lenguaje, y sería fácil confundirlos en operaciones con punteros. No obstante, cualquier error de asignaciones de punteros en Pascal, sería prevenido por una verificación de tipos en tiempo de ejecución.

De acuerdo con el párrafo anterior, programar punteros en Pascal es más seguro que en cualquier otro lenguaje con punteros sin tipos (como PL/I). Los punteros son usados en la implementación de objetos y tipos abstractos donde la explícita manipulación de los primeros es localizada y escondida; además, tienen dos tipos de roles distintos: aplicar tipos recursivos y compartir tipos de datos. Por eso, ahorran almacenamiento, pero lo más importante, reducen el riesgo de la inconsistencia de datos.

Debemos tener en cuenta que los tipos primitivos no fueron bien entendidos cuando se diseñó el lenguaje Pascal. Además, el criterio de Wirth consistió en hacer

un modelo simple de computación que permita a los programadores ver si correrán eficientemente sus programas. Así las cosas, no proveer tipos recursivos directos en Pascal fue razonable.

El repertorio de expresiones de Pascal es muy limitado, no considera las condicionales de ninguna clase. Además, carece de expresiones no triviales de tipos compuestos, debido al seguro de los agregados y porque las funciones no pueden devolver resultados compuestos. Estas omisiones fuerzan al programador a usar comandos en situaciones donde las expresiones podrían haber sido suficientes. Por ejemplo, la única forma de construir un valor de registro o arreglo es asignando uno a uno los componentes de una variable, arreglo o registro. En consecuencia, Pascal es un lenguaje de programación muy imperativo (soporta sólo abstracciones de procedimientos y funciones, que en Pascal no son valores de primera clase, pero pueden ser asignados o usados como componentes de tipos compuestos).

Pascal tiene gran repertorio de comandos. Los comandos *si* y *case* se emplean para ejecuciones condicionales; *repeat* y *while,* para iteraciones indefinidas; y For para finitas. Estos comandos pueden ser compuestos libremente.

En Pascal podemos declarar constantes, tipos, variables y abstracciones de funciones y procedimientos. Las declaraciones sólo pueden ser ubicadas en la cabecera de los bloques que van dentro de una función, procedimiento o del programa principal (por lo que los programadores están obligados a poner todas sus declaraciones en lugares comparativamente pequeños). Además, las declaraciones de Pascal deben ser agrupadas por clases, lo que dificulta mantenerlas relacionadas.

Hay cuatro mecanismos de variables: parámetros por valor, variable, procedural y funcional. Éstos permiten que los valores, referencias a variables, abstracciones de procedimientos y funciones puedan ser pasados como argumentos.

Los procedimientos y funciones solamente son clases de módulos que soporta Pascal. El concepto de encapsulamiento no está definido o soportado directamente en Pascal, pero puede ser usado en algunos de sus programas muy particulares.

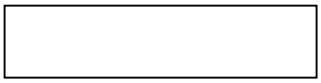
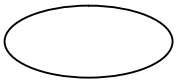
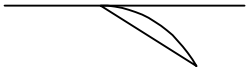
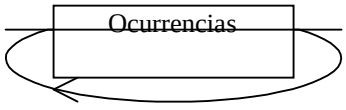
1.5. Definición formal de un lenguaje de programación mediante diagramas sintácticos de tren y ecuaciones de Backus-Naur

Los diagramas sintácticos de tren o de vías son una técnica para especificar la sintaxis de cada proposición. Ayudan al usuario a tener una comprensión precisa de las reglas del lenguaje Pascal en el momento de escribir o depurar programas.

Los círculos en los diagramas son palabras reservadas y símbolos especiales del lenguaje Pascal. Además, los elementos encerrados en los rectángulos son otras formas sintácticas (a cada una de ellas corresponde un diagrama separado). Cada diagrama debe ser rastreado en la dirección indicada por la flecha (de izquierda a derecha).

Pueden rastrearse muchas rutas a través del diagrama sintáctico para una proposición compuesta –según el ciclo–. El diagrama indica que una proposición compuesta está siempre delimitada por las palabras BEGIN y END. Además, debe utilizarse punto y coma para separar cada proposición de la siguiente.

Los símbolos utilizados en los diagramas sintácticos tienen su equivalente en las ecuaciones Backus-Naur, como lo muestra el siguiente cuadro.

Símbolo	Uso	Ecuaciones de Backus-naur
	Subproceso. Referencia a otro bloque mas detallado.	Se indica con itálicas o entre los signos “<” y “>”.
	Se escriben las palabras reservadas.	Se indican con negritas.
	Secuencia del texto.	
	Bifurcaciones. Son caminos alternos que indican rutas o textos opcionales.	Se encierran entre corchetes “[“ y “]”.
	Bucles. Son ciclos que indican repeticiones u ocurrencias múltiples de un elemento	Se indica por medio de llaves “{“ y “}”.

Ejemplo de utilización

Definición de un programa:
<programa>::=PROGRAM <identificador><parámetros del programa>;
<parte de declaración><cuerpo>

Direcciones electrónicas

http://elvex.ugr.es/etexts/spanish/pl/pascal.htm
http://www.itlp.edu.mx/publica/tutoriales/pascal/u1_1_4.html
http://aketzali.super.unam.mx/~edma/programacion/estructurada.html#estructurada
http://www.itlp.edu.mx/publica/tutoriales/pascal/u1_1_4.html
http://tarwi.lamolina.edu.pe/~jsalinas/pascal-1.html
http://www.e-mas.co.cl/categorias/informatica/turbo.htm

Bibliografía de la Unidad

Unidad 2. Datos, constantes, variables, tipos, expresiones y asignaciones

Temario detallado

2. Datos, constantes, variables, tipos, expresiones y asignaciones
   - 2.1 Identificadores
   - 2.2 Constantes
   - 2.3 ¿Qué es un tipo?
   - 2.4 Tipos nativos del lenguaje de programación Pascal
   - 2.5 Operadores aritméticos
   - 2.6 Expresiones aritméticas
   - 2.7 Operadores lógicos
   - 2.8 Expresiones booleanas

Identificadores

Se utilizan para dar nombre (secuencia de longitud que oscila entre 8 y 64 caracteres, de acuerdo con el compilador usado) a los objetos de un programa (constantes, variables, tipos de datos, procedimientos, funciones, unidades, programas y campos de registros). Los nombres aplicados deben ser significativos, es decir, que nos informen algo sobre el objeto que designan (por ejemplo, nombre, apellido y dirección); evitaremos nominaciones como xyz, rewlja, etcétera.

Un identificador se caracteriza por observar las siguientes reglas:

Inicia con una letra (A-Z) mayúscula o minúscula; no tiene espacios ni símbolos (&, !, *…) ni es alguna palabra reservada de Pascal como: PROGRAM, BEGIN, END, CONST, VAR, TYPE, ARRAY, RECORD, SET, FILE, FUNCTION, PROCEDURE, LABEL, PACKED. AND, OR, NOT, DIV, MOD, IF, THEN, ELSE, CASE, OF, REPEAT, UNTIL, WHILE, DO, FOR, TO, DOWNTO, WITH, GOTO, NIL, IN, etcétera.

Letras, dígitos y caracteres subrayados (_) están permitidos después del primer carácter.

Para Pascal no hay diferencias entre mayúsculas y minúsculas. Así, a un identificador denominado "valor" se le puede referir como "VALOR" o "VaLoR".

Hay dos tipos de identificadores: predefinidos, de Pascal, y definidos por el programador.

Algunos de los identificadores predefinidos son: *integer, real, byte...*

Los identificadores definidos por el programador son los elementos del lenguaje: variables, procedimientos, funciones, etcétera.

Todo identificador en Pascal debe ser definido previamente a su utilización.

Constantes

Una constante es un valor que no puede cambiar durante la ejecución del programa. Por ejemplo, si un programa va a utilizar el valor PI para algunos cálculos, puede definirse un identificador PI con valor de 3.1415926 constante, de tal forma que PI no pueda variar de valor; así, ya no es necesario escribir todo el número cada vez que se necesite en el programa, sólo PI.

Las constantes son de todos tipos: numéricas –con cifras enteras o reales–, caracteres, cadenas de caracteres, etcétera.

Las constantes, que se identifican con un nombre y un valor asignados, deben presentarse en el programa con el siguiente formato:

Const
Identificador = valor

Ejemplos:

```
const
Pi = 3.141592        {valor real}
Caracter = 'FCA'     {carácter}
Cuenta = 510         {número entero}
```

Hay constantes literales, que utilizan valores sin nombre, tales como 3, ´Francisco´ o true, 0.20; nominales, identificadores que asignan un valor a un nombre (por ejemplo, IVA=0.15, pi=3.1416, día=´miércoles´); y constantes de expresión,

donde se asigna al nombre un valor, resultado de una expresión (varios operadores y operandos) situada a la derecha del signo ´=´ (salario_trimestral=(40.35*365)/4, asigna el resultado 3681.94 a la constante salario_trimestral al realizar primero el cálculo matemático).

¿Qué es un tipo?

Los tipos de datos que se manejan en el programa son de dos clases: variables o constantes. Lo primeros, pueden cambiar a lo largo de la ejecución de un programa; en cambio, los constantes, son valores fijos durante todo el proceso. Por ejemplo, tenemos un tipo variable cuando sumamos dos números que serán introducidos por el usuario del programa, que puede agregar dos valores cualesquiera.

Hay diferentes tipos de datos. Algunos de los que usa Pascal principalmente son:

- *Integer* Números enteros sin parte decimal de 16 bits, desde –32568 hasta +32567.
- *Char* Caracteres del código ASCII de 8 bits definidos del 0 al 127.
- Bolean Pueden contener los valores falso o verdadero.
- Real Números que pueden incluir una parte decimal, normalmente almacenados en 32 ó 64 bits.
- *String* En una secuencia de caracteres que se trata como un solo dato.

Cada nuevo tipo se declara en la sección TYPE del programa principal y en cada bloque de éste. El tipo se distingue con un nombre seguido del signo “=” y el tipo asignado, terminando con “;”. Este último puede ser uno de los básicos, subrango de enteros, tipo enumerado o estructurado como arreglo, registro, conjunto o archivo.

Tipos nativos del lenguaje de programación Pascal

TIPOS ENTEROS (*integer*). Se expresan en el rango de valores.

TIPOS REALES (real). Son aquellos que representan al conjunto de los números reales.

TIPOS CARÁCTER (*char*). Pueden contener un solo carácter. Los tipos carácter pueden manifestarse gracias al código ASCII (en realidad ASCII ampliado).

TIPO CADENA (*string*). Secuencia de cero o más caracteres correspondientes al código ASCII, escritos en una línea sobre el programa y encerrados entre apóstrofos.

Operadores aritméticos

Los operadores aritméticos utilizados por Pascal para realizar operaciones matemáticas son los siguientes:

+	Identidad y suma
-	Negación y resta
*	Multiplicación
/	División de números fraccionarios
**, ^	Exponenciación
div	División entera con truncamiento
mod	Módulo o residuo de una división

Los símbolos +, -, *, ^, y las palabras clave *div* y *mod* son conocidos como operadores aritméticos. En la expresión *8 + 6* los valores 8 y 6 se denominan operandos. El valor de la expresión 8+6 es llamado también resultado de la expresión. Los cálculos que implican tipos de datos reales y enteros suelen ser normalmente del mismo tipo de los operandos.

OPERADOR	SIGINIFICADO	TIPO DE OPERANDOS	TIPO DE RESULTADO
^,**	Exponenciación	Entero o real	Entero o real
+	Suma	Entero o real	Entero o real
-	Resta	Entero o real	Entero o real
*	Multiplicación	Entero o real	Entero o real

/	División	Real	Real
Div	División entera	Entero	Entero
Mod	Módulo (residuo)	Entero	Entero

Además de los operadores aritméticos básicos, tenemos los div y mod (se pueden usar únicamente con números enteros).

Operador DIV

Calcula el cociente entero de la división de dos números: el símbolo / se utiliza para la división real y el operador *div* representa la entera.

C div D

Sólo se puede utilizar si C Y D son expresiones enteras y se obtiene la parte entera de C/D. Así, el resultado de *27 div 9* es *3*.

Formato

Operando1 div operando2

Operador MOD

El operador mod calcula el residuo de la división.

Formato

Operando1 mod operando2

Ejemplo: 16 mod 3 = 1

Expresiones aritméticas

Las expresiones aritméticas son análogas a las fórmulas matemáticas. Las variables y constantes son numéricas (reales o enteras) y las operaciones, aritméticas.

El lenguaje Pascal permite representar fórmulas con expresiones que pueden usar operadores múltiples y paréntesis, con lo que se consigue buena capacidad matemática. Los paréntesis se emplean para agrupar términos de la misma manera como se utilizan en álgebra.

Las expresiones que tienen dos o más operandos requieren reglas matemáticas (de prioridad o precedencia) que permitan determinar el orden de las operaciones. Las principales reglas son las siguientes:

Las operaciones que están estructuradas entre paréntesis se evalúan primero. Si existen diferentes paréntesis anidados (interiores unos a otros) las expresiones más internas se evalúan primero.

Las operaciones aritméticas dentro de una expresión suelen seguir el siguiente orden de importancia:

- Operador exponencial (^, **)
- Operadores *, /
- Operadores +, -
- Operadores div y mod

En caso de coincidir varios operadores de igual prioridad en una expresión o subexpresión encerrada entre paréntesis, el orden de relevancia va de izquierda a derecha.

Ejemplo:

```
PROGRAM FÓRMULA (INPUT,OUTPUT)
VAR
A,B,C,D:REAL
BEGIN
    (*ASIGNACIÓN DE VALORES A LAS VARIABLES*)
    A:=5
    B:=15
    C:=2
    D:=10
    (*IMPRESIÓN DEL RESULTADO DE LA FÓRMULA*)
    WRITELN(A+B*((D-A)/C))
END
```

La evaluación de la expresión se lleva a cabo en el orden siguiente: +,-,* y /, tomando en cuenta la agrupación de los operandos. Así, en el programa anterior obtenemos un resultado dando los siguientes pasos:

Paso 1: sumar la variable A más la variable B: 5 + 15 = 20

Paso 2: restar a la variable D la variable A: 10 – 5 = 5

Paso 3: dividir el resultado anterior entre la variable C: 5 / 2 = 2.5

Paso 4: multiplicar el resultado del paso 1 por el resultado del paso anterior: 20*2.5=50

Resultado: 50

Operadores lógicos

Los operadores lógicos o booleanos básicos son NOT (negación no), AND (conjunción y) y OR (disyunción o).

OPERADORES LÓGICOS		
Operador lógico	Expresión lógica	Significado
NO (NOT)	No p (not p)	Negación de p
Y (AND)	P y q (p and q)	Intersección de p y q
O (OR)	P o q (p or q)	Unión de p y q

Las definiciones de las operaciones NO, Y, O se resumen en las tablas denominadas de verdad.

TABLAS DE VERDAD

A	**NO A**
Verdadero	Falso
Falso	Verdadero

A	B	A y B
Verdadero	Verdadero	Verdadero
Falso	Verdadero	Falso
Verdadero	Falso	Falso
Falso	Falso	Falso

A	B	A o B
Verdadero	Verdadero	Verdadero
Falso	Verdadero	Verdadero
Verdadero	Falso	Verdadero
Falso	Falso	Falso

En las expresiones lógicas es posible mezclar operadores de relación y lógicos.

Operadores de relación

Los operadores relacionales o de relación permiten hacer comparaciones de valores de tipo numérico o carácter. Sirven para expresar las condiciones en los algoritmos.

OPERADORES DE RELACIÓN	
<	Menor que
>	Mayor que
=	Igual que
<=	Menor o igual a
>=	Mayor o igual a
<>	Distinto de

El formato general para las comparaciones:

Expresión 1 *operador de relación* Expresión 2

El resultado de la operación será verdadero o falso.

Los operadores de relación pueden aplicarse a cualquiera de los cuatro tipos de datos estándar: entero, real, lógico y carácter.

Prioridad de los operadores lógicos y relacionales

Los operadores lógico-relacionales también siguen un orden de prioridad cuando hay más de un operador en la expresión.

Expresiones booleanas

El valor de este tipo de expresiones lógicas es siempre verdadero o falso. Se denominan booleanas en honor al matemático británico George Boole, quien desarrolló el álgebra lógica o de Boole.

Las expresiones lógicas se forman combinando constantes o variables lógicas y otras expresiones de este rango, utilizando los operadores lógicos NOT, AND y OR y los relacionales (de relación o comparación) =, <, >, <=, >=, <>.

Ejemplos de expresiones lógicas:

NOT ((altura<1.20) OR (altura>1.80)): evalúa primero los paréntesis internos; la altura debe estar entre 1.20 y 1.80 para que el resultado sea verdadero.

(curso_inglés=true) AND (posgrado=maestría): si la variable posgrado toma el valor de maestría y la variable curso_inglés contiene el valor true, entonces toda la expresión es cierta.

(sabe_inglés=true) OR (sabe_programar=true): si sabe inglés o sabe programar, la expresión es cierta.

Direcciones electrónicas

http://aketzali.super.unam.mx/~edma/programacion/operadores.html

http://www-gris.ait.uvigo.es/~belen/isi/doc/tutorial/unidad1p.html

http://tarwi.lamolina.edu.pe/~jsalinas/pascal-1.html

Bibliografía de la Unidad

Unidad 3. Control de flujo en el lenguaje de programación pascal

Temario detallado

3. Control de flujo en el lenguaje de programación Pascal
   - 3.1 Condicionales
   - 3.2 Iteraciones
   - 3.3 Salto incondicional

3.1 Condicionales

Un algoritmo es ejecutado siguiendo la lista de los pasos, determinados por los valores de ingreso, que lo describen. Estos valores están contenidos en variables, con base en las cuales se evalúa una condición: el resultado especifica los puntos a seguir**.**

Las condiciones (condicionales) de las decisiones se valoran mediante una expresión de Boole, cuyo resultado puede ser verdadero o falso. Asimismo, la decisión se representa mediante una expresión booleana (describe una relación entre dos variables por medio de los operadores de relación mayor que >, menor que <, igual que =, diferente de <>, mayor o igual que >= o menor igual que <=).

Para ilustrar lo anterior, tenemos el siguiente algoritmo:

Requisitos para licencia de conducir:

1. Lee nombre del aspirante
2. Lee edad
3. Lee sabe_conducir
4. Lee ve_bien
5. Si edad<=18, ir al paso 9
6. Si sabe_conducir=”N”, ir al paso 9
7. Si ve_bien=”N”, ir al paso 9
8. Otorgar licencia a nombre del aspirante e ir al paso 10

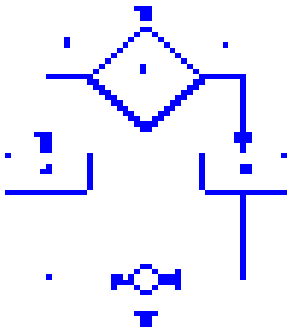
9. Negar licencia al aspirante.
10. Terminar

Como podemos observar, los números 5-7 contienen expresiones booleanas en las que, si la evaluación da resultado verdadero, el proceso va hasta el paso 9 (en el que se niega la licencia al aspirante) para, luego, terminar el algoritmo. En caso de resultar falso, se ejecuta el paso siguiente (número 8: se otorga la licencia sólo si el aspirante tiene 18 años o más, sabe conducir y posee buena vista), para luego, en el punto 10, concluir el algoritmo.

El término decisión, en programación, hace referencia a la estructura SI-CIERTO-FALSO por medio de la cual analizamos una condición o predicado para obtener un resultado, que plantea dos alternativas: verdadero o falso. El lenguaje Pascal expresa la decisión como IF...THEN...ELSE. Su sintaxis es:

IF: condición o predicado
THEN
Proposición verdadera
ELSE
Proposición falsa

También podemos decir: si (IF) la expresión (condición o predicado) se evalúa y resulta verdadera, entonces (THEN) ejecuta las instrucciones establecidas en verdadero; de lo contrario, si no (ELSE), lleva a cabo las órdenes marcadas en falso. Esto, gráficamente, se representa:



En el diagrama anterior, la estructura IF. THEN ELSE tiene una entrada; la condición C es evaluada para obtener un resultado, si éste es cierto se ejecuta el proceso A; de lo contrario, el B (cualquiera que sea la alternativa elegida, tiene la misma salida de la estructura). De hecho, los procesos A y B son en sí mismos estructuras que pueden contener otras más complejas.

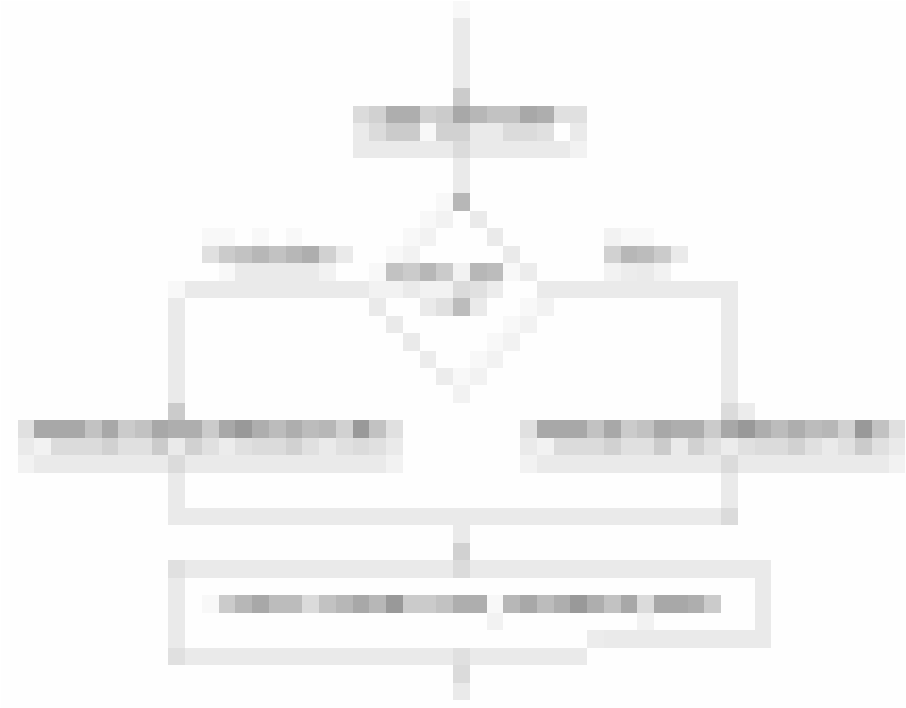
Ejemplificamos lo anterior en el siguiente algoritmo:

Algoritmo: descuentos

1. Declarar a precio como constante y asignarle el valor de 10.
2. Declarar a las variables cant_art y a precio_neto como reales.
3. Leer cant_art.
4. Si cant_art>=10, entonces, precio_neto=precio*0.8; si no, precio_neto=precio*0.90.
5. Imprimir “TOTAL A PAGAR = “,cant_art*precio_neto.
6. Terminar.

Explicación: si el cliente compra de 1 a 10 artículos, se le descuenta 10% (el precio neto se obtiene multiplicando el valor por 0.90). Ahora, si nuestro cliente adquiere más de 10 artículos, el descuento asciende a 20% (para obtener el precio neto, se multiplica el valor por 0.80). Al final, se imprime el total a pagar multiplicando la cantidad de artículos compradas por el precio, incluido el descuento.

En un diagrama de flujo, podemos expresar así lo anterior:



Podemos codificar el mismo ejemplo en lenguaje Pascal:

```
PROGRAM Descuento;

CONST
precio = 10;

VAR
    cant_art, precio_neto : Real;

BEGIN
    (*LECTURA DE DATOS*)
    READLN(cant_art);
    IF cant_art>=10 THEN precio_neto:= precio*.8 ELSE
    precio_neto:=precio*.9;
    WRITELN('TOTAL A PAGAR',cant_art*precio_neto)
END.
```

Explicación: en este programa llamado "descuento", se le asigna a la constante PRECIO el valor 10 (venta normal al público) y se declaran las variables CANT_ART y PRECIO_NETO de tipo real. En el cuerpo del programa, se le solicita al usuario el ingreso del valor de la variable CANT_ART (condición a evaluar en la expresión booleana CANT_ART>=10). Si es verdad que CANT_ART es mayor o igual a 10, se asignará a la variable PRECIO_NETO el 80% de la variable PRECIO; si no, el 90%. Aquí termina la estructura de decisión para, luego, poder imprimir la cadena "TOTAL A PAGAR: " y el producto CANT_ART por el PRECIO_NETO, como resultado del precio neto total a pagar por el cliente. (En esta estructura es posible omitir ELSE con sus instrucciones, pues son opcionales).

Además, a la estructura IF THEN ELSE se le pueden anidar otras del mismo tipo (sentencias IF anidadas). Así, cuando la condición es verdadera, pueden ser

ejecutadas una o más estructuras IF THEN ELSE dentro de la primera. Sucede lo mismo con la alternativa falsa, a la que también se le puede anidar este tipo de estructuras. En forma lógica, se evalúa una condición dentro de otra y así, sucesivamente, las veces necesarias. Por ejemplo:

```
IF qualification>5 THEN
BEGIN
IF calificacion=6 THEN final:=´Suficiente´;
IF calificacion=8 THEN final:=´Bien´;
IF calificacion=10 THEN final:=´Excelente´;
END;
ELSE final:=´No Acreditado´
```

3.2 Iteraciones

Las iteraciones constituyen la ejecución repetida de un conjunto de instrucciones mientras se cumple determinada condición. Las estructuras que cubren este concepto son: FOR, WHILE y REPEAT-UNTIL, que estudiamos a continuación.

- ✓ **FOR**

Cumple una serie de órdenes las veces que la variable de control del ciclo cambie de valor, desde el inicial hasta el final. Para comprender mejor está definición, mencionemos la sintaxis de este ciclo:

```
FOR identificador: = valor inicial TO valor final DO
Instrucciones.
```

Donde la variable identificador cambiará de valores desde el inicio hasta el final. Las instrucciones son las sentencias que serán ejecutadas por cada valor nuevo que tome el identificador.

Consideremos las siguientes peculiaridades de los ciclos FOR: en la variable identificador, los valores inicial y final pueden ser de cualquier tipo: CHAR, INTEGER o BOLEAN, y representar una variable, constante o expresión. La variable identificador no cambiará mientras se ejecuta el ciclo; una vez terminado, aquélla se considera como no definida. El valor final puede cambiar una vez que se cumpla el ciclo, independientemente del número de veces que se ejecutan las instrucciones. El valor final debe ser mayor que el inicial; si no, el ciclo no podrá ejecutarse, a menos que se utilice la forma:

```
FOR identificador := valor inicial DOWNTO valor final DO
Instrucciones.
```

En este caso, la variable identificador sigue un conteo decreciente.

El siguiente programa escribe los números del 1 al 100:

```
PROGRAM Conteo;
VAR
ciclo : Integer;

BEGIN
FOR ciclo := 1 to 100 DO
Writeln(ciclo);
END.
```

La desventaja de los ciclos FOR es que, una vez iniciados, se ejecutarán sólo el número de veces definido en identificador.

Para que un ciclo cuente en forma decreciente, debemos utilizar la palabra DOWNTO en lugar de TO, y el valor inicial debe ser mayor que el final. Tomemos el ejemplo anterior:

```
PROGRAM Conteo;
VAR
ciclo : Integer;
BEGIN
        FOR ciclo := 100 downto 1 DO
                Writeln(ciclo);
END.
```

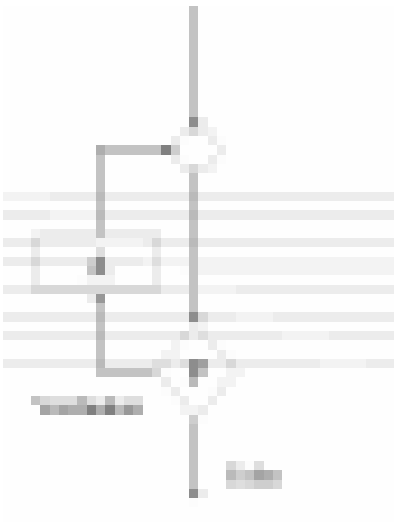
- ✓ **WHILE**

La sintaxis de WHILE es la siguiente:

```
WHILE condición DO
Instrucciones
```

En esta estructura, primero se evalúa la condición. Si el resultado es verdadero, se ejecutan las instrucciones y se evalúa nuevamente la condición; si resulta falsa, se sale del ciclo (en caso contrario, reinicia el procedimiento).

El diagrama de flujo para esta estructura es:



En este caso, el bloque A se ejecuta repetidamente mientras que la condición P es verdadera. También tiene entrada y salida únicas. Asimismo, el proceso A puede ser cualquier estructura básica o conjunto de estructuras.

Si la condición resulta falsa en la primera evaluación de la condición, es posible que las instrucciones no se lleguen a ejecutar. Una vez que se sale del ciclo, se llevarán a cabo las órdenes que siguen a la estructura WHILE.

Pongamos como ejemplo este programa, que imprime una lista de precios:

```
PROGRAM lista_precios (input, output);

CONST
punit=10;
VAR
cant,precio : integer;
BEGIN
cant:=0;
WHILE cant<=100 DO
BEGIN
        cant := cant+1;
        precio := cant*punit;
        WRITE(cant,precio);
END (*WHILE*)
END.
```

Explicación: se declara como constante el Punit y se designan como enteras las variables Cant y Precio; se asigna valor inicial de 0 a Cant; se prueba Cant antes de que el ciclo se ejecute (Cant<=100); se incrementa Cant en la unidad (Cant := Cant+1); se calcula el nuevo Precio y se escriben en pantalla las variables Cant y Precio. Se reinicia el ciclo probando la variable Cant hasta que alcance valor de 101, para que resulte falsa la condición Cant<=100 y se salga del ciclo (así, en total el ciclo puede repetirse 100 veces).

En ciertos problemas, es necesario volver a calcular el valor de la variable de control del ciclo, que deberá almacenar nuevos valores en la ejecución de las instrucciones de WHILE. Por ejemplo, un programa de inventarios. En éste, se pueden dar salidas por la venta de artículos, actualizando la variable de control de

existencias hasta que llegue a un tope mínimo y abandone el ciclo para solicitar el reabastecimiento por medio de la compra de más artículos.

Los ciclos WHILE brindan la ventaja de que la tarea se realiza mientras se cumpla una condición: es posible controlar el número de repeticiones una vez iniciados.

- ✓ **REPEAT-UNTIL**

En este caso, el ciclo se ejecuta mientras la evaluación de la condición resulte falsa; si es verdadera, se sale del ciclo y continuará la ejecución con la siguiente proposición del programa.

Diferencias con WHILE: REPEAT-UNTIL opera mientras la condición sea falsa, en WHILE es a la inversa; REPEAT ejecuta las instrucciones de su estructura al menos una vez, puede ser que WHILE no lo haga ni una sola, debido a que al principio realiza la prueba de evaluación de la condición (en REPEAT esta prueba se encuentra al final de las instrucciones); REPEAT no necesita encerrar sus instrucciones dentro de un BEGIN y END (como WHILE), ya que la prueba de condición delimita el final de la estructura.

Como ejemplo tenemos el siguiente programa, que calcula la suma de 10 números. Primero, mediante la proposición REPEAT-UNTIL y después con WHILE, para apreciar las diferencias entre ambas.

```
número := 0
suma_acum. := 0
REPEAT
WRITE (número,suma_acum);
número = número+1;
suma_acum. = suma_acum.+número
UNTIL número>10
```

```
número := 0
suma_acum. := 0
WHILE número<=10 DO
BEGIN
WRITE (número,suma_acum.);
número = número+1;
suma_acum. = suma_acum.+número
END(*Fin de WHILE*)
```

La expresión booleana de la condición REPEAT usa la comparación número>10; WHILE, número<=10. Ambas estructuras usan distintos operadores en la expresión: mayor que (>) en REPEAT contra el menor igual (<=) de WHILE.

3.3 Salto incondicional

- ✓ **Sentencia GOTO**

Es usada para realizar saltos incondicionales en la ejecución de un programa. Así, en lugar de seguir la aplicación en la línea siguiente, se realiza la transferencia a otra del cuerpo del programa, que puede encontrarse antes o después de donde se da la llamada GOTO. De este modo, cambia el flujo del programa.

Para poder realizar este tipo de salto, es necesario declarar etiquetas, las cuales deben asociarse con las proposiciones GOTO donde están contenidas. Dichas etiquetas pueden considerarse como identificadores que marcan el lugar a donde se dirigirá el flujo del programa al momento de usar el GOTO. Deben declararse en la sección propia de declaración, antes que la de constantes y variables.

La palabra reservada para declarar etiquetas es LABEL y tiene la forma:

```
LABEL
etiqueta_1, etiqueta_2, etiqueta_3.....etiqueta_n;
```

El nombre de la etiqueta es un identificador como cualquier otro, pero al utilizarse debe terminar con dos puntos ":". La sintaxis del comando es: GOTO etiqueta Por ejemplo:

```
PROGRAM Ordena;
LABEL
salto,salto2;

CONST
a = 20;
b = 15;
c = 5;
VAR
temp.: integer;

BEGIN
        IF a<=b THEN GOTO salto ELSE
        BEGIN (*INTERCAMBIO DE VALORES*)
                temp:=a;
                a:=b;
                b:=temp;
        END
        salto:
        IF a<=c THEN GOTO salto2 ELSE
        BEGIN (*INTERCAMBIO DE VALORES*)
                temp:=a;
                a:=c;
                c:=temp;
        END;
        IF B<=C THEN
        BEGIN (*INTERCAMBIO DE VALORES*)
                temp:=b;
                b:=c;
                c:=temp;
        END;
        WRITELN('La serie ordenada es: ',a,b,c)
END.
```

Explicación: se realiza una ordenación ascendente de tres números enteros positivos. La primera variable, ‘a’, se compara con ‘b’ y ‘c’; si es mayor, intercambia su valor con cualquiera de las dos, usando una variable temporal y dejando el valor menor en la primera variable. Posteriormente, la variable ‘b’ se compara con ‘c’ y se repite el procedimiento ya descrito. Una vez ordenada la lista, se imprime su salida en pantalla.

GOTO realiza dos saltos incondicionales en la corrida del programa: uno hacia la etiqueta ‘salto’ y otro hacia la etiqueta ‘salto2’.

GOTO no se debe utilizar en exceso, pues hace que los programas sean difíciles de seguir y depurar. En caso de que el programa tenga algún error o se tenga que actualizar, debe revisarse todo el cuerpo, que dará varios saltos a otras proposiciones del mismo programa (esto hace tediosa su revisión y/o modificación). Además que está propensa a errores, esta forma de programar se conoce como programación libre, totalmente ajena a la programación estructurada de Pascal.

Direcciones electrónicas

http://www.itlp.edu.mx/publica/tutoriales/pascal/u1_1_4.html
http://www-gris.ait.uvigo.es/~belen/isi/doc/tutorial/unidad3p.html
http://www-gris.ait.uvigo.es/~belen/isi/doc/tutorial/unidad3p.html

Bibliografía de la Unidad

Unidad 4. Procedimientos y funciones

Temario detallado

4. Procedimientos y funciones
   - 4.1 Concepto de procedimiento
   - 4.2 Concepto de función en programación
     - 4.2.1 Funciones internas
     - 4.2.2 Funciones propias
   - 4.3 Alcance de variables
     - 4.3.1 Locales
     - 4.3.2 Globales
   - 4.4 Pase de parámetros
     - 4.4.1 Por valor
     - 4.4.2 Por referencia
   - 4.5 Funciones recursivas

4.1 Concepto de procedimiento

Los programas pueden estar estructurados de manera jerárquica, es decir, que los niveles inferiores realicen las operaciones en forma detallada y los superiores trabajen sólo con la información más general. Esto se logra al incluir procedimientos en los programas. Un procedimiento es un conjunto de instrucciones que efectúan una labor específica. Si se requiere multiplicar una misma tarea en la ejecución de un programa, no será necesario volver a escribir el código completo, tan sólo se llama al procedimiento por medio del nombre que ya tiene asociado.

El procedimiento está escrito como una estructura similar a un programa de Pascal. Las diferencias son que el primero realiza una tarea específica cada vez que el programa principal lo llama por medio del nombre –proposición de Pascal– y que

está contenido en el mismo. Podemos decir que un procedimiento es como un subprograma.

Por ejemplo:

```
PROGRAM columnas (INPUT,OUTPUT);
VAR
        i:INTEGER;
PROCEDURE asteriscos;
BEGIN
        WRITELN(‘ ** ** ** ** ‘)
END;

BEGIN
FOR i:=1 TO 3 DO
        BEGIN
        asteriscos;
        WRITELN(‘ >> Hola <<’)
        END;
READLN
END.
```

El programa produce la siguiente salida en pantalla:

```
** ** ** **
>> Hola <<
** ** ** **
>> Hola <<
** ** ** **
>> Hola <<
```

En el programa ‘columnas’ se declara la variable **i** como entera y, para poder utilizarlo, después se crea el procedimiento “asteriscos” dentro de “columnas”. El procedimiento actuará como instrucción de Pascal (para ser ejecutado, debe teclearse su nombre –llamada al procedimiento–).

El procedimiento ‘asteriscos’ se declara con la palabra reservada Procedure y sólo imprime en pantalla una serie de asteriscos y luego inserta un renglón con la instrucción WRITELN. Después, inicia el cuerpo del programa encerrado dentro de BEGIN y END. Se utiliza la estructura FOR para hacer que el ciclo siguiente se multiplique tres veces: el programa llama al procedimiento ‘asteriscos’ y luego escribe la frase ‘ >> Hola <<’; posteriormente, la instrucción READLN hace que el programa se pause hasta que el usuario presione alguna tecla, con el fin de que pueda verse el resultado y luego terminar el programa.

Un procedimiento puede ser llamado las veces que el programador considere necesarias (en el ejemplo anterior, se ha invocado al procedimiento en tres ocasiones, ahorrando así la codificación de la tarea).

Como ya expresamos, un procedimiento tiene estructura parecida a la de un programa, es decir:

- ✓ El encabezado del procedimiento, formado por el nombre de éste y la palabra reservada Procedure.
- ✓ La sección de las declaraciones de constantes y variables.
- ✓ El cuerpo del procedimiento, conformado por la palabra reservada BEGIN, seguida por una serie de proposiciones y END –debe terminar con punto y coma; pero el END que termina al programa cierra con punto–.
- ✓ La declaración del procedimiento debe estar entre la sección de declaraciones de constantes y variables del programa principal y el cuerpo del mismo. Las organizaciones de la estructura del programa y del procedimiento son muy parecidas: ambas contienen declaración de variables y constantes, cuerpo y palabras reservadas BEGIN y END (esta estructura es conocida como de bloque).

A continuación mencionamos algunas de las ventajas al usar los procedimientos:

- ✓ Es posible utilizarlos varias veces, ahorrando bastante programación. Incluso existen bibliotecas de procedimientos listas para implementarse en los programas.
- ✓ Permiten la división del trabajo entre los programadores: cada uno puede realizar, en forma independiente, uno particular.
- ✓ Facilitan la programación modular con diseño descendente, ya que un procedimiento principal puede llamar a otros a la vez.
- ✓ Como son en sí mismos programas, para verificar que no tienen errores, pueden probarse de manera separada al programa principal. Lo anterior facilita en mucho la corrección y depuración de programas.

4.2 Concepto de función en programación

La función es una unidad independiente que tiene sus propias etiquetas, constantes, variables y un conjunto de sentencias que al ejecutarse ofrecen un valor. Comparadas con los procedimientos –que realizan determinadas tareas–, las funciones dan valor nuevo.

La declaración de una función se hace mediante la palabra reservada Function, que debe escribirse después de las declaraciones de variables del programa principal. Sintaxis: FUNCTION nombre (parámetros): tipo de datos;
La estructura de la función:

```
FUNCTION cubo (a : REAL) : REAL
BEGIN
    cubo := a*a*a
END;
```

La primera línea se conoce como encabezado, ahí, se encuentran el nombre de la función y, entre paréntesis, las variables que se usan como entrada a la función y su tipo de datos; posteriormente, se define el resultado como real. El cuerpo del programa de la función está entre las palabras reservadas BEGIN y END (aquí se debe asignar el resultado de una expresión al nombre dado a la función). En el ejemplo anterior, la función tiene el nombre cubo; en el cuerpo del programa, se le asigna a ese mismo nombre el producto de a*a*a (potencia cúbica de la variable ‘a’). Ésta es la única forma de lograr que la función pueda devolver algún valor.

A continuación, un ejemplo más:

```
PROGRAM Funciones
VAR
A, B, C : Real;
FUNCTION cociente (x, y : Real) : Real;
BEJÍN
cociente := (x / y);
END;
BEJÍN
a := 21.75;
b := 7;
c := cociente (a, b);
WRITELN('El cociente de ',a,' y ',b,' es: ',c);
END.
```

En el programa anterior, la función ‘cociente’ divide dos números de tipo real para obtener como resultado un cociente del mismo tipo. En dicho programa, se hace una llamada a la función con la expresión Cociente (a,b), y el resultado se asigna a la variable real ‘c’. (Nota cómo los parámetros de la función cociente ‘x’ y ‘y’ son sustituidos por las variables ‘a’ y ‘b’, declaradas en el programa principal, cuando se invoca a la función).

Asimismo, las funciones pueden contener programas en su estructura, aspecto que analizamos a continuación.

4.2.1 Funciones internas

Pascal incluye un conjunto de funciones internas que efectúan operaciones comunes y rutinarias; no es necesario que el programador las codifique, basta incorporarlas al programa. Para utilizarlas, sólo se teclea el nombre de la función seguido de un paréntesis que contiene los argumentos de ésta. Por ejemplo, para obtener la raíz cuadrada de 36, debemos escribir sqrt(36), que da como resultado 6.

Las funciones, como ya explicamos, tienen dos tipos de datos: uno para los argumentos de la función y otro para su resultado. No es necesario que ambos sean semejantes. Por ejemplo: TRUNC(8.95) da como resultado 8; esta función acepta un valor real y da como consecuencia un tipo entero. A las funciones que aceptan dos tipos de datos en su declaración se les llama de transferencia.

Además, la clase de datos que arrojan estas funciones debe guardar consistencia con los valores que después van a usar. El dicho de “no sumar naranjas con manzanas” es aplicable en este caso, ya que si el resultado de la función es real, sólo pueden hacerse operaciones con valores del mismo tipo. Por ejemplo: sqrt(36)+4.568 es expresión válida, ya que suma el resultado real de la raíz cuadrada a un número de tipo real como 4.568. Caso contrario, trunc(8.562)/1.54 es incorrecta, puesto que la función trunc obtiene un número entero que se divide entre uno real.

Estas funciones son estándar, aun sin declararlas, están presentes en cualquier versión de Pascal (ya están programadas por los fabricantes del Pascal y son parte de ese lenguaje). También muchas versiones del lenguaje incorporan funciones que no pueden encontrarse en versiones antiguas.

Enseguida puntualizamos las funciones internas de Pascal.

✓ **Truncamiento y redondeo**

El truncamiento permite eliminar la parte decimal de un número real, la función TRUNC devuelve un número de tipo entero. Sintaxis: Var_Entera := TRUNC(Num_Real) ;

El redondeo, por su parte, cierra el número entero más cercano, lo que se define con sus decimales: mayor o igual a 0.5 sube al número entero consecutivo, menor a 0.5 trunca la parte decimal, la función es ROUND( ), devuelve un numero de tipo entero. Sintaxis: Var_Entera := ROUND(Num_Real) ;

Por ejemplo:

```
PROGRAM cierra_dec ;
BEGIN
    WRITELN( ' Truncar 5.658 = ', TRUNC( 5.658 ) ) ;
    WRITELN( ' Redondear 8.5 = ', ROUND( 8.5 ) ) ;
    WRITELN( ' Redondear 9.4 = ', ROUND( 9.4 ) ) ;
END.
```

Lo anterior da como salida:

```
5
9
9
```

✓ **Funciones exponenciales y logarítmicas**

La función SQR( ), da el cuadrado de un número; su tipo es real o entero, según el caso. Sintaxis: Variable := SQR(número) ;

La función SQRT( ), da la raíz cuadrada de un número; devuelve un valor real. Sintaxis: Variable_Real := SQR(Número) ;

La función EXP( ), da el valor de ‘e’ elevado a la X; regresa un valor real. Sintaxis: Variable_Real := EXP(Número) ;

La función LN( ), da el logaritmo natural de un número, devuelve un valor real. Sintaxis: Variable_Real := LN(Número) ;

Por ejemplo:

```
PROGRAM Exp_Log ;
BEGIN
    WRITELN( ' Cuadrado 9.6 = ', SQR( 9.6 );
    WRITELN( ' Raíz 54 = ', SQRT( 54 );
    WRITELN( ' e ^ 4 = ', EXP( 4 );
    WRITELN( ' Logaritmo 5 = ', LN( 5 );
    WRITELN( ' 3 ^ 4 = ', Exp( 4 * Ln( 3 ) );
END.
```

- ✓ **Funciones aritméticas**

La función ABS( ), da el valor absoluto de un número o variable. Sintaxis: Variable := ABS(Valor) ;

El procedimiento DEC( ), decrementa el valor de una variable en cantidad de unidades especificados. Sintaxis: DEC(Variable, Valor) ;

El procedimiento INC( ), incrementa el valor de una variable en porción de unidades especificadas. Sintaxis: INC(Variable, Valor) ;

La función INT( ), obtiene la parte entera de un número real; devuelve un valor real. Sintaxis: Variable := INT(Valor) ;

La función FRAC( ), obtiene la parte decimal de un número real; regresa un valor real. Sintaxis: Variable := FRAC(Valor) ;

- ✓ **Funciones trigonométricas**

Seno, coseno y arcotangente son las únicas funciones trigonométricas que existen en Pascal, a partir de las cuales pueden calcularse las demás:

Función	Operación
Tangente(X)	Sin(x) / Cos(x)
Cotangente(X)	Cos(x) / Sin(x)
Secante(X)	1 / Cos(x)
Cosecante(X)	1 / Sin(x)

✓ **Números aleatorios**

La función RANDOM genera un número decimal comprendido entre 0 y 1. Si utilizamos un parámetro X, obtenemos un número entero entre 0 y X-1. El procedimiento RANDOMIZE alimenta de nuevo al semillero de números aleatorios en cada ejecución.

Por ejemplo:

```
PROGRAM Random_Randomize;
VAR
        i : Integer ;
BEGIN
        RANDOMIZE ;
        FOR i := 1 TO 20 DO
        WRITELN( ' Número : ', RANDOM( 100 ) ) ;
END.
```

4.2.2 Funciones propias

Aunque el lenguaje Pascal permite variedad de funciones internas que están listas para usarse y son las más empleadas en el campo científico y matemático, el programador puede declarar las propias (de modo similar a la declaración de procedimientos).

Como ejemplo tenemos una función para calcular el cuadrado de un número -en Pascal es estándar, pero en este caso la realiza el programador–:

```
FUNCTION cuadrado(a:REAL):REAL;
BEGIN
        cuadrado:=a*a
END;
```

La función es declarada real tanto en su variable ‘a’ como en su resultado; en el cuerpo de la función se asigna ‘a’ al producto de a*a, y ésta es el valor de la función ‘cuadrado’.

Para compartir procedimientos y funciones (subprogramas), el programador puede construir bibliotecas. Ahora bien, para usar las rutinas de la biblioteca escrita en Pascal, debe sustituirse una declaración externa por la común del procedimiento o función. El identificador extern ha de usarse como sigue:

```
PROGRAM lineas(INPUT,OUTPUT);
PROCEDURE trazo(a,b:real:pendiente:integer);extern
.
{comienza el programa principal}
begin
        trazo(a,b,1),
end.
```

El procedimiento para trazar una línea no aparece en el cuerpo del procedimiento –que se declara externo– porque está almacenado en una biblioteca de programas. Las variables que deben utilizarse para emplear subprogramas de biblioteca son los parámetros por valor y referencia (puntos 4.4.1 y 4.4.2 de este tutorial).

El recurso de las bibliotecas reviste un ahorro de tiempo considerable en el proceso de programación. Además, si se quiere que los programas estén disponibles para cualquier persona, hay que incorporar a la biblioteca general la documentación necesaria que detalle las actividades, parámetros y manera de emplearse.

4.3 Alcance de variables

En Pascal, cada identificador tiene sólo un campo de acción fuera del cual no es posible utilizarlo: esto se conoce como alcance de variables. Las variables definidas en el programa principal pueden usarse en todo el programa (incluidos subprogramas); el alcance de las definidas dentro de los subprogramas se limita sólo al mismo. Los ejemplos más claros son las variables locales y globales, que estudiamos a continuación.

4.3.1 Locales

Una variable local puede ser utilizada únicamente por el procedimiento en el que está declarada; el programa principal y los otros procedimientos la toman como inexistente. Ejemplo:

```
PROGRAM var_local;
VAR
bienvenida : String;
PROCEDURE despedida;
VAR
adios : String;
BEGIN
        adios=’Adiós, vuelva pronto’;
        WRITELN(adiós);
END;
BEGIN
        bienvenida:=’Hola, bienvenidos al programa’;
        WRITELN(bienvenida);
        despedida;
        WRITELN(adiós)
        {La última línea genera error de compilación}
END.
```

La variable ‘adios’ se declara dentro del procedimiento ‘despedida’, lo que origina que dicha variable sea local y no reconocida por el programa principal (‘var_local’).

Se declaran la variable ‘bienvenida’ de tipo cadena y posteriormente el procedimiento ‘despedida’ (sólo imprime en pantalla la cadena ‘Adiós vuelva pronto’). En el cuerpo del programa principal, a la variable ‘bienvenida’ se le asigna una cadena de texto y luego se imprime en pantalla; se invoca al procedimiento ‘despedida’, así, es posible imprimir el contenido de la variable local ‘adios’, porque está declarada dentro del procedimiento llamado; sin embargo, al tratar de imprimir ‘adios’ en el programa principal, hay error de compilación (envía el mensaje ‘identificador no encontrado’), porque la variable ‘adios’ no existe para el programa principal, es local.

Para corregir el programa anterior, debe borrarse la línea WRITELN(adiós).

4.3.2 Globales

Una variable global puede ser utilizada por cualquier parte del programa, incluyendo todos los procedimientos. Por ejemplo:

```
PROGRAM var_global;
VAR
hola : String;
PROCEDURE dentro;

BEGIN
        hola := 'Hola, estoy dentro del procedimiento';
        WRITELN(hola);
END;
```

```
BEGIN
        dentro;
        hola := 'Ahora, estoy dentro del programa principal';
        WRITELN (hola);
END.
```

En este programa, se declara la variable ‘hola’ como tipo cadena y luego el procedimiento ‘dentro’, que usa la global ‘hola’ del programa principal. El procedimiento asigna una cadena de texto a dicha variable y luego la imprime.

En el cuerpo del programa principal, se invoca al procedimiento ‘dentro’, que imprime en pantalla la cadena ‘Hola, estoy dentro del procedimiento’; después, a la misma variable ‘hola’ se la asigna otra cadena de texto y luego se manda imprimir en pantalla, que da como resultado: ‘Ahora, estoy dentro del programa principal’. Al declararse en el programa principal, la variable ‘hola’ se convierte en global, lo que significa que puede usarse tanto en el mismo programa como en todos los procedimientos empleados.

Además, es posible darle el mismo nombre a una variable local y a una global en el mismo programa, pero entonces el procedimiento no podrá utilizar la global, ya que da preferencia a locales sobre globales.

Por ejemplo:

```
PROGRAM var_loc_glob;
VAR
        hola : STRING;

PROCEDURE dentro;

VAR
        hola:STRING;
```

```
BEGIN
        hola := 'Hola, soy una variable local';
        WRITELN(hola);
END;

BEGIN
        hola:='Hola, soy una variable global';
        WRITELN(hola);
        dentro;
        WRITELN (hola);
END.
```

El programa anterior da como salida:

```
        Hola, soy una variable global
        Hola, soy una variable local
        Hola, soy una variable global
```

La salida anterior se debe a que ha sido declarada la variable ‘hola’, de tipo cadena, en la zona de declaraciones del programa principal. Después, se realiza el procedimiento ‘dentro’, donde, al declararse otra variable también nominada ‘hola,’ aparece otra local. De este modo, tenemos un par de variables con el mismo nombre, una global y otra local.

En el cuerpo del programa principal se hace la asignación de una cadena a la variable global ‘hola’ y se imprime en pantalla; posteriormente, se invoca al procedimiento ‘dentro’, que imprime la variable local ‘hola’ –que tiene diferente contenido respecto de la primera variable–; y luego se vuelve a imprimir la global ‘hola’.

4.4 Pase de parámetros

Los parámetros sirven para pasar información del programa principal a los procedimientos o funciones, o entre los mismos subprogramas. Son las “vías de comunicación” opcionales de los datos; no deben usarse en exceso para no complicar la codificación del programa. Se declaran bajo la siguiente sintaxis:

PROCEDURE nombre (parámetros : tipo de datos);

Los parámetros están compuestos por los nombres de los mismos y el tipo de datos que representan; los de igual tipo se separan por medio de comas "**,**", y los diferentes, con punto y coma "**;**". Ejemplo:

Procedure Practica(r, s : Real; x, y : Integer);

Para invocar un procedimiento que utiliza parámetros, pueden utilizarse como tales otras variables o constantes, siempre y cuando sean del mismo tipo que los declarados. Ejemplo:

Practica(4.56, 2.468, 7, 10);

Ejemplo de un programa con procedimiento que utiliza un parámetro:

```
PROGRAM Parametros;
VAR
    titulo : String;
PROCEDURE encabezado (a : String);

BEGIN
    WRITELN(a);
END;

BEGIN
    titulo := 'Compañía X, S.A. de C.V.';
    encabezado(título);
    encabezado('Relación de ventas del mes');
END.
```

Lo anterior imprime en pantalla:
'Compañía X, S.A. de C.V
'Relación de ventas del mes

En la primer llamada al procedimiento, ‘encabezado’, el parámetro ‘titulo’ le pasa al ‘a’ del procedimiento; en la segunda, se comunica una cadena como constante ‘Relación de ventas del mes’.

Es válido crear un procedimiento que llame a otro, siempre y cuando el que es llamado haya sido declarado con anticipación por quien lo usará.

4.4.1 Por valor

Los parámetros por valor no permiten cambios del original. Se utilizan cuando dichos parámetros deban protegerse contra cualquier tipo de cambio.

Los parámetros por valor operan haciendo una `copia’ del original para llevarlo al procedimiento. El parámetro original queda como inaccesible para el procedimiento; no así el parámetro copiado, que acepta cualquier modificación, sin afectar al original.
Por ejemplo:

```
PROGRAM porvalor;
VAR
        par_orig:INTEGER;

PROCEDURE ejemplo(y:integer);
BEGIN
        y:=y*y,
        WRITELN(y)
END;
```

```
BEGIN
        par_orig:=5;
        ejemplo(par_orig);
        WRITELN(par_orig)
END.
```

La salida es:

25

5

En el caso anterior, la variable par_orig se declara como entera. Luego, se expresa el procedimiento ‘ejemplo’ –que maneja el parámetro ‘y’ como entero–; el procedimiento obtiene el cuadrado de la variable ‘y’ y lo imprime en pantalla.

Además, en el programa principal se le asigna a la variable par_orig un valor de 5, que posteriormente pasa al procedimiento ‘ejemplo’ y hace una copia que toma la variable ‘y’; al ser invocado el procedimiento, aparece en la impresión de pantalla el número 25 (cuadrado de 5). Como podemos apreciar, la variable ‘y’ del procedimiento ha sufrido cambio, sin embargo, la instrucción que le sigue a la llamada del procedimiento imprime la variable par_orig, que sigue teniendo el valor 5 (el valor original no ha sido modificado).

En este caso, como el valor original no se altera, los resultados no pueden retornarse al programa principal para ser guardados en los parámetros originales; éste es su inconveniente.

4.4.2 Por referencia

En el pase de parámetros por referencia, el subprograma puede obtener del programa principal los valores de los parámetros originales y cambiarlos por los que arroje como resultado el subprograma. El parámetro original es totalmente

reemplazado por el resultado del procedimiento o función. De este modo, cualquier modificación al parámetro del subprograma altera los originales del programa “llamador”, que pierden protección.

Para especificar el pase por referencia, debe definirse la palabra reservada VAR antes del parámetro variable en la declaración del procedimiento o función. Por ejemplo:

```
PROGRAM por_ref;
VAR
        par_orig:INTEGER;

PROCEDURE ejemplo(VAR y:integer);
BEGIN
        y:=y*y,
        WRITELN(y)
END;

BEGIN
        par_orig:=5;
        ejemplo(par_orig);
        WRITELN(par_orig)
END.
```

La salida es:

25

25

En el mismo programa del apartado anterior, sólo con ‘y’ declarada como parámetro variable, el resultado cambia, ya que el valor original par_orig, es

reemplazado por ‘y’ dentro del procedimiento y da un resultado que es almacenado como nuevo valor en par_orig.

Por conveniencia, deben emplearse parámetros de valor a un procedimiento o función y por referencia para retornar su valor.

4.5 Funciones recursivas

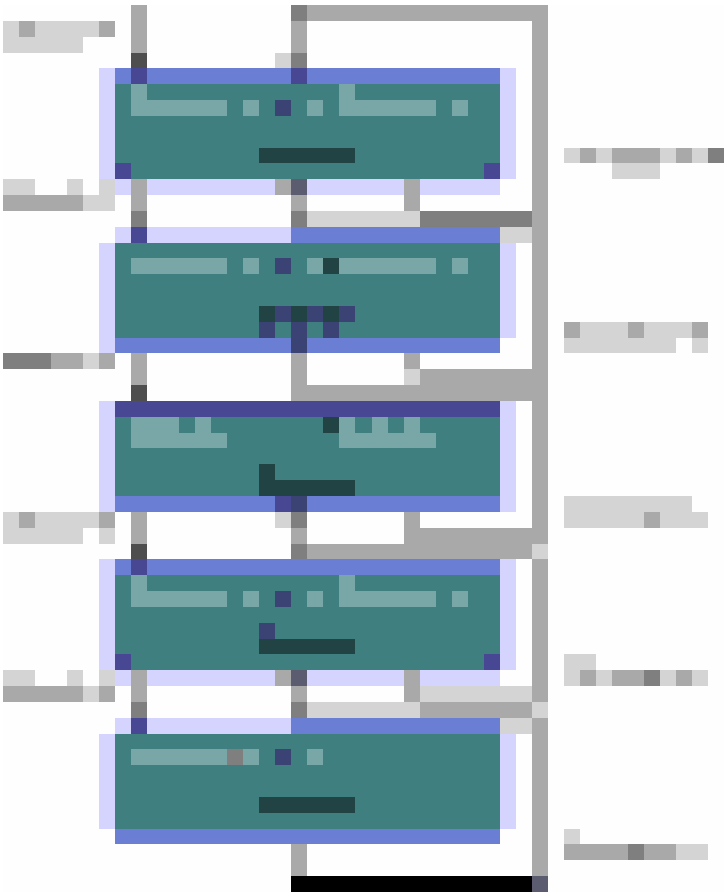
Los procedimientos y las funciones se pueden invocar a sí mismos; en el cuerpo de sus instrucciones, deben tener una línea de código por medio de la cual invoquen su mismo nombre. Este proceso se llama recursión.

La función recursiva más común es la que calcula el factorial de un número entero positivo. Entonces, debemos tener en cuenta las siguientes consideraciones:

- El factorial de cero es igual a uno.
- El factorial de un número n es igual a n * factorial(n-1), para cualquier número entero mayor a cero.

```
FUNCTION factorial(número:integer):integer;
BEGIN
    IF numero = 0 THEN
        factorial := 1
    ELSE
        factorial := numero * factorial(numero -1)
END;
```

Si la variable ‘numero’ es igual a 4, la función realiza lo siguiente:



En la línea factorial(4)= 4 * factorial(3), la función se invoca a sí misma, pero con valor 3 en la variable ‘numero’. A la vez, esta función se autoinvoca, con valor 2 en ‘numero’. Asimismo, se vuelve a invocar, pero con ‘numero’ 1… La última autollamada la realiza cuando ‘numero’ es igual a cero; entonces, el valor de la última función factorial es 1. Éste se retorna a la función de llamada anterior, que toma el valor 1 y realiza el cálculo factorial=2*1. Luego, el valor de ésta función es 2, que regresa a la función de llamada y lleva a cabo el cálculo factorial=3*2; esta función tiene valor 6, que retorna a la primer función de llamada y ejecuta la expresión factorial=4*6, dando como resultado final 24, como factorial del número 4.

Las llamadas recursivas son un ciclo en el que se van llamando programas de niveles inferiores (funciones, en el ejemplo), hasta que se cumpla la condición numero=0. Entonces, el valor de la última función llamada adquiere el valor 1, que es

retornado y asciende a los programas superiores de llamada; así, va generando los resultados 2, 6 y 24.

También una función o procedimiento puede llamar a otro subprograma y éste a otro… y así sucesivamente. Si el último subprograma hace una llamada a la primera función o procedimiento, se cierra el ciclo. A este proceso lo conocemos como recursividad indirecta.

En la siguiente estructura de procedimiento, tenemos un ejemplo de recursividad indirecta:

```
PROGRAM rec_ind;
PROCEDURE a;
        BEGIN       invoca al procedimiento b END;
PROCEDURE b;
        BEGIN       invoca al procedimiento c END;
PROCEDURE c;
        BEGIN       invoca al procedimiento d END;
PROCEDURE d,
        BEGIN if condicion THEN invoca al procedimiento a END.

BEGIN       invoca al procedimiento a END.
```

Así pues, la recursividad es una herramienta potente que ahorra mucha codificación; pero debemos tener cuidado de no usarla en exceso y complicar así el entendimiento del programa.

Direcciones electrónicas

http://www-gris.ait.uvigo.es/~belen/isi/doc/tutorial/unidad4p.html
http://luda.uam.mx/cursoc1/pascal/tema3/tecp02.html#ppr
http://quantum.ucting.udg.mx/uctn/cursos/pascal/unidad4p.html#u3.2

http://www.itlp.edu.mx/publica/tutoriales/pascal/u6_6_0.html
http://members.nbci.com/tutoriales/unidad4.htm
http://www.itlp.edu.mx/publica/tutoriales/pascal/u6_6_1.html
http://www.itlp.edu.mx/publica/tutoriales/pascal/u6_6_2.html
http://members.nbci.com/tutoriales/unidad5.htm
http://www.itlp.edu.mx/publica/tutoriales/pascal/u6_6_3.html
http://www.itlp.edu.mx/publica/tutoriales/pascal/u6_6_4.html
http://www.itlp.edu.mx/publica/tutoriales/pascal/u6_6_4_1.html
http://www.itlp.edu.mx/publica/tutoriales/pascal/u6_6_4_2.html
http://www.itlp.edu.mx/publica/tutoriales/pascal/u6_6_5.html

Bibliografía de la Unidad

Unidad 5. Definición de tipos

Temario detallado

5. Definición de tipos
    - 5.1 Escalares
    - 5.2 Subintervalos
    - 5.3 Arreglos
        - 5.3.1 Unidimensionales
        - 5.3.2 Multidimensionales
    - 5.4 Registros.

5. Definición de tipos

La definición de tipos consiste en que el lenguaje Pascal nos permite configurar algunas propiedades de tipos abstractos para su manipulación posterior. Hay tipos de variables estándar, como real, integer, boolean y char, de los cuales Pascal ya tiene determinadas sus características. Esto nos brinda la ventaja de poder expresar los problemas de modo más claro si empleamos definiciones de tipo adecuadas; además, podemos dar más información al compilador acerca de nuestro caso, que puede usar para verificar errores.

La definición de tipos se manifiesta junto con las declaraciones de constantes y variables. Las definiciones de tipo declaradas dentro de procedimientos son locales a éstos.

5.1 Escalares

La declaración de una variable de tipo escalar es cuando ésta toma sus valores de una lista de datos. Ejemplo:

```
TYPE
    oficina = (cuadernos, plumas, gomas);
```

La variable “oficina” puede tomar cualquiera de los valores “cuadernos”, “plumas” o “gomas” y nada más. El tipo “oficina” puede ser usado en la declaración de una variable como si fuera tipo estándar, como sigue:

```
VAR
    consumibles : oficina;
```

Algunos ejemplos de declaración de tipo escalar:

```
TYPE
    meses = (enero, febrero, marzo, abril, mayo, junio, julio, agosto, septiembre,
        octubre, noviembre, diciembre);
    leyes = (isr, cff, iva);
    inventario = (mesa, silla, banca);
```

Una vez declarados los tipos, pueden expresarse las siguientes variables:

```
VAR
    ejercicio, anual : meses;
    marco_legal : leyes
    almacen, lote : inventario;
```

Un valor no puede pertenecer a más de una lista de valores dentro de un tipo. El siguiente ejemplo causa un error de ejecución porque el valor “discos” se usa en dos listas distintas:

```
TYPE
    pc = (pantalla, teclado, mouse, discos);
    computadora = (bocinas, micrófono, discos, escáner);
```

Los valores listados en la declaración de un tipo escalar son constantes de ese tipo. Entonces, podemos escribir:

ejercicio := octubre;
marco_legal := iva;
lote := mesa;

Los valores declarados en un tipo escalar, relacionados con una variable, no pueden asignarse a variables declaradas con otro tipo escalar. Ejemplo:

ejercicio := mesa;

Los operadores de relación pueden emplearse con las variables escalares para dar un resultado booleano. Retomando como base los ejemplos anteriores, el resultado de las siguientes expresiones es verdadero:

enero > diciembre
isr > iva
silla < mesa

Hay funciones que trabajan con argumentos escalares, como ORD, SUCC y PRED, que explicamos a continuación:

La función ORD regresa la posición de un valor de la lista. Ejemplo:

ORD(mayo) = 4
ORD(iva) = 2
ORD(mesa) = 0

Las funciones PRED y SUCC dan como resultado el mismo tipo de los argumentos. Sirven para obtener un valor predecesor o uno sucesor del valor actual de la lista. Ejemplo:

SUCC(enero) = febrero
PRED(silla) = mesa

No pueden imprimirse los valores escalares. Aunque no haya error de compilación, la siguiente instrucción WRITELN no genera resultado alguno:

ejercicio := junio;
WRITELN(ejercicio)

Se puede utilizar WRITELN(ORD(ejercicio)) para dar como resultado 5.

Las variables escalares pueden emplearse en los ciclos FOR, de la siguiente manera:

FOR prueba := enero TO diciembre DO
    saldo

En el ejemplo anterior, “enero” es tomado como 0 y “diciembre” como 11. Es decir, el ciclo FOR repetirá 12 veces el procedimiento “saldo”. No sucede lo mismo con el ciclo WHILE, ya que aquí no puede ser usado este tipo de variables.

5.2 Subintervalos

Un tipo subintervalo es aquel que define un rango de valores; usa dos constantes, una, marca el límite inferior y otra, el superior del rango. Ejemplo:

```
TYPE
rango = 1..100;
```

Los tipos permitidos de las constantes son escalares, integer o char. No se pueden utilizar subintervalos de tipo real. Ejemplo:

```
TYPE
alfa = 'a'..'z';
num = 1..100;
ejerc_fiscal = enero..diciembre;
```

"ejerc_fiscal" es un subintervalo del tipo escalar "meses", anteriormente definido. Las variables de subintervalo deben ser declaradas en la sección de declaración de variables. Ejemplo:

```
VAR
alfabeto, letras : alfa;
intervalo : num;
```

Estos mismos ejemplos también pueden definirse de la forma siguiente:

```
alfabeto, letras    :   'a'..'z';
intervalo           :  1..100;
```

Lo anterior no es muy recomendable, porque primero debe declararse el tipo de subintervalo y después las variables de este tipo.

Si pretendemos asignar a una variable de subintervalo un valor más allá de su rango, habrá error en la compilación, porque está fuera de rango. Por ejemplo, en un rango declarado de 1 a 100 el valor 101 seguramente causará error al compilar el programa.

Las declaraciones de subintervalo le facilitan la comprensión del programa al usuario, pues le dan a conocer el rango de valores que puede tener una variable.

5.3 Arreglos

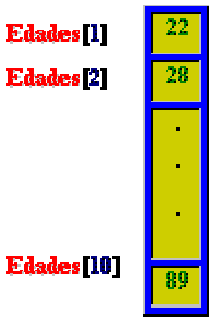
Un arreglo es un conjunto de variables del mismo tipo y nombre, identificadas con un número denominado índice. Así, el manejo del programa resulta más claro y entendible. Hay arreglos uni y multidimensionales.

5.3.1 Unidimensionales

Un arreglo unidimensional o vector es el que tiene una sola dimensión. Para referirnos a un elemento del arreglo, debemos indicar su índice (número de posición que ocupa dentro del arreglo). El arreglo tiene la siguiente sintaxis:

```
VAR
        identificador : ARRAY [valor inicial .. valor final] OF tipo
```

Donde “tipo” puede ser cualquier tipo de datos como integer, char real, etcétera.



Por ejemplo, supongamos que deseamos guardar las edades de 10 personas. Para hacerlo, damos un nombre al arreglo (array), definimos el número de posiciones que va a contener y almacenamos en cada una de ellas la edad de la persona. Veámoslo gráficamente:

Así, el nombre del vector es “edades”, los índices son [1], [2]... y su contenido es edades[10]=89.

Se puede trabajar con elementos del vector individualmente, la siguiente línea asigna al elemento que ocupa la posición 10 del vector “A”, el valor 20.

A[10] := 20;

Como ejemplo tenemos el siguiente programa que utiliza un vector con 10 elementos o datos y devuelve el promedio de los mismos:

```
PROGRAM promedio (INPUT, OUTPUT);
VAR
        lugar : INTEGER;
        suma : REAL;
        X : ARRAY [1 .. 10] OF REAL;

BEGIN
        suma := 0.0;
        FOR lugar := 1 TO 10 DO
                BEGIN
                        WRITELN ('Ingrese dato ', lugar);
                        READLN (X[lugar]);
                        suma := suma + X[lugar];
                END;
        FOR lugar := 1 TO 10 DO
                WRITELN (X[lugar]);
        WRITELN ('El promedio del arreglo es: ', suma/10);
        READLN
END.
```

Se declaran las variables “lugar” como integer, “suma” como real y “X” como arreglo de tipo real. Comienza el programa y se inicializa la variable “suma” en 0; en el siguiente ciclo FOR, se solicita al usuario ingresar un dato el cual es almacenado en la posición de X[lugar]. Si sabemos que la variable “lugar” tendrá los valores del 1 al 10, entonces, en estas posiciones relativas del arreglo “X” se irán almacenando los

datos del usuario. La variable “suma” se emplea después para acumular el valor de la posición del arreglo “X” más el valor que traiga la misma variable “suma”.

Posteriormente, continúa otro ciclo FOR: primero se exhiben en pantalla todos los valores almacenados en el arreglo “X” –mediante la instrucción WRITELN(X[lugar])– y después se imprime el promedio de los números ingresados dividiendo el valor acumulado de la variable “suma” entre 10.

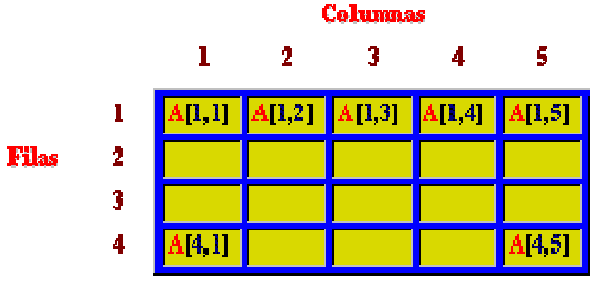
5.3.2 Multidimensionales

Un arreglo multidimensional es el que tiene dos o más dimensiones. Un arreglo con dos índices es bidimensional, se le conoce también como tabla o matriz. Éste necesita los índices del renglón y de la columna para definir la posición de cualquiera de sus elementos. La posición del elemento nos sirve para guardar y recuperar cualquier valor de éste. Su sintaxis es la siguiente:

```
VAR
        identificador : ARRAY [[valor inicial..valor final] ,
                               [valor inicial..valor final]] OF tipo
```

Donde “tipo” puede ser cualquier tipo de datos como integer, real, char, etcétera.

Para localizar un determinado valor se debe dar como coordenada los índices de filas y columnas:



Entonces, para asignar un valor a un elemento de la matriz, tenemos que hacer referencia a ambos índices, como lo muestra la siguiente línea: A[4,5] := 40;

Aquí se almacena el número 40 en la posición de la fila 4 con la columna 5 de la matriz “A”.

Como ejemplo tenemos este programa, que lee una matriz de 3 filas y 3 columnas, y después exhibe en pantalla el contenido de la matriz fila por fila.

```
PROGRAM Multidim (INPUT, OUTPUT);

CONST
    fila = 3;
    columna= 3;
VAR
    i, j : INTEGER;
    X:ARRAY[1..fila,1..columna] OF INTEGER;
BEGIN
    FOR i:=1 TO fila DO
    FOR j:=1 TO columna DO
    BEGIN
        WRITELN ('Ingrese el valor a (Fila: ',i,' y Columna: ',J,')');
        READLN (X[i, j]);
    END;

    WRITELN ('La matriz contiene los valores siguientes:');
    FOR i:=1 TO fila DO
    BEGIN
    FOR j:=1 TO columna DO
        WRITE(X[i, j],' , ');
        WRITELN
    END;
READLN
END.
```

El programa anterior acepta los valores de la matriz que introduce el usuario y luego los imprime en pantalla. Primero, se declaran las constantes fila y columna, ambas con valor 3; después, las variables “i” y “j” como enteras y la variable “X” como arreglo multidimensional de tipo entero.

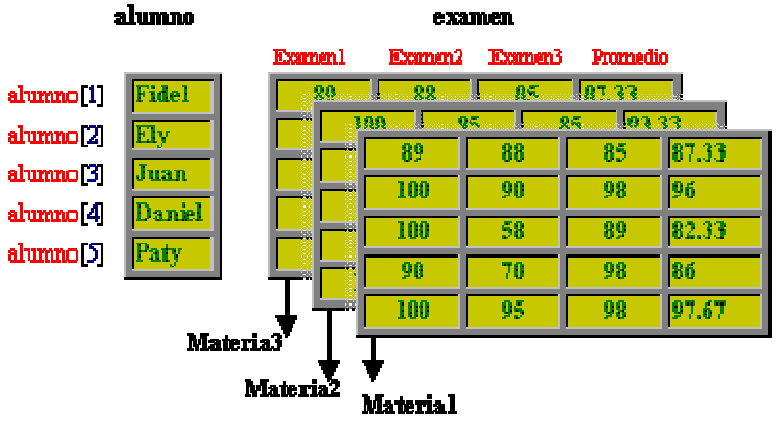
Comienza el programa con el ciclo FOR, controlado por la variable “i” que ejecutará el ciclo interno FOR 3 veces; el ciclo anidado FOR para la variable “j” solicitará al usuario que ingrese para la fila y columna (i,j) un valor de tipo entero. Los siguientes ciclos FOR escribirán en pantalla el contenido de la matriz “X” por medio de la instrucción WRITE(X[i, j],' , '), y las variables “i” y “j” tomarán su valor de los ciclos FOR en el que están contenidos. La salida será un desplegado de valores que ingresó previamente el usuario al programa, en forma de matriz.

En un arreglo que contenga más de dos dimensiones, debe declararse el tipo de cada subíndice. Opera bajo la siguiente sintaxis:

identificador = array [índice1] of array [índice 2] of array
[índice n] of tipo de elemento

identificador = array [índice 1, índice 2,...,índice n] of tipo de elemento

En el siguiente ejemplo usamos un arreglo tridimensional. Este arreglo debe ser capaz de contener los nombres de los alumnos, sus calificaciones de todos los exámenes y el promedio de cada una de sus materias:



Las siguientes declaraciones pueden formar parte del programa para los arreglos de este tipo. La declaración de constantes es:

```
Const
  MaxAlumno  = 5;
  MaxExamen  = 4;
  MaxMateria = 3;
  materia    : array[1..3] of string[8]=('Física','Inglés','Historia');
```

Y la declaración de variables:

```
Var
  Alumno     : array [1..MaxAlumno] of string[10];
  examen     : array [1..MaxAlumno,1..MaxExamen,1..MaxMateria] of real;
  aux_examen : array [1..MaxExamen] of real;
```

Con lo anterior, reservamos 60 posiciones de memoria de datos reales : 5 filas por 4 columnas y 3 dimensiones.

5.4 Registros

Un registro es una variable estructurada con una cantidad finita de componentes denominados campos. Éstos pueden ser de diferente tipo y deben tener un identificador de campo. A diferencia del arreglo, los registros son precisados por medio del nombre y no por un índice.

Un registro utiliza en su definición las palabras reservadas RECORD y END. Los campos estarán listados entre esas dos palabras. El formato que se emplea, en este caso, es:

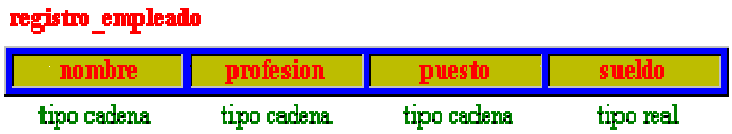
```
type
  tipo_reg = record
              lista id1:tipo 1;
              lista id2:tipo 2;
                    .
                    .
                    .
              lista idn:tipo n
            end;
```

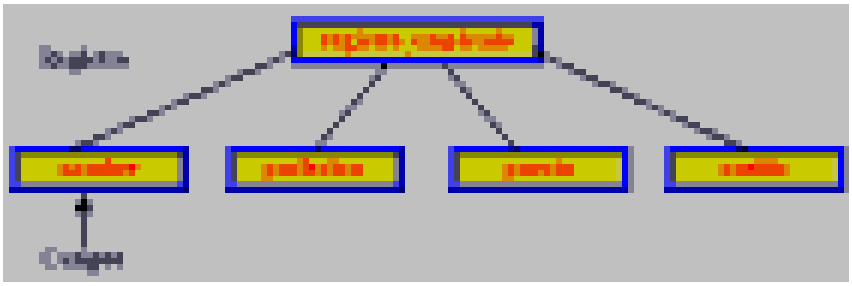
tipo_reg **nombre de la estructura o dato registro**

lista id **lista de uno o más nombres de campos separados por comas**

tipo **puede ser cualquier tipo de dato estándar o definido por el usuario**

Como ejemplo, declaramos un registro llamado registro_empleado, que contendrá los datos generales de nuestros empleados, agrupados en los siguientes campos: nombre, profesión, puesto y sueldo. La representación gráfica queda así:





Veamos un ejemplo más:

Se declara el tipo “auto” como un registro formado por tres campos, a saber:

```
TYPE
  auto=   RECORD
              pieza          :     ARRAY [1..250] OF char;
              pza_defectuosa:      Boolean,
              peso           :      real
          END;
```

Así, podemos describir al registro “auto” por el conjunto de sus campos. En este ejemplo, el campo “pieza” se usa para hacer una descripción breve de una parte del auto, el estado de las piezas. Se prueba con el tipo booleano true / false del campo “pza_defectuosa” y el campo “peso” contendrá una cantidad real.

Las variables se declaran como sigue:

```
VAR
  motor,estructura:auto;
```

Se accede a un campo usando el nombre de la variable de tipo arreglo y el nombre del campo, ambos separados por un punto. Ejemplo:

motor.pieza	es el nombre de la pieza del motor
estructura.pza_defectuosa	si la pieza de la estructura está defectuosa o no

Las siguientes asignaciones son correctas:

```
motor.pieza          :=   ’disco de embrague   ‘;
estructura.pieza     :=   ’salpicadera delantera derecha...’;
motor.pza_defectuosa :=   true;
estructura.peso      :=   20.50,
```

Es posible asignar los valores de una variable a otra:

motor := estructura

Lo anterior asigna los valores de todos los campos de la variable “estructura” a cada uno de los campos de la variable “motor”. El nombre de los campos dentro de los registros no puede estar repetido; aunque puede usarse una misma nominación en una variable o en otro registro.

Regularmente, se ingresa a los mismos campos de un registro, pero es una tarea tediosa. Con el uso de las instrucciones WITH y DO, se ahorra bastante codificación y facilita en gran medida la manipulación de los registros. Su sintaxis es:

```
WITH variable de tipo registro DO
proposiciones
```

En las proposiciones se emplea sólo el nombre del campo, por lo que se evita escribir por cada campo el nombre del registro. Por ejemplo, para poder leer datos usamos:

```
WITH motor DO
BEGIN
    WRITE('PIEZA : ');
    READLN(pieza);
    WRITE('DEFECTUOSA? : ');
    READLN(pza_defectuosa);
    WRITE('PESO : ');
    READLN(peso);
END;
```

Como podemos observar, no hay necesidad de teclear motor.pieza o motor.peso para identificar el campo del registro, basca con usar el nombre del campo. Debemos tomar en cuenta el siguiente programa para demostrar el uso de WITH DO:

```
PROGRAM Registro;
```

```
{programa que lee 3 registros que contienen los datos generales de trabajadores}

USES Crt; {el programa prepara la librería estándar Crt para ser usada}
CONST
        MaxEmpleados=3;
TYPE
        empleado = RECORD {se declara el tipo empleado}
                        nombre:string[30];
                profesion:string[20];
                        puesto:string[20];
                sueldo:real
                END;
VAR
        reg:empleado; {la variable reg se declara como tipo arreglo}
        i,col,ren:byte;
BEGIN
      CLRSCR; {función que borra la pantalla}
      WRITE('  Nombre          Profesión          Puesto          Sueldo');
      col:=1;ren:=2;
      FOR i:=1 TO MaxEmpleados do {ciclo que se ejecuta 3 veces}
      BEGIN
        WITH reg DO {se usa with…do para simplificar la lectura de campos}
        BEGIN
                    GOTOXY(col,ren); {posiciona cursor en coordenada col,ren}
                    READLN(nombre); {lee el campo nombre de variable reg}
                    GOTOXY(col+21,ren);
                    READLN(profesion);
                    GOTOXY(col+40,ren);
                    READLN(puesto);
                    GOTOXY(col+59,ren);
                    READLN(sueldo);
                    INC(ren); {función que incrementa ren en 1}
                    col:=1;
        END
      END;
      READLN
END.
```

Este programa carga los datos a los campos de 3 registros de empleados y los exhibe en pantalla en forma tabular. Primero se declara el tipo empleado con sus campos de nombre, profesión, puesto y sueldo, y luego se usa para declarar la variable “reg” que contendrá la información general del empleado. El ciclo FOR se ejecuta 3 veces; dentro de éste se procesan WITH y DO con la variable de tipo registro “reg” para facilitar la captura de datos a cada uno de los campos contenidos en la variable. Como podemos apreciar, la labor de codificación se ve simplificada.

Como complemento a la explicación en el programa se usan las llaves { }, para poder hacer comentarios al programa; éstas no se toman en cuenta en la ejecución. También se hace uso de la biblioteca de funciones estándar de Pascal, llamada Crt, que contiene funciones rutinarias.

Direcciones electrónicas

http://www.fi.uba.ar/ftp.pub/materias/7540/apuntes/pascal1.htm
http://www.itlp.edu.mx/publica/tutoriales/pascal/u5_5_1.html
http://luda.uam.mx/cursoc1/pascal/tema6/clp02.html#arreglos
http://www.di-mare.com/adolfo/binder/c03.htm
http://www.itlp.edu.mx/publica/tutoriales/pascal/u5_5_2_2.html
http://physinfo.ulb.ac.be/cit_courseware/pascal/pas053.htm
http://www.itlp.edu.mx/publica/tutoriales/pascal/u5_5_2_3.html

Bibliografía de la Unidad

Unidad 6. Manejo dinámico de memoria e introducción a la implantación de estructuras de datos

Temario detallado

6. Manejo dinámico de memoria e introducción a la implantación de estructuras de datos
   - 6.1 Apuntadores
   - 6.2 Listas ligadas
   - 6.3 Pilas
   - 6.4 Colas
   - 6.5 Árboles binarios

6. Manejo dinámico de memoria e introducción a la implantación de estructuras de datos

Los ejemplos clásicos de las estructuras de datos que no cambian su tamaño durante la ejecución del programa son los arreglos (*arrays*) y los registros (*records*). En éstas, la cantidad de memoria asignada se determina cuando el programa se compila (este tipo de almacenamiento es llamado de asignación estática). También hay estructuras dinámicas como apuntadores, listas ligadas, colas, pilas y árboles binarios, que permiten asignar localidades de memoria cuando el programa es ejecutado; este proceso se conoce como asignación de almacenamiento dinámico, que produce una lista de registros ligados unos con otros (así, algunos registros pueden ser insertados o removidos de cualquier punto en la lista, sin alterar en lo mínimo a los demás).

6.1 Apuntadores

Un apuntador es una variable que se usa para almacenar una dirección de memoria que corresponde a una variable dinámica (que se crea y se destruye cuando se ejecuta el programa). Por medio del apuntador, accedemos a la variable ubicada en la dirección de memoria contenida en el mismo. Es decir, el valor de la variable tipo apuntador no es otra cosa que una dirección de memoria. Se dice que el apuntador señala la variable almacenada en la dirección de memoria que la contiene; sin embargo, lo importante es obtener el valor de esa variable. No hay que confundir la variable apuntada con el apuntador.

Para comprender mejor lo anterior, consideremos un tipo de arreglo y las variables declaradas para ser usadas en el programa. Nuestro arreglo se define como:

```
TYPE
prueba = ARRAY[1..50] OF INTEGER
```

Bien, de acuerdo con el ejemplo, la computadora reserva 50 localidades para cada arreglo declarado de tipo “prueba”. Si se usa un límite arriba de 50, el compilador desplegará un mensaje de error, porque está fuera de rango. Por el contrario, el método de asignación dinámica nos permite adicionar y eliminar localidades según lo requiera el programa, ya que éstas no están definidas previamente en un cierto número (como sucede con los arreglos). Lo anterior, porque las localidades de memoria pueden ser usadas dinámicamente en la ejecución del programa, y cada una de ellas tiene una dirección numérica interna a la que la computadora puede referirse. Las direcciones internas no están disponibles para el programador, pero es posible instruir a la computadora para que asigne una dirección de localidad disponible a un tipo de variable (apuntador).

Un tipo apuntador se define de la forma siguiente:

```
TYPE
    Tipo Apuntador = ^Tipo de Variable Apuntada;
```

Un tipo de apuntador para una localidad entera se define como:

```
TYPE
    Apuntador_entero = ^Integer;
```

El acento circunflejo (^) antes de INTEGER significa apuntador a enteros. Podemos definir como tipo de apuntador a cualquier tipo, inclusive a uno estructurado. Posteriormente, declaramos la variable:

```
VAR
    Apuntador : Apuntador_entero;
```

Un apuntador también puede ser asignado a otro apuntador. Por ejemplo, si A y B son dos variables de tipo Apuntador_Entero, podemos tener que A:=B; de este modo, A contendrá la misma dirección de memoria que B. Además, podemos usar la constante NIL para indicar que un apuntador no señala alguna localidad:

```
    A := NIL;
```

Empleamos “=” para hacer que dos apuntadores señalen a la misma dirección y “<>” para que no lo hagan.

A la variable señalada por un apuntador se le llama referenciada. Para accederla, debemos seguir estos criterios: si el apuntador es denotado por A, entonces la variable referenciada es A^. Por ejemplo, si A es del tipo de un apuntador entero, podemos usar A^ como si fuera una variable cualquiera de tipo INTEGER.

Pero si A y B son de tipo apuntador entero, con la expresión A := B hacemos que A apunte a la misma dirección que B; mientras que con A^ := B^ el entero apuntado por A toma el valor del entero apuntado por B. Ejemplo:

```
TYPE
    apuntador = ^Integer;
VAR
    A , B : apuntador;
```

De este modo, A y B están declaradas como variables de tipo apuntador. Pero aún no se establece alguna asociación entre esas variables y las direcciones disponibles en la memoria. Debemos usar el procedimiento estándar NEW, en la forma NEW(A), en donde la dirección de la primera localidad disponible en la memoria es asignada a la variable apuntador “A”. Si es un apuntador de tipo entero, la localidad estará disponible para almacenar un valor de esa característica, por lo que la cantidad de espacio en memoria es para un entero.

Ejemplificamos el proceso de asociación de localidad de memoria por NEW. Supongamos que escribimos un valor como 100 en una localidad, asignamos dicho valor a la localidad indicada por el apuntador. La localidad indicada por “A” es escrita como A^, y la asignación es:

```
BEGIN
    NEW (A);          asocia la localidad de memoria disponible al puntero A
    A^ := 100;        asigna 100 como contenido a una dirección de memoria
    WRITELN(A^);      escribirá el número100,
```

Sin embargo, WRITELN(A) causará error de compilación, porque una dirección de memoria no está disponible para el programador.

La diferencia entre usar NEW(A) y A^:=50, es que la primera asigna la siguiente dirección de memoria disponible de la computadora al apuntador A; y la

otra, un nuevo valor (en este caso, 50) a la dirección de memoria referenciada por el apuntador A. Veámoslo en el siguiente ejemplo:

```
PROGRAM prueba (INPUT,OUTPUT);
TYPE
        apuntador=^INTEGER;
VAR
        A,B: apuntador;
BEGIN
        NEW (A);
        NEW (B);
        A^:=50;             {Dirección de memoria 1 de A guarda un 50}
        B^:=100;            {Dirección de memoria 2 de B guarda un 100}
        WRITELN (A^);       {Imprime contenido de dirección 1: 50}
        WRITELN (B^);       {Imprime contenido de dirección 2: 100}
        NEW(A);             {Asigna dirección de memoria 3 al apuntador A}
        A^:=150;            {Dirección de memoria 3 de A guarda un 150}
        B^:=200;            {Dirección de memoria 2 de B guarda un 200}
        WRITELN (A^);       {Imprime contenido de dirección 3: 150}
        WRITELN (B^)        {Imprime contenido de dirección 2: 200}
END.
```

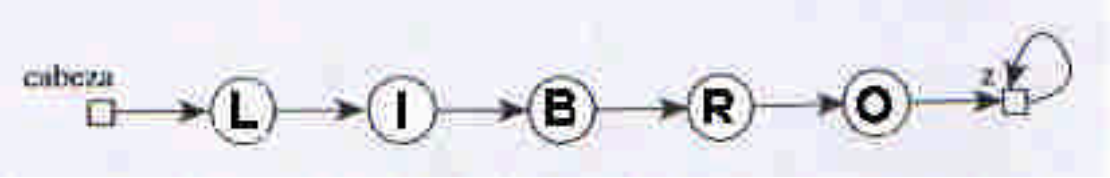
La salida es del programa en pantalla es:

```
50
100
150
200
```

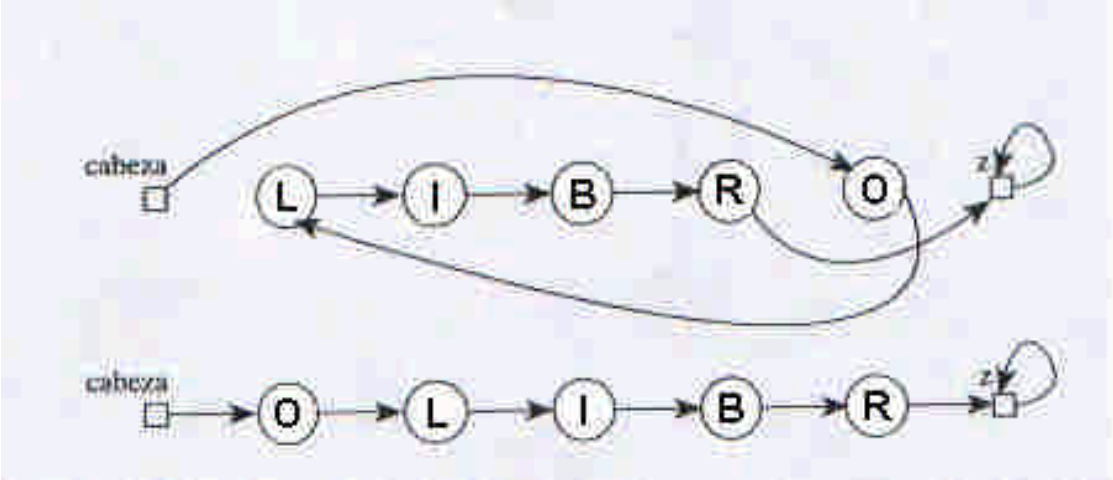
Del programa “prueba”, podemos comentar lo siguiente: se declara el tipo de apuntador a enteros y se definen las variables A y B como apuntadores; el procedimiento estándar NEW asigna la primera posición libre de la memoria al apuntador A y la segunda, al apuntador B; en la dirección de memoria 1, denotada por A^, se almacena el valor 50 y en B^, el 100; se escribe el contenido de las direcciones de memoria 1 y 2, y se imprimen en pantalla los valores de 50 y 100; se asigna la tercera posición libre de la memoria al puntero A y se almacena en éste el valor 150 (con esto, se pierde la asociación de la dirección de memoria 1 con el puntero A, perdiendo por completo el valor contenido en esta dirección –50–); a la variable referenciada B^ se le asigna el valor 200, que reemplaza el contenido de la segunda dirección de memoria –100–; finalmente, se escriben en pantalla las direcciones de memoria 3 y 2, lo que da como resultado 150 y 200, respectivamente.

6.2 Listas ligadas

Una lista ligada es un conjunto de elementos llamados nodos, que están organizados en forma secuencial (semejante a la estructura de los arreglos). En la lista, cada nodo está compuesto por dos elementos: un valor de una variable dinámica y una dirección de memoria que apunta o enlaza al nodo siguiente. Así, todo nodo de la lista ligada debe tener un enlace; el primero y el último de la lista deben contar con enlaces. Por eso, hay dos nodos ficticios, uno se encuentra al principio de la lista (cabeza), que apunta al primer nodo, y otro (Z), al que se dirigirá el último nodo. El objetivo de los nodos ficticios es que puedan hacerse manipulaciones con los enlaces del primer y último de la lista. Ejemplo:



Estas ligas facilitan una ordenación más eficiente que la proporcionada por los arreglos. Si por ejemplo, deseamos mover la O desde el final hasta el principio de la lista, se cambiarían sólo tres ligas: la liga de la R apuntaría a la Z, la liga de la cabeza a la O y la liga de la O a la L; la reordenación queda así:



Aunque físicamente los contenidos de los nodos permanecen en las localidades de memoria de la lista original ligada, sólo cambian los apuntadores de los nodos a nuevas direcciones de memoria.

El procedimiento anterior da más flexibilidad a las listas sobre los arreglos, ya que de otra manera tendrían que recorrerse todos los elementos, desde el principio del arreglo, para dejar un espacio al nuevo elemento. La desventaja es que resta rapidez de acceso a cualquier nodo de la lista.

Otra ventaja es que el tamaño de la lista puede aumentar o disminuir durante la ejecución del programa. El programador no necesita conocer su tamaño máximo. En aplicaciones prácticas, pueden compartir el mismo espacio varias estructuras de datos, sin considerar su tamaño relativo.

La lista ligada permite insertar un elemento, lo que aumenta su longitud en la unidad. Si se inserta la M entre la I y la B, retomando el ejemplo anterior, sólo se modifican dos ligas: la I a la M, y de ésta a la B. Esto en un arreglo sería imposible.

Además, se puede eliminar un elemento de la lista ligada, lo que disminuiría su longitud. Si se quita la M, la liga de la I apunta al nodo B, saltándose la M. El nodo de la M sigue existiendo –de hecho, tiene liga con el nodo B–, pero la M ya no es representativa en la lista.

Hay operaciones que no son propias de las listas, como encontrar un elemento por su índice. En un arreglo sería muy sencillo, basta con acceder a[x]; pero en la lista hay que moverse, enlace por enlace, hasta dar con el elemento buscado.

Otra operación que no es característica de las listas ligadas consiste en localizar un elemento anterior a uno dado. Si el único dato de la lista del ejemplo es el enlace a B, entonces la forma exclusiva de poder encontrar el enlace a B es comenzar desde la cabeza y recorrer la lista para ubicar el nodo que apunta a R.

6.3 Pilas

Las pilas son estructuras dinámicas de datos que pueden ingresar elementos, acumulándolos de modo que el primero se almacene debajo de toda la pila, y sobre éste los siguientes. Con este sistema, después se pueden obtener los datos en orden inverso de como fueron ingresados: del último al primero. Por eso, las pilas trabajan bajo el concepto FILO First Input/ Last Output), que significa “primero en entrar”/“último en salir”.

Una pila necesita sólo un apuntador para que haga referencia a su tope. Los datos que forman la pila son registros enlazados entre sí, al igual que las listas ligadas.

Ejemplo de codificación de una pila:

```
PROGRAM pilas (INPUT,OUTPUT);

TYPE
      tipo_apuntador=^Node;
Node=RECORD
      dato:CHAR;
      localidad:TIPO_APUNTADOR
END;
VAR
      pila:TIPO_APUNTADOR;
      letra:CHAR;
```

Declaraciones necesarias para una pila

El proceso de adición de un dato a una pila se llama PUSH o “empuja hacia abajo de la pila”; el procedimiento de eliminación, POP o “saltando de la pila”. A continuación, mostramos ambos procedimientos:

```
PROCEDURE push(VAR lista:TIPO_APUNTADOR;letra:CHAR);
{se coloca un dato en el tope de la pila}

VAR prueba:TIPO_APUNTADOR;

BEGIN
      NEW(prueba); {crea una variable nueva}
      prueba^.dato:=letra; {asigna el campo de datos}
      prueba^.localidad:=lista; {apunta la lista actual}
      lista:=prueba {define la lista nueva}
END;
```

```
PROCEDURE pop(VAR pila:TIPO_APUNTADOR;VAR letra:CHAR);
{toma el primer elemento de la lista}
VAR guarda:TIPO_APUNTADOR;

BEGIN
      IF pila<>NIL THEN
            BEGIN
                  guarda:=pila;
                  letra:=pila^.dato;
                  guarda:=pila^.localidad;
                  dispose(guarda);{libera la zona de memoria asociada al puntero
                                    guarda}
                  Error:=false;
            END
      ELSE
            BEGIN
                  Error:=true;
                   WRITELN('Error en la pila.');
            END
END;

BEGIN {programa principal}
      pila:=NIL;
      WRITELN('Teclee los caracteres sobre una línea sobre la pila');
      WHILE NOT EOLN DO {eoln es true cuando el siguiente carácter es un enter}
      BEGIN
            READ(letra);
            PUSH(pila,letra) {push adiciona nodos a la pila}
      END;
      WRITELN;
```

```
    WRITELN('Los caracteres a la inversa son');
    WHILE pila<>NIL DO
    BEGIN
        pop(pila,letra); {pop remueve nodos de la pila}
        WRITE(letra)
    END
END.
```

El programa anterior acepta caracteres ingresados por el usuario y, como salida, los imprime en pantalla, pero en orden inverso de como fueron ingresados. Por ejemplo, si ingresamos AEIOU, la salida es UOIEA.

6.4 Colas

Las colas son estructuras de datos que tienen la restricción de acceder a los elementos por ambos lados; es decir, puede insertarse un elemento al principio de la cola y eliminarse otro al final (por ejemplo, cuando los usuarios de un banco hacen fila; las personas se añaden al final de la cola y salen conforme los cajeros los van atendiendo). Las colas trabajan bajo el concepto FIFO (First Input / First Output), que quiere decir: “lo primero en entrar es lo primero en salir”. Conforme los elementos van llegando se van eliminando.

En una cola se necesitan dos apuntadores para que registren los datos del principio y el final. Las entradas de los apuntadores se parecen a las utilizadas en los enlaces de las listas ligadas: algunos campos almacenan los datos y uno guarda un apuntador que hace la referencia hacia el dato.

Para implementar una cola, necesitamos hacer uso de la siguiente codificación:

```
type
        tipo_apun_cola  :  ^.ent_cola;
        ent_cola  :  record
                data  :  integer;
                next  :  tipo_apun_cola
                end;
var
        apunt_princ,
        apunt_final,
        apunt_ent  :  tipo_apun_cola;
```

{se usan los apuntadores apunt_princ y apunt_final para apuntar los datos del principio y final de la cola; el procedimiento añade apunt_ent a la cola, que definen los apuntadores enunciados}

```
procedure ins_cola(var apunt_princ,
                                apunt_final  :  tipo_apun_cola;
                                value  integer);
```

{inserta un nuevo valor Value al final de la cola.}

```
var
        apunt_ent  :  tipo_apun_cola;

begin {ins_cola}
        new(apunt_entl);
        apunt_ent^..data  :=  value,
        apunt_ent^..next  :=  nil;
        if apunt_final  :=  nil  {cola vacía} then
                begin
                        apunt_final  :=  apunt_ent;
```

```
                apunt_princ  :=  apunt_ent
            end
        else
            begin
                apunt_final^.next  :=  apunt_ent;
                apunt_final  :=  apunt_ent
            end
end; {ins_cola}
```

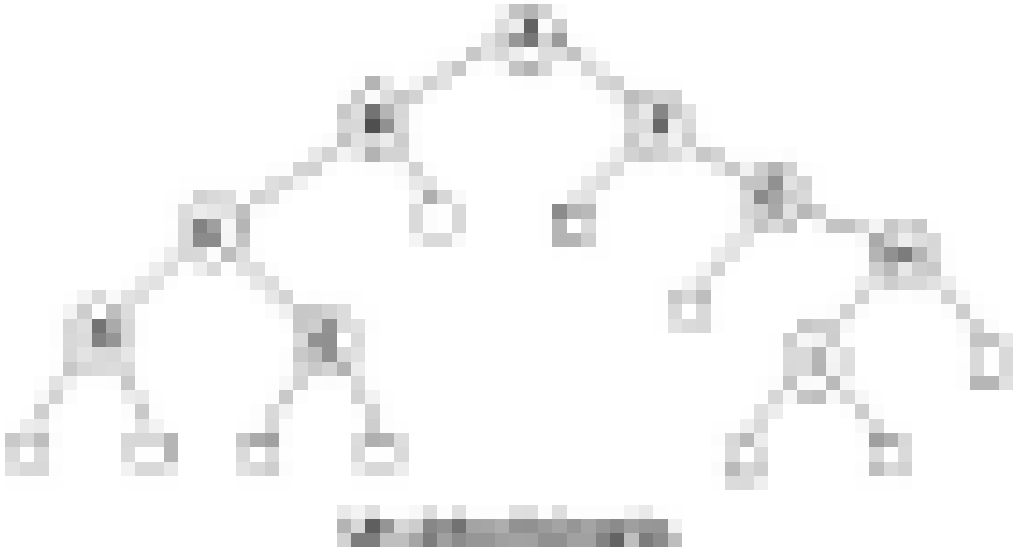
La eliminación de un dato de la cola se lleva a cabo con el siguiente procedimiento:

```
procedure borrar_cola(var apunt_princ,
                          apunt_final  .  cola_ptr;

var
        value  :  integer;

begin
        if apunt_final <> nil then
        begin
            value  :=  apunt_princ^.data;
            apunt_princ  := apunt_princ^.next
        end
end;
```

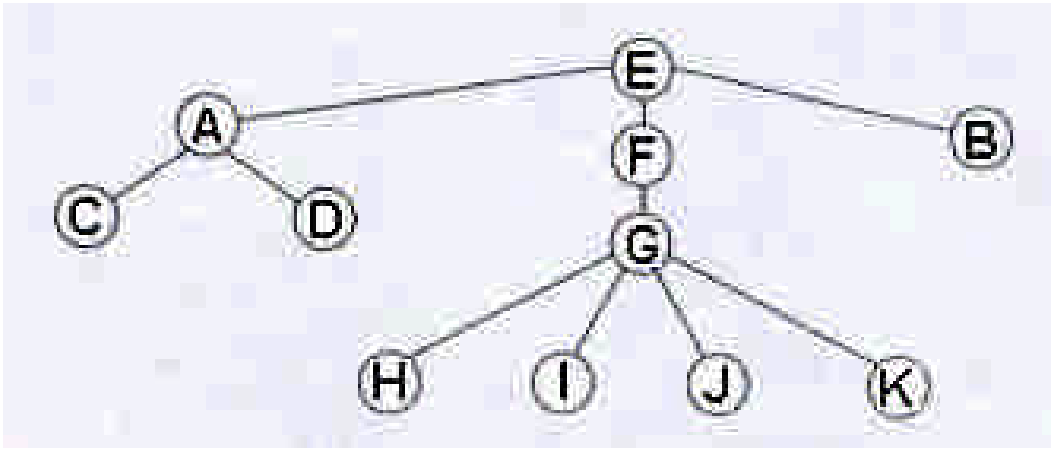
6.5 Árboles binarios

Los árboles binarios son un conjunto de nodos conectados entre sí en forma jerarquizada y cuentan siempre con nodos descendientes Pueden ser de dos tipos:

externos, sin nodos descendientes o nodos hijos; e internos, con un par de nodos hijos organizados siempre en un sentido (izquierdos o derechos). Cualquiera de ellos podría ser nodo externo, es decir, sin hijos. Los nodos externos sirven como referencia para los nodos que carecen de hijos. Veamos la siguiente figura:



Ahora bien, un árbol es un conjunto de vértices y aristas; un vértice, un nodo que tiene nombre e información asociada; una arista, una conexión entre dos vértices; un camino en un árbol, una lista de vértices distintos en la que dos consecutivos se enlazan mediante aristas. Asimismo, a uno de los nodos del árbol se le designa raíz. Lo más característico de un árbol es el camino que existe entre una raíz y un nodo. Ejemplo:



Como observamos en la figura anterior, hay jerarquía en la posición de los nodos. El nodo raíz es el único que no tiene antecesor, pero sí varios nodos sucesores que se van a encontrar debajo de éste. Estos nodos a su vez tendrán descendencia, y así sucesivamente. A los nodos que se encuentran arriba de otros, se les llama nodos padre y a los que dependen de éstos, hijos u hojas. Los nodos que no tienen más sucesores son conocidos como terminales, y los que tienen por lo menos uno, no terminales (pueden constituir un árbol). El conjunto de estos subárboles es un bosque.

La longitud del camino entre un nodo y la raíz está representada por el número de nodos por los que se pasa para ir de aquél a ésta o viceversa. Así, la altura de un árbol es el camino más largo entre la raíz y algún nodo, y la longitud del camino del árbol, la suma de todos los caminos, desde la raíz hasta cada uno de los nodos que lo conforman.

Hay nodos no terminales que tienen un número determinado de hijos predispuestos en un orden (árboles multicamino). Dentro de este tipo, los que poseen la estructura más sencilla son los árboles binarios, que ya fueron definidos al principio del tema.

Para implementar un árbol binario, tenemos que echar mano de la siguiente codificación: _Arbol.pas

```
Módulo con la implementación genérica de los tipos Arbol y NodoArbol.
Debe ser incluido luego de la definición del tipo Dato.
}

type
  Arbol = ^NodoArbol;
  NodoArbol  = record
                 dat      : Dato;
                 izq, der : Arbol
               end;
```

```
procedure Anular (var arb: Arbol);
  begin
    arb := nil
  end;

function Vacio (arb: Arbol): boolean;
  begin
    Vacio := arb = nil
  end;

function Construir (var arb: Arbol; x: Dato; iz, de: Árbol): boolean;
  begin
    new (arb);
    if arb = nil then Construir := true
    else with arb^ do
      begin
        dat := x;
        izq := iz;
        der := de;
        Construir := false
      end
  end;

function HijoIzquierdo (arb: Arbol): Arbol;
  begin
    if Vacio (arb) then HijoIzquierdo := nil
    else HijoIzquierdo := arb^.izq
  end;
```

```
function HijoDerecho (arb: Arbol): Arbol;
 begin
   if Vacio (arb) then HijoDerecho := nil
   else HijoDerecho := arb^.der
 end;

function Raiz (arb: Arbol; var x: Dato): boolean;
 begin
   if Vacio (arb) then Raiz := true
   else
     begin
       x := arb^.dat;
       Raiz := false
     end
 end;
```

Y también:

```
{
 _NodoA.pas

Módulo con la implementación genérica del tipo NodoArbol.
Debe ser incluido luego de la definición del tipo Dato.
}

type
 PNodoArbol = ^NodoArbol;
 NodoArbol  = record
                dat : Dato;
                izq, der : PNodoArbol;
              end;
```

```
function NuevoNodo (var nod: PNodoArbol; x: Dato): boolean;
{
Crea en memoria dinámica un nuevo nodo y se lo asigna a nod.
Si tiene éxito, le asigna x al dato, y nil al izquierdo y al derecho.
}
  begin
    new (nod);
    if nod = nil then NuevoNodo := true
    else with nod^ do
      begin
        dat := x;
        izq := nil;
        der := nil;
        NuevoNodo := false
      end
  end;

procedure EliminarNodo (var nod: PNodoArbol);
{
Eliminar ambos subárboles de nod, y antes de quitar nod dejarlo en nil.
}
  begin
    if nod^.izq <> nil then EliminarNodo (nod^.izq);
    if nod^.der <> nil then EliminarNodo (nod^.der);
    dispose (nod);
    nod := nil
  end;
```

Direcciones electrónicas

http://www.fortunecity.es/imaginapoder/participacion/153/eda.html

http://www.geocities.com/SiliconValley/Park/4768/index.html

http://www.algoritmica.com.ar/cod/tad/main.html

http://www.ei.uvigo.es/~pavon/temario_aedi.html

http://www.upv.es/protel/usr/jotrofer/pascal/punteros.htm

http://www.geocities.com/SiliconValley/Park/4768/listas.html

http://www.iunav.org/~mayr/materia/estrucdatos/repaso04.htm

http://www.geocities.com/SiliconValley/Park/4768/pilas.html

http://www.iunav.org/~mayr/materia/estrucdatos/

http://www.iunav.org/~mayr/materia/estrucdatos/

http://www.algoritmica.com.ar/cod/arbn/main.html

http://agamenon.uniandes.edu.co/~jvillalo/html/cap4.html

Bibliografía de la Unidad

Unidad 7. Archivos

Temario detallado

7. Archivos
   - 7.1 Secuenciales
   - 7.2 De texto
   - 7.3 De entrada-salida

7. Archivos

En lenguaje Pascal, un archivo es una estructura de datos conformada por una colección de registros afines, que está guardada físicamente en un dispositivo de almacenamiento secundario, como cintas, discos flexibles o duros, etcétera, y no en la memoria principal de la computadora. Esto nos posibilita generar archivos de datos que pueden ser usados por varios programas. Así, un programa es capaz de crear un archivo de datos que puede ser utilizado como archivo de entrada para otro programa. Por ejemplo: en una escuela tienen guardados, en un archivo en disco, al inicio del periodo escolar, los datos de todos los alumnos inscritos. El archivo va a ser actualizado en el transcurso de un mes para operar las nuevas altas, cambios de grupo o bajas.

7.1 Secuenciales

Pascal sólo permite el acceso a los datos de un archivo de forma secuencial. Esto quiere decir que para localizar un dato específico dentro del archivo se debe comenzar a leer desde el inicio. Si en nuestro archivo de alumnos necesitamos recuperar el nombre del alumno que está grabado en el registro número 099, Pascal

debe iniciar a leer en serie desde el registro del alumno 001 hasta dar con el que solicita. Recordemos que un registro es una colección de campos.

Veamos la siguiente figura:

La organización y acceso a los registros en este tipo de archivos no presentan dificultad, pero son inconvenientes cuando tenemos archivos muy grandes, en los que el tiempo puede ser determinante en la lectura de registros. En cambio, son recomendables en aplicaciones donde se usan archivos en los que se procesa la mayoría de los registros; por ejemplo, control de alumnos, nóminas, listas de precios, etcétera.

Primero, se declara el tipo de archivo mediante las palabras reservadas FILE OF, posteriormente, con éstas se establecen las variables archivo (deben ser del mismo tipo que las de los registros). Si los registros a almacenar son del tipo entero, la variable o variables archivo deben ser declaradas como de tipo entero. Veamos el siguiente ejemplo:

```
TYPE
        elementos=FILE OF INTEGER;
VAR
        registros:elementos;
```

Es común utilizar estructuras de registros (RECORD) en combinación con archivos:

```
TYPE
        empleado=RECORD
                nombre:string[25];
                edad:integer;
                sueldo:real;
        end;
VAR
        registros:FILE OF EMPLEADO
BEGIN
        ASSIGN(registros,'empleados.dat');
        .................
```

En el ejemplo anterior, se crea como registro el tipo “empleado”, con tres campos de distinto tipo: nombre, edad y sueldo. Luego, la variable archivo “registros” se declara del mismo tipo que “empleado”.

Notemos que se usa el procedimiento ASSIGN. Su función, en el ejemplo, es asociar la variable archivo “registros” con el archivo que físicamente se va a almacenar en el disco o cinta “empleados.dat”. Asimismo, es importante considerar que se debe especificar el trayecto o ruta donde se piensa almacenar el archivo de datos, en el orden siguiente: unidad de disco, ruta o sendero y nombre con su extensión de archivo.

Ejemplo:

```
ASSIGN(registros,’C:\PERSONAL/VENTAS/EMPLEADOS.DAT’);
```

En este caso, el archivo “empleados.dat” se almacenará en la carpeta personal, subcarpeta ventas en la unidad de disco duro C.

Veamos el siguiente programa, en donde se crea un archivo secuencial llamado “alumnos.dat”, cuyos registros contienen los campos no_cta, nombre, sexo, edad, turno, carrera y grupo:

```
PROGRAM arch_secuencial;
TYPE
reg_alumnos=RECORD
        no_cta:STRING[8];
        nombre:STRING[25];
        sexo:CHAR;
        edad:INTEGER;
        turno:CHAR;
       carrera:STRING[3];
       grupo:STRING[4]
   END;
archivo=FILE OF reg_alumnos;
VAR
      base_alumnos:archivo;
      inscripciones:reg_alumnos;
      otro_registro:CHAR;
BEGIN
      ASSIGN(base_alumnos,'alumnos.dat');
      REWRITE(base_alumnos);   {abre el archivo en modo escritura}
      REPEAT
      WITH inscripciones DO
      BEGIN
            WRITE('Numero de cuenta: ');
            READLN(no_cta);
            WRITE('Nombre del alumno: ');
            READLN(nombre);
            WRITE('Sexo: ');
            READLN(sexo);
            WRITE('Edad: ');
            READLN(edad);
```

```
            WRITE('Turno: ');
            READLN(turno);
            WRITE('Carrera: ');
            READLN(carrera);
            WRITE('Grupo: ');
            READLN(grupo)
        END;
WRITE(base_alumnos,inscripciones);
WRITE('Otra inscripcion? (S/N)');
READLN(otro_registro)
UNTIL UPCASE(otro_registro)='N';
CLOSE(base_alumnos)
END.
```

Ya está creado el archivo “alumnos.dat”, que contiene los datos generales de cada alumno. Ahora, consideremos un programa para leer secuencialmente el archivo:

```
PROGRAM leer_arch_sec(INPUT,OUTPUT);
uses
      Crt;
TYPE
reg_alumnos=RECORD
         no_cta:STRING[8];
         nombre:STRING[25];
         sexo:CHAR;
         edad:INTEGER;
         turno:CHAR;
        carrera:STRING[3];
        grupo:STRING[4]
         END;
```

```
archivo=FILE OF reg_alumnos;
VAR
      base_alumnos:archivo;
      inscripciones:reg_alumnos;
BEGIN
      clrscr;
      ASSIGN(base_alumnos,'alumnos.dat');
      RESET(base_alumnos);   {abre el archivo en modo de lectura}
      WHILE NOT EOF(base_alumnos) do
      BEGIN
            READ(base_alumnos,inscripciones);
            WITH inscripciones DO
            BEGIN
                  WRITELN('Numero de cuenta: ',no_cta);
                  WRITELN('Nombre del alumno: ',nombre);
                  WRITELN('Sexo: ',sexo);
                  WRITELN('Edad: ',edad);
                  WRITELN('Turno: ',turno);
                  WRITELN('Carrera: ',carrera);
                  WRITELN('Grupo: ',grupo);
                  WRITELN
              END
      END;
READLN;
CLOSE(base_alumnos)
END.
```

Frecuentemente, hay que actualizar los archivos de datos con nueva información. Lo anterior se logra creando un archivo en el que se grabarán los registros modificados, con lo que se eliminan o editan los registros que sufren cambios. Debe seguirse este proceso:

- Abrir en modo lectura el archivo original que se va a modificar.
- Abrir en modo escritura el archivo en el que se van a salvar los cambios.
- Copiar al nuevo archivo los registros originales que no sufran modificación.
- Leer los registros originales a cambiar, modificarlos y guardarlos en el nuevo archivo.
- Borrar el archivo original, y renombrar el nuevo con el nombre que tenía aquél.

Se debe ocupar un segundo archivo para poder actualizar uno secuencial, pues el proceso no se hace en forma directa. La desventaja es que consume tiempo y cubre más espacio en disco, al menos cuando se está actualizando el archivo.

El acceso de registros secuenciales tiene el inconveniente de tratar uno por uno los registros en el orden como están almacenados. Sin embargo, Pascal puede manipular archivos con registros de igual formato y tamaño, para acceder a un cierto registro, por medio de un número de tipo LONGINT, que contiene una posición lógica del registro buscado. Cuando se abre un archivo, ese número (puntero del archivo) se inicializa con 0 para referenciar al primer registro, y cuando se escriben o leen los registros, va incrementándose de 1 en 1 para poder identificar a cada registro.

Para abrir un archivo indistintamente en modo escritura o lectura, respectivamente, la apertura de los archivos con acceso directo al registro se lleva a cabo con los procedimientos REWRITE o RESET. Estos archivos de acceso directo siempre están dispuestos para realizar operaciones de escritura y lectura.

Abierto el archivo, se usa la función FILESIZE para saber cuántos registros están almacenados. Sintaxis: Num_Reg = FILESIZE(variable_archivo).

Si el archivo no contiene registros, la función devuelve 0, pero si los tiene, regresará el número de éstos.

Una función usada para conocer la posición del registro actual es FILEPOS. Sintaxis: Reg_corriente = FILEPOS(variable_archivo).

Para que podamos ir a un registro específico, de modo que el puntero del archivo se situé en el registro buscado, debemos usar el procedimiento SEEK. Sintaxis: SEEK(variable_archivo,numero_reg). El inconveniente es que el usuario debe conocer el número de posición que guarda el registro en el archivo.

Ejemplos:

SEEK(var_arch,0); {va al primer registro}

SEEK(var_arch,5); {brinca al sexto registro}

SEEK(var_arch,FILESIZE(var_arch)-1); {va al último registro}

Una vez situado en el registro, podemos realizar operaciones de escritura, con WRITE, o de lectura, con READ. Hecha alguna de estas operaciones, el puntero se mueve al siguiente registro. Y ya efectuadas las operaciones necesarias sobre el archivo, se debe cerrar con CLOSE(var_arch).

7.2 De texto

Los archivos de texto están formados por una serie líneas de cadenas de caracteres con distinta longitud, las cuales se limitan con un retorno de carro, también conocido como Enter, que indica el final de la línea (la longitud máxima de las líneas es de hasta 127 caracteres). Este tipo se designa con la palabra reservada TEXT y no necesita ser declarado en la sección de tipos.

En estos archivos pueden escribirse letras, números y símbolos de puntuación. Los caracteres son representados por su correspondiente código ASCII y tiene su representación decimal. Así, una letra “A” (equivalente decimal 65) es diferente de “a” (equivalente decimal 97).

Sus cadenas de caracteres son líneas que se pueden editar o borrar. El fin de las líneas queda determinado por EOLN (End Of Line) y el término de los archivos texto por EOF (End of File) o fin de archivo. Así, el contenido de un archivo de este tipo se asemejaría al siguiente fragmento:

Pascal es lenguaje de programación de computadoras de alto nivel EOLN
con propósito general. Fue desarrollado entre 1967 y 1971 por el EOLN
Profesor Niklaus Wirth, en la Universidad Técnica de Zurich, Suiza.EOF

Sintaxis de las funciones EOF y EOLN:

- EOF(variable archivo);
  acepta el nombre de la variable de archivo, y da como resultado “true” (verdadero) si no encuentra más datos para leer o reconoce la marca de fin de archivo CTRL-Z (eof, end of file).
- EOLN(variable_archivo);
  acepta el nombre de la variable de archivo, y da como resultado “true” si no hay más caracteres para leer en la línea actual.

  En la declaración de archivos de texto, debe definirse primero una variable como archivo de texto, usando la siguiente sintaxis:

  ```
  VAR
      variable:TEXT;
  ```

  Luego, con la instrucción ASSIGN, se asocia esta variable con algún archivo físico en un disco. Formato: ASSIGN(variable,nombre_del_archivo).

  La variable de tipo archivo es la que identifica al archivo externo dentro del programa. El nombre_del_archivo debe contener el nombre con el que el sistema operativo de la computadora identifica al archivo en el medio de almacenamiento secundario donde se vaya a guardar. Además, podemos usar sólo el nombre de archivo, o bien, acompañarlo con una ruta si es que deseamos que se almacene en una carpeta distinta a la actual.

Ejemplos:

‘archivo.txt’,

o también una ruta como ‘\carpeta\archivo.txt’

o inclusive la unidad de disco ‘c:\carpteta\archivo.txt’.

Después, se genera un nuevo archivo o abre uno existente en el modo escritura, con el procedimiento REWRITE. Sintaxis: REWRITE(variable);

Si el archivo de texto ya existe, la instrucción REWRITE destruirá los datos y los reemplazará con los nuevos que se ingresen al archivo. Luego, se escriben datos

en el archivo de texto usando las palabras reservadas WRITE y WRITELN, que insertan las variables dentro del archivo. Sintaxis:

```
WRITE(variable_archivo,var1,var2,var3.....var n);
WRITELN(variable_archivo,var1,var2,var3.....var n);
```

Se usan igual que cuando se escriben las variables en la pantalla de la computadora. La única diferencia es que ahora hay una variable archivo, que envía la salida a un archivo de texto. La instrucción WRITELN inserta, además, la marca de fin de archivo después de la ultima variable.

Una vez que se termina de escribir un archivo, se debe cerrar con la instrucción CLOSE. Sintaxis: CLOSE(variable);

Ya que contamos con las instrucciones necesarias para crear un archivo de texto, pasemos a nuestro siguiente programa:

```
PROGRAM archtexto(INPUT,OUTPUT);
VAR
    Var_Arch:TEXT;
BEGIN
    ASSIGN(Var_Arch,'ARCHIVO.txt');
    REWRITE(Var_Arch);
    WRITELN(Var_Arch,'Este archivo de texto');
    WRITELN(Var_Arch,'fue abierto en modo escritura');
    WRITELN(Var_Arch,'para poder ingresar estas líneas.');
    CLOSE(Var_Arch) {instrucción de cerrar el archivo}
END.
```

La salida del programa es un archivo con el nombre “archivo.txt”, que contiene las siguientes líneas:

Este archivo de texto
fue abierto en modo escritura
para poder ingresar estas líneas.

Una vez creado un archivo, es necesario abrirlo para poder tener acceso a los datos. Para esto, usaremos el procedimiento RESET, que tiene la función de abrir un archivo existente en el modo de lectura, pero no se pueden modificar sus datos. Sintaxis: RESET(variable);

Si no está en disco el archivo al que hicimos referencia con la instrucción anterior, entonces se produce un error de compilación.

Una vez abierto el archivo de texto, usamos las proposiciones READ y READLN. Sintaxis:

READ(variable_archivo,var1,var2,var3.....var n);
READLN(variable_archivo,var1,var2,var3.....var n);

Ambas instrucciones realizan la lectura de datos contenidos en el archivo de texto. Pero ReadLn salta al inicio de la siguiente línea del archivo cuando se han asignado valores a la lista de variables var1, var2, ..var n del procedimiento; en caso contrario, el procedimiento continúa asignando información sin tomar en cuenta el final de la línea (EOLN). Una vez leído todo el archivo, debe cerrarse con la instrucción CLOSE.

Ejemplo:

```
PROGRAM archtexto(INPUT,OUTPUT);
VAR
    Var_Arch:TEXT;
    letra:STRING[40];
BEGIN
```

```
    ASSIGN(Var_Arch,'ARCHIVO.txt');
    RESET(Var_Arch);
    READLN(Var_Arch,letra);
    WRITELN(letra);
    READLN(Var_Arch,letra);
    WRITELN(letra);
    READLN(Var_Arch,letra);
    WRITELN(letra);
    CLOSE(Var_Arch) {instrucción de cerrar el archivo}
END.
```

El programa abre el “archivo.txt” en modo lectura y escribe en pantalla su contenido:

```
Este archivo de texto
fue abierto en modo escritura
para poder ingresar estas líneas.
```

En resumen, debemos crear primero un archivo de texto en modo escritura, con la instrucción REWRITE, y posteriormente abrirlo en modo lectura, con RESET. Así, podemos acceder a los datos del archivo.

7.3 De entrada-salida

Un archivo de entrada-salida puede ejecutar dos operaciones: leer desde un archivo guardado en un medio de almacenamiento secundario y escribir datos en él.

Hay dos archivos de texto que, por valor predeterminado, son de entrada-salida. Tienen correspondencia con las entradas de datos del usuario, a través del teclado (INPUT), y con las salidas, por el monitor de la computadora o la impresora (OUTPUT).

Pascal trata estos procedimientos (entrada-salida) como si fueran archivos que leen y escriben datos por medio de los dispositivos periféricos ya mencionados. Razón por la que deben usarse, en el encabezado de todo programa en Pascal, así: PROGRAM nombre_programa(INPUT,OUTPUT);

Con lo anterior, los procedimientos READ y WRITE leen y escriben los dispositivos estándar (teclado y monitor).

A veces, el ingreso de datos se puede llevar a cabo desde un archivo de texto u otro dispositivo periférico diferente del teclado; también la salida de datos del programa, además de visualizarse en el monitor, puede almacenarse en un archivo externo, o salir en una copia permanente a través de la impresora.

El archivo es, pues, una colección de datos que pueden estar organizados en líneas de texto, como números o registros. Esta colección produce información que ponemos a disposición bajo un nombre de archivo.

Pascal usa dos nombres de archivo por omisión: INPUT y OUTPUT, asociados al teclado y a la pantalla, respectivamente. Si los datos tienen diferente origen que el estándar, entonces se debe especificar en la cabecera del programa, así:

```
PROGRAM  prueba_salida(INPUT,arch_datos);
```

Aquí, se está cambiando la salida estándar del monitor a un archivo llamado arch_datos.

La variable arch_datos corresponde a un archivo que va a contener una serie de caracteres, para lo cual, Pascal tiene la palabra reservada TEXT. Sintaxis:

```
VAR   arch_datos : TEXT;
```

Ejemplo:

```
PROGRAM  texto(INPUT,OUTPUT,archdato);
        VAR
             archdato:TEXT;
            ch:CHAR;
            nomarch:PACKED ARRAY[1..15] OF CHAR;
        BEGIN
            WRITELN('Ingrese el nombre del archivo de texto: ');
             READLN(nomarch);
             ASSIGN(archdato,nomarch);
            REWRITE(archdato);
            READLN;
            READ(ch);
            WHILE  ch<>'*' DO
            BEGIN
              WRITE(archdato,ch);
              READ(ch)
            END;
            WRITE(archdato,'*');
            CLOSE(archdato)
        END.
```

El programa “texto”, aparte de la entrada y salida estándar, asigna una salida extra hacia el archivo “archdato” en la cabecera del programa, declarado como de texto.

En la ejecución, se solicita al usuario el ingreso del nombre de un archivo para almacenarse dentro de la variable “nomarch”, ésta se asocia (ASSIGN) con la variable “archdato”. Se lee del teclado y almacena en la variable “ch”; si su contenido es diferente del asterisco (*), entonces se graba dentro del archivo de texto y se vuelve a leer del teclado el contenido de la variable “ch”. El ciclo continúa hasta que el usuario teclee un asterisco, al hacerlo, se guarda al final del archivo y se cierra (CLOSE).

Un ejemplo de salida puede ser la siguiente lista:

lenguaje
de
programación
Pascal
*

Hay que aclarar que la salida de este programa se lleva a cabo tanto en pantalla (OUTOUT) como en archivo de texto (“archdato”).

Por último, hay que tomar en cuenta algunas reglas para los archivos de entrada-salida:

- Los archivos estándar INPUT y OUTPUT ya están definidos; no deben redefinirse.
- Los archivos de entrada y salida no deben declararse como tipo de archivo (FILE OF tipo).
- El procedimiento REWRITE no se debe de utilizar con la salida estándar OUTPUT.
- El procedimiento RESET no se debe de emplear con la entrada estándar INPUT.

NOTA: los procedimientos WRITE, WRITELN, READ Y READLN se utilizan como ya explicamos en temas anteriores.

Direcciones electrónicas

http://cub.upr.clu.edu/~jsola/3012L1.htm
http://mx9.xoom.com/tutoriales/unidad8.htm
http://www.itlp.edu.mx/publica/tutoriales/pascal/u8_8_1.html

http://www.itlp.edu.mx/publica/tutoriales/pascal/u8_8_7.html
http://www.telesup.edu.pe/biblioteca_vitual/textos_manuales/formacion_tecnologica/lprog/pascal/pascal.htm
http://www.geocities.com/SiliconValley/Horizon/5444/pas059.htm
http://www.itlp.edu.mx/publica/tutoriales/pascal/u8_8_6.html
http://www-gris.ait.uvigo.es/~belen/isi/doc/tutorial/unidad8p.html
http://www.geocities.com/SiliconValley/Horizon/5444/pas056.htm

Bibliografía de la Unidad

Unidad 8. Compilación separada y diseño de programas

Temario detallado

8. Compilación separada y diseño de programas
   - 8.1 Diseño descendente y ascendente de programas
   - 8.2 Encapsulamiento de módulos
   - 8.3 Uso de módulos en otros programas
   - 8.4 Las secciones de interfase e implementación
   - 8.5 Uso de los comandos make y build de Turbo Pascal

8. Compilación separada y diseño de programas

El código fuente de un programa muy largo con muchos módulos (procedimientos y funciones) suele ser difícil de manejar para el programador que lo hizo y mucho más para el personal ajeno al programa. Afortunadamente, Pascal soluciona este problema con la característica del lenguaje para agrupar tipos de datos, constantes, procedimientos y funciones en unidades que se compilan de manera separada.

Compilación es el proceso de traducir un código fuente, escrito bajo la sintaxis de un lenguaje de programación a un código objeto que contiene código máquina –binario-, el único que puede reconocer los dispositivos digitales de las computadoras. Estas rutinas independientes deben poseer en su encabezado el objetivo de la unidad, las relaciones con otras unidades o con sus detalles más relevantes.

Además, el diseño de programas se basa en el análisis de los datos de entrada y reportes de salida, sus archivos, procedimientos y funciones para el nuevo programa. Para diseñarlo, ha de recopilarse toda la información del personal involucrado con el nuevo sistema. Los programadores deben familiarizarse con el programa con el que van a trabajar y los usuarios participar activamente con el equipo de diseño (los destinatarios finales son una fuente muy valiosa de información).

El diseño comienza con la definición de salidas del programa (son los resultados de éste y deben satisfacer las necesidades de los usuarios). Posteriormente, se procede a diseñar los formatos de entrada y los archivos que va a usar el programa. Las salidas del programa permiten saber al programador los datos necesarios de entrada para producir tal información. Los archivos, a su vez, deben tener la información necesaria para almacenar los datos del sistema (por esto debe diseñarse adecuadamente la estructura de los campos y registros de archivos). Realizado lo anterior, se desarrollan las especificaciones del programa, éstas incluyen los nombres de los archivos, los registros y sus campos, así como su tamaño, la definición de tipos de datos, constantes, variables, etiquetas, procedimientos y funciones, control de flujo del programa, expresiones, abreviaturas y cualquier nota aclaratoria para los programadores.

Se lleva a cabo el desarrollo del programa, haciendo evaluaciones parciales para, en caso necesario, realizar los cambios pertinentes de modo que se ajuste el programa a las especificaciones de diseño. Una vez completado, se efectúa una prueba con datos ficticios. Corridos los ajustes al programa, debe capacitarse al personal encargado de operar el programa.

Si el programa pasa satisfactoriamente la prueba inicial, se le cargan los datos reales y se lleva a cabo un periodo de transición. En éste, el nuevo programa trabaja paralelamente con el sistema actual con el fin de medir su rendimiento. Los resultados que arroje nuestro programa deben ser comparados con el del sistema instalado. En esta fase pueden llegarse a detectar problemas que no se percibieron en la ejecución individual de módulos, por lo que es necesario refinar el programa.

Una vez puesto a punto el programa, se procede a su implantación, sustituyendo, paulatina parcial o totalmente, el sistema anterior.

El proyecto de diseño debe concluir con la recopilación de toda la documentación del nuevo programa, que debe de detallar cada elemento del programa y sus aspectos operativos.

Asimismo, el programador debe documentar el programa durante su desarrollo, con el fin de que personas ajenas a su código fuente tengan una guía y puedan conocer las partes de éste y el funcionamiento de cada módulo.

Pocos programadores tienen el hábito de la documentación; se limitan a desarrollar e implementar los programas. Es importante crear conciencia que todo programa debe estar debidamente diseñado y documentado con manuales, gráficas, diagramas de flujo, glosario de identificadores usados, etcétera. Esto facilita su consulta y depuración.

De manera general, los pasos que conforman la documentación de programas son:

- Problema a resolver. Definición del objetivo del programa.
- Planteamiento del problema. Se prueba a detalle el método de solución en la forma más abstracta posible. Se puede hacer uso de herramientas como el seudocódigo o un diagrama de flujo.
- Definición de módulos e interfaces. Se describen las diferentes partes que conforman un programa, estableciendo las especificaciones de las relaciones entre los módulos del programa principal. De preferencia, las relaciones deben ilustrarse con gráficos.
- Análisis de la implementación. Se hace la descripción de cada uno de los identificadores usados en el programa en un documento que va a servir para futuras consultas.
- Descripción de archivos usados. Se definen los campos empleados, así como las estructuras e interrelaciones de los archivos.
- Descripción de entradas y salidas. Análisis de los datos de entrada al sistema y las salidas en reportes e informes. En este apartado, el sistema se toma como una caja negra sin importar cómo se lleva a cabo el procedimiento de modificación de datos (lo que interesa es saber si el sistema está produciendo buenos informes).
- Requerimientos especiales. Se estipulan las restricciones que da el programa al problema y los alcances del mismo.
- Manual de uso del programa. Son los instructivos para los operadores del sistema. En ellos se indica cómo se instala el programa en la computadora, los archivos utilizados y las relaciones de éstos; la descripción de los menús, la forma de configurar el entorno de trabajo, la descripción de los

campos solicitados por el programa…, todo el detalle necesario para manejar el programa.

- Descripción de datos de prueba. Agrupa al conjunto de datos usados para probar el correcto funcionamiento del programa. Normalmente, se usan datos ficticios para ingresar al sistema, con los cuales se ven los posibles errores en la impresión de reportes, con el fin de depurar los programas.
- Glosario e índices. Lista de todos los nombres de los identificadores usados en el sistema, como etiquetas, constantes, variables, etcétera, y una breve descripción de su uso.

Los documentos anteriores deben encuadernarse y tener una buena presentación; no se debe abusar de tecnicismos, su redacción ha de usar lenguaje llano y conciso de modo que pueda ser entendido por cualquier persona. Debe estar ampliamente ilustrado y organizado bajo un buen sistema de numeración de apartados e incluir un índice para facilitar su consulta.

8.1 Diseño descendente y ascendente de programas

Los programas tienen procedimientos en varios niveles, organizados jerárquicamente. Con base en esta jerarquía, hay dos tipos de diseños de programas: descendente y ascendente.

Descendente

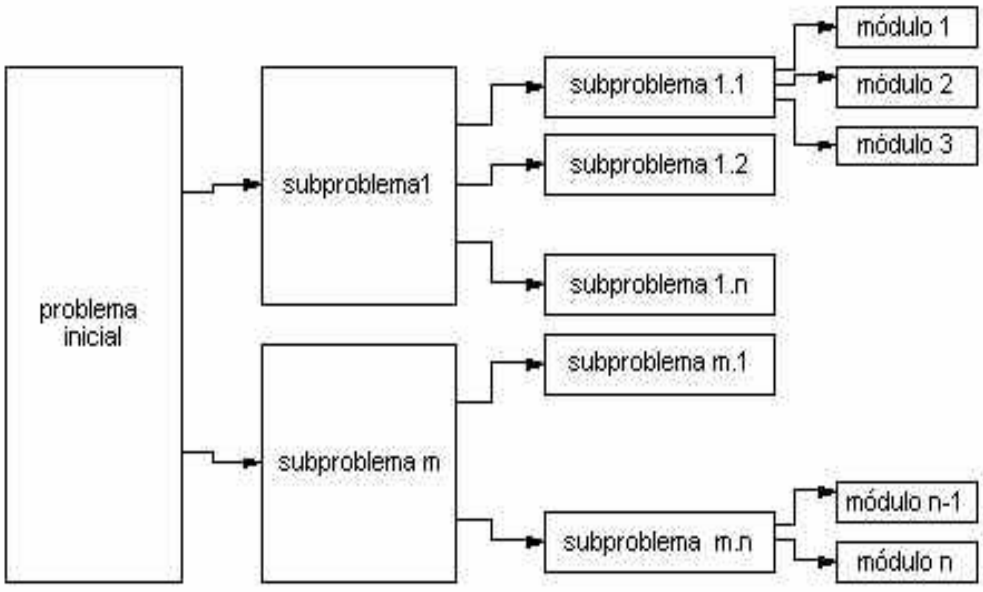
Es aquel donde se plantea el programa desde la parte más alta de la jerarquía hasta los procedimientos más inferiores. Los procedimientos superiores dan una solución general al problema, solución que se va afinando por niveles hasta llegar a los procedimientos más inferiores. Los procedimientos de alto nivel son usados para expresar la abstracción de la solución del problema, éstos, a su vez, llaman a procedimientos de niveles inferiores que contendrán los detalles de la ejecución.

Los niveles jerárquicos están relacionados por medio de las entradas y salidas de datos e información. Así, una salida de un procedimiento es a la vez entrada de datos para el procedimiento inmediato posterior. En cada nivel, se va dando un refinamiento progresivo del conjunto de instrucciones.

Este tipo de diseño es recomendado por la mayoría de programadores, debido a las siguientes ventajas:

- El problema se va descomponiendo en cada nivel, por lo que su complejidad va de más a menos.
- Es factible la división del trabajo para desarrollar los procedimientos.
- Uniformidad y consistencia entre los procedimientos.
- Todos los procedimientos tienen la logística del programa global, que es el diseño de lo general a lo particular.
- El programa final es modular, lo que lo hace más sencillo su mantenimiento.

Lo anterior no garantiza que los programas estén exentos de errores; a veces será necesario deshacer ciertos procedimientos de niveles inferiores y volver a diseñarlos desde la parte más alta de la jerarquía. Sin embargo, es buen método para construir, en la mayoría de los casos, programas funcionales. Veamos la siguiente figura:



El refinamiento de los procedimientos se logra gracias a tres técnicas:

- Divide y vencerás. Un programa grande debe dividirse en subprogramas independientes.
- Ciclos finitos. Son una serie de instrucciones que se ejecutan sucesivamente para ir avanzando hacia la solución del problema. Las estructuras que se usan son FOR, WHILE y REPEAT.
- Análisis de casos. Dividir el problema por grupos afines plenamente identificados. Se usa la estructura CASE, y si son pocos casos, entonces se opta por IF o IF compuesta.

Para ejemplificar las técnicas de refinamiento, consideremos el siguiente problema: hacer un algoritmo que acepte las altas, cambios y bajas de los alumnos de una escuela.

Si procedemos a dividir el problema principal, encontramos que podemos hacerlo en tres partes:

```
leer datos del alumno;
procesar los datos;
imprimir comprobante;
```

Los tres procedimientos son válidos para un alumno; para un número indeterminado, es el siguiente:

```
REPEAT
        leer datos del alumno;
        procesar los datos;
        imprimir comprobante
UNTIL no haya más datos que leer
```

Sin embargo, con lo anterior no sabemos qué alumnos acaban de inscribirse, cuáles se dieron de baja o simplemente actualizaron su registro. Entonces, necesitamos refinar los grupos de procedimientos:

```
REPEAT
        leer datos del alumno;
        CASE grupo OF
                altas:
                procesar inscripciones;
                cambios:
                procesar cambios;
                bajas:
                procesar bajas
        END;
        procesar datos;
        imprimir comprobante
UNTIL no haya más datos que leer
```

Ahora pasemos a ver un ejemplo del diseño descendente:

```
PROGRAM descendente(INPUT,OUTPUT);
VAR
num1,num2,may:INTEGER;

FUNCTION mayor(val1,val2:INTEGER):INTEGER;
BEGIN
    IF val1>val2 THEN
        mayor:=val1
    ELSE
        mayor:=val2
END;
```

```
PROCEDURE tabla(val:INTEGER);
VAR
      i,res:INTEGER;
BEGIN
      FOR I:= 1 TO 10 DO
        BEGIN
            res:=val*I;
            WRITELN(val,' X ',i,' = ',res)
      END
END;

BEGIN
      WRITE('Captura el primer numero: ');
      READLN(num1);
      WRITE('Captura el segundo numero: ');
      READLN(num2);
      may:=mayor(num1,num2);
      tabla(may);
      READLN
END.
```

En este ejemplo, el programa principal solicita al usuario que ingrese dos números enteros; éstos se envían a la función llamada mayor, que devuelve como resultado el número más grande de entre ambos. El programa principal pasa el valor del resultado anterior al procedimiento llamado tabla, que despliega en la pantalla de la computadora una tabla de multiplicar con ese valor, del 1 hasta el 10. Se retorna el control al programa principal y termina.

Ascendente

El proyecto de construcción de programas comienza desde sus cimientos, es decir, desde los procedimientos de los niveles inferiores que tienen a su cargo la ejecución del programa. Es posible que el analista de sistemas diseñe cada módulo por separado conforme vayan apareciendo, para luego unir los procedimientos en niveles superiores hasta llegar al módulo de control de todo el programa.

El analista puede asignar cada módulo, para su codificación, prueba y depuración, a programadores diferentes. Cada uno deberá tratar su módulo por separado, ingresándole datos de prueba y corriéndolo (así, pueden obtener resultados hasta asegurarse que cada módulo se ejecuta sin errores). Una vez probados los módulos, los programadores más experimentados realizarán el ensamble de todos en los procedimientos de control de los niveles más altos de la jerarquía.

En la práctica, muchos programadores hacen uso de esta técnica, comienzan por los pies del programa hasta llegar a la cabeza, ensamblando todas las partes del cuerpo al final y así poder probar todo el programa en forma general.

Por lo siguiente, el diseño ascendente no es el más recomendado:

- Dificulta la integración de los procedimientos en detrimento de su desempeño global.
- Al integrar los procedimientos, la depuración de errores en sus interfaces (relación entre las partes) es costosa y sólo puede llevarse a cabo al final del proyecto.
- No se prevén las interfaces en el diseño de procedimientos.
- Aunque en forma individual trabajen bien, los procedimientos tienen muchas limitaciones en forma global.
- Se duplican esfuerzos y recursos al introducir una gran cantidad de datos.
- Falta de uniformidad y consistencia entre el diseño de los procedimientos.
- Los objetivos globales del proyecto no se consideran en un diseño ascendente, razón por la cual no se cumplen.

En la construcción de programas, independientemente de que se utilice el diseño descendente o ascendente, intervienen las siguientes fases: diseño, implementación y prueba. De esta manera, siempre que tengamos un problema que pueda ser resuelto empleando la computadora, debemos establecer el objetivo del problema; posteriormente, diseñar el algoritmo (establecer la secuencia detallada de pasos para solucionar el problema); luego, codificarlo en las instrucciones de un lenguaje de programación para construir lo mejor posible nuestro programa de computadora; y por último, una vez obtenido el programa, lo depuramos en caso de que contenga errores hasta afinarlo completamente.

Resulta útil para la construcción de programas emplear las funciones estándar y las bibliotecas de programas. Las primeras contienen tareas preprogramadas que vienen ya con el lenguaje de programación y no es necesaria su recodificación; se ejecutan insertando en el programa una línea de código que posea el nombre de la función; las segundas comprenden una colección de unidades (construidas por programadores) que son usadas para realizar ciertas tareas, y están dispuestas para ser utilizados por cualquier persona interesada en implementarlas en sus programas.

8.2 Encapsulamiento de módulos

El encapsulamiento de módulos consiste en agrupar, conforme con su estructura, el código fuente de un programa grande en unidades más pequeñas y manejables. La modularización de programas presenta la ventaja de trabajar los procedimientos y las interfaces que hay entre éstos, de modo que cada módulo sea parte de un proceso de refinamiento de la solución al problema.

El programa puede descomponerse identificando sus tareas específicas que puedan tratarse como módulos individuales. Así, se logra una especificación más detallada de cada módulo y a la vez se simplifica su manejo.

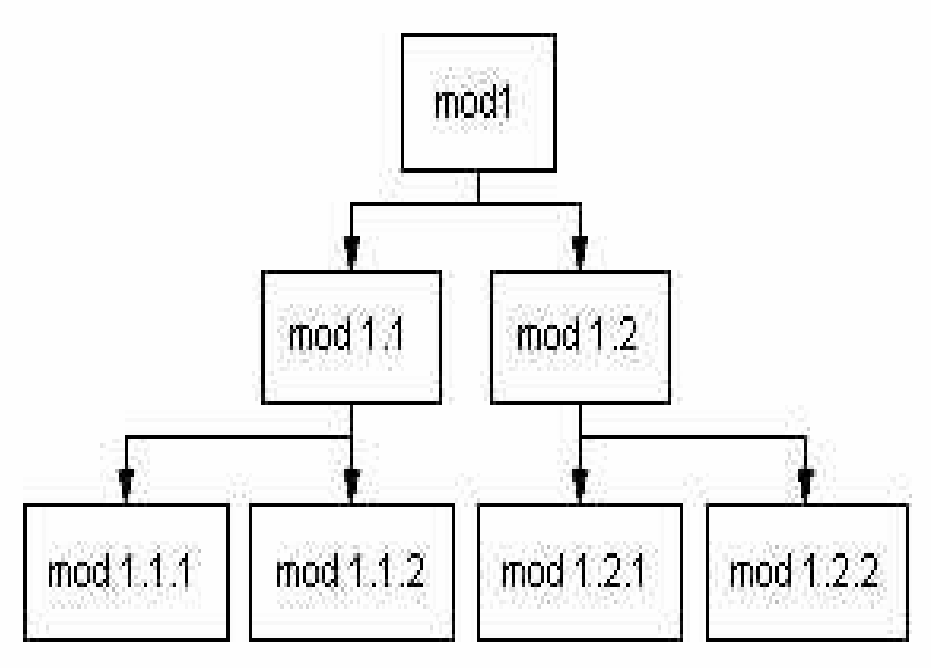
Descomposición vertical del problema:

Deben definirse el objetivo, las entradas de datos y las salidas de información para cada módulo. La información que genere un módulo servirá como ingreso de datos al siguiente.

Los módulos de niveles inferiores tendrán especificaciones más detalladas de la tarea del módulo.

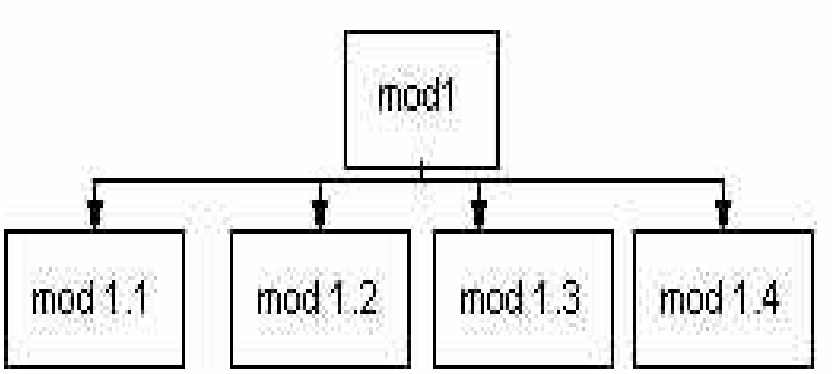
En forma general, podemos apreciar la jerarquía de entrada, proceso y salida.

En la siguiente figura tenemos una descomposición vertical del problema:



Descomposición horizontal del problema.

El programa se divide en procedimientos o funciones. Posteriormente, el programa principal o un módulo hace la llamada al procedimiento A, que a la vez llama al B y éste al C, y así sucesivamente hasta que el último procedimiento retorna el control al procedimiento inicial.



Ahora bien, fraccionar un programa grande para simplificarlo implica considerar una serie de conceptos y reglas sobre la estructura modular. Es lo que revisamos brevemente a continuación.

El grado en que las instrucciones de los módulos ejecutan una tarea o función específica se le conoce como cohesión. Esto implica que no deben fragmentarse los procesos fundamentales de los módulos, y no hay que juntar procesos que no guarden relaciones afines dentro de los mismos.

Cualquier cambio en la programación de un módulo no debe incidir en los otros. Se conservará siempre la misma forma de entrada y salida de datos del módulo con respecto de los otros.

El grado en que las interfaces o relaciones entre los módulos son más fuertes se le llama acoplamiento. Entre mayor acoplamiento tengan los módulos será más difícil realizar cambios en el sistema, lo que puede causar problemas en su implantación y dificultar futuras actualizaciones o correcciones al mismo.

Se recomienda la mínima relación de datos entre los módulos para que no dependan tanto uno de otro y los cambios en un módulo no se propaguen al resto del sistema. Se deben usar acoplamientos con listas y datos sencillos y evitar en lo posible que se compartan estructuras complejas entre módulos.

El tamaño del módulo debe ser suficiente para ejecutar una tarea específica. Lo ideal es que la codificación no rebase en promedio una cuartilla.

El alcance de control de un módulo indica la cantidad de módulos que llama. Se recomienda que no exceda de seis módulos inmediatos inferiores; si rebasa esta cifra, conviene insertar módulos de control intermedios (de no hacerlo, la complejidad de la lógica de su código aumenta en detrimento de su comprensión).

El efecto del control de un módulo que toma alguna decisión debe incidir en otro subordinado, directa o indirectamente. Es decir, aunque no lo afecte en un nivel inmediato, debe estar comprendido en la misma rama en algún nivel inferior que dependa de ese módulo. De no ser así, se establecerían relaciones inadecuadas generando más decisiones con mayor acoplamiento.

Deben economizarse los recursos, realizando sólo las tareas que son estrictamente necesarias, a esto se le conoce como parsimonia.

Finalmente, los módulos deben tener la capacidad de administrar sus errores internos (detección y corrección), para evitar, en lo posible, que las fallas se transmitan a otros módulos aumentando el acoplamiento.

8.3 Uso de módulos en otros programas

Habitualmente, se emplean muchos programas y procedimientos que pueden ser útiles para otros programas, razón por la que es necesario guardarlos como unidades independientes y poder reutilizarlos posteriormente. Este método es más eficiente que rescribir, depurar y compilar el procedimiento.

Los módulos se declaran como unidades, las cuales son en sí un código fuente precompilado (traducidos al lenguaje máquina) y están independientes del programa principal (se crean y guardan en archivos de disco, que pueden contener tipos de datos, constantes, variables, funciones y procedimientos e inclusive llamadas a otras unidades).

Las unidades precompiladas ofrecen las ventajas siguientes:

- Ahorro en el tiempo de compilación del programa principal debido al fácil acceso al código objeto de las unidades.

- Pueden aislarse las unidades en librerías, en lugar de estar distribuidas en varios programas.
- Generan programas nuevos más concisos y sencillos al no contener las declaraciones de identificadores de las unidades.
- Al abocarse a procedimientos nuevos, la depuración de programas es más fácil, ya que los procedimientos de las unidades ya están probados.
- Puede hacerse la división de trabajo de un proyecto grande entre un equipo de programadores.
- Las unidades no se vuelven a compilar debido a modificaciones en el programa que las llama.
- El código de una unidad puede modificarse sin necesidad de recompilar los módulos que usa.
- Si los procedimientos y funciones son módulos que se consideran herramientas, las unidades son una poderosa caja de herramientas.

Gracias a las ventajas anteriores, el desarrollo y depuración de nuevos programas se agiliza y simplifica: contamos con procedimientos y funciones precompilados en unidades que nos facilitan esa tarea.

Las unidades no pueden ejecutarse en forma independiente, están sujetas al programa que las llama para poder realizar su tarea o función. En sí no tienen cuerpo de programa; opcionalmente, se les puede incluir una serie de instrucciones encerradas entre BEGIN y END.

Después del encabezado, los programas (sentencia PROGRAM) cuentan con una sección de unidades para declarar su uso, empleando la cláusula USES seguida del nombre de todas las unidades –separadas por comas– que se utilizarán en el programa. Sintaxis: USES mi_modulo, lista, ordena;

mi_modulo, lista y ordena son unidades que se van a enlazar al programa principal. En éstas se encuentran contenidos los procedimientos y funciones que se van a usar. El compilador busca el código en las unidades declaradas (por lo que

verifica en el disco la existencia de los archivos que almacenan a las unidades). En caso de no encontrarlas, el compilador se detendrá notificando el error al usuario.

Declaradas las unidades pueden ejecutarse sus instrucciones en cualquier parte del programa como si formaran parte del mismo código fuente del programa principal.

Pascal incorpora unidades predefinidas que contienen gran cantidad de rutinas integradas, entre las que destacan las siguientes.

CRT	Rutinas referentes al teclado y al monitor.
PRINTER	Manejo de la impresora.
DOS	Manejo de funciones y archivos bajo el sistema operativo de disco (DOS).
GRAPH	Gráficos.
SYSTEM	Acceso a funciones del sistema operativo.
OVERLAY	Rutinas de transferencia de memoria entre las partes de un programa.

Las unidades están compuestas de cuatro partes: declaración, inicialización, interfase e implementación. De las dos últimas hablaremos mas adelante, pasemos a definir las dos primeras:

Declaración. Es la parte en la que se da nombre a la unidad. Este nombre es indispensable, pues identifica a la unidad con el programa llamador. La nominación de la unidad debe ser corresponder con el objetivo de la unidad. Sintaxis: UNIT nombre_de_la_unidad;

Inicialización. En esta parte, se definen los valores de las variables que se van a emplear en la unidad. Su uso es opcional y su sintaxis se encuentra entre las palabras reservadas BEGIN y END.

8.4 Las secciones de interfase implementación

Imaginemos que compramos una computadora. Ésta viene empacada en varias cajas que contienen respectivamente el monitor, teclado, CPU, bocinas... Ahora, supongamos que las cajas sólo vienen identificadas con una etiqueta con el número de parte asignado por el proveedor para distinguirla. Bien, si nuestro pedido nos lo entregan a domicilio, nos deben proporcionar una factura que precisa tanto el número de parte como la descripción del producto. De este modo, sin necesidad de abrir las cajas, sabremos lo que contienen con base en su número de referencia.

Apliquemos el ejemplo a nuestra materia. Las unidades, como ya vimos, son módulos independientes que pueden considerarse como cajas cerradas. Posiblemente no conozcamos su contenido, pero si nos describen lo que hacen, sabremos para qué podemos utilizarlas.

Las unidades tienen dos secciones: interfase e implementación.

Interfase. Es como la lista de productos referenciados en la factura de compra de la computadora. Es una descripción general de los componentes de la unidad: tipos de datos, constantes, variables, encabezados de procedimientos y funciones a los que tiene acceso cualquier programa que llame a la unidad (procedimientos, funciones u otras unidades). Una instrucción de la interfase es: PROCEDURE tarea(var1:integer);

No vamos a incluir en esta sección todo el código del procedimiento “tarea”, sólo su encabezado, en el que están contenidos el nombre del procedimiento, parámetros que usa y el tipo de datos.

El código fuente con el que se ejecuta el procedimiento lo tendremos en la sección de implementación.

Para declarar una interfaz: hacemos uso de la palabra reservada INTERFACE, que va inmediatamente después de UNIT. Veamos cómo hacerlo:

```
UNIT prueba;
INTERFACE
USES CRT, DOS;
```

```
TYPE
tipo = dec_tipo,
VAR
        variable : tipo;

PROCEDURE proced(parámetro : tipo);
FUNCTION func(a,b . INTEGER) : INTEGER;
```

La palabra reservada USES es opcional. En el caso anterior, se indica el uso de las unidades predefinidas CRT y DOS. Se pueden definir tipos de datos personales y luego usarlos para declarar variables. Éstos van a ser visibles para cualquier programa que use la unidad “prueba”; asimismo “proced”, y la función “func” (sólo debe incluirse su encabezado).

Implementación. Puede contener etiquetas, constantes y variables que sólo van a ser usadas dentro de la unidad. Éstas son necesarias para la ejecución de las funciones y procedimientos, que tiene su código fuente completo en esta sección y que ya habían sido previamente referenciados en la sección de interface. Los identificadores privados de las unidades, aunque no estén visibles a programas ajenos, tienen cierta influencia indirecta sobre éstos, por medio de los procedimientos y funciones de esta sección.

La declaración de la implementación se hace a través de la palabra IMPLEMENTATION. Se usa de la forma siguiente:

```
IMPLEMENTATION
VAR
var_priv : tipo;

PROCEDURE proced(parámetro : tipo);
VAR
y,z:INTEGER;
BEGIN
```

```
serie de instrucciones;
END;

FUNCTION func(a,b . INTEGER) : INTEGER;
BEGIN
serie de instrucciones;
END;
```

En esta sección se incluyen el código del procedimiento “proced” y la función “func”, que van a ejecutar los procesos definidos en la sección de la interfase. Observamos que se usa una variable llamada var_priv, que será ‘invisible’ a los demás programas: su empleo se restringe a la unidad ‘prueba’.

Veamos a continuación un caso completo de una unidad con sus dos secciones y el programa que va a hacer uso de ésta.

El archivo que almacena la unidad se llama “FACTOR.PP” (en Turbo Pascal, la extensión de los archivos de unidades es TPU) y está compilado en forma independiente.

```
UNIT factor;
INTERFACE
USES CRT;
FUNCTION factorial(n:INTEGER):INTEGER;

IMPLEMENTATION

FUNCTION factorial(n:INTEGER):INTEGER;
BEGIN
    IF n>1 THEN
        factorial := n * factorial(n-1)
    ELSE
        factorial := 1
END;
END.
```

El archivo que contiene el programa se llama “factorial.pas” y también está compilado aparte:

```
PROGRAM recursivo;
USES CRT,factor;
VAR
    x:INTEGER;
BEGIN
    CLRSCR;
    WRITE('Introduce el numero para calcular el factorial: ');
    READLN(x);
    WRITELN('El factorial del numero es: ',factorial(x));
    WRITELN('Presione <ENTER> para finalizar el programa');
    READLN
END.
```

El programa principal se llama “recursivo” y usa dos unidades, la estándar CRT y la personal “factor”. Declara a la variable “x” como entera y luego solicita al usuario el ingreso de su valor. Éste se envía a la función factorial(x), la cual forma parte de la unidad llamada “factor”, que tiene una sección de interfase donde se declaran la función factorial y otra sección de implementación, que contiene el código de la misma y sirve para obtener el factorial de un número por medio de la recursividad de la función. La función factorial devuelve el factorial de “x”, y el resultado se exhibe en pantalla. Así termina el programa.

La unidad que contiene la función factorial puede ser usada por otros programas. Esto se obtiene incluyéndoles la cláusula USES en su código para abrir la unidad, y añadiéndoles el nombre de los procedimientos y funciones que forman parte de la unidad a emplear.

Como ejemplo de unidades precompiladas tenemos las rutinas READLN y WRITELN, que estamos acostumbrados a usar dentro de nuestros programas.

Ambas rutinas son unidades que cuentan con secciones de interfase e implementación y cuyo código fuente se encuentra oculto al usuario. Aunque desconozcamos su codificación, sabemos cómo emplearlas.

8.5 Uso de los comandos make y build de Turbo Pascal

Turbo Pascal maneja un entorno de desarrollo integrado (IDE) de menús mucho más familiar al usuario que el de Pascal. En Turbo es posible editar, compilar, ejecutar y depurar nuestros programas: están a nuestra disposición todos los comandos afines, agrupados en sus menús correspondientes. En el menú FILE encontramos las operaciones que se pueden llevar a cabo con los archivos; y en el menú EDIT, mediante el uso de un pequeño procesador de textos, desarrollaremos el código fuente de nuestros programas o unidades.

Para compilar un programa o una unidad, preparándola para su uso en otros programas, se usa la opción COMPILE (del menú del mismo nombre), que contiene, entre otras, dos alternativas importantes:

MAKE. Comando que se usa para compilar un programa, puede ser distinto al programa actual que reside en la memoria de la computadora. Esto se logra gracias a que el comando permite que se especifique el nombre del programa a compilar.

BUILD. Trabaja en forma similar que MAKE: recompila el programa fuente disponible de los archivos que guardan las unidades, que a su vez forman parte del proyecto del programa actual.

Los comandos anteriores producen un módulo objeto que es intermedio entre el código fuente (escrito por el programador) y el módulo ejecutable (.EXE) o el de unidad (TPU), que también se generan en este proceso.

El módulo ejecutable es independiente del lenguaje que lo originó. Por esto, se puede grabar para su ejecución en otra computadora que tenga el mismo sistema operativo.

Debe verificarse dentro del mismo menú que la opción DESTINATION dirija su almacenamiento al disco (flexible 3 ½" o duro), ya que es necesario guardar el módulo compilado para poder usarlo en otras sesiones. De lo contrario, se enviará la compilación a la memoria de la computadora; y una vez que se salga del entorno de desarrollo se perderá el contenido y habrá que recompilar nuevamente la unidad para poder usarla.

Direcciones electrónicas

http://www.cs.clemson.edu/html_docs/SUNWspro/pascal/user_guide/pascalug_sepcomp.doc.html

http://www.unizar.es/eupt/asignaturas/programacion/Tema13/index.htm

http://www.inf.udec.cl/~mvaras/estprog/cap3.html

http://atenea.ipvg.cl/~rgarrido/metodologia/cap2-0.html

http://www.inf.udec.cl/~mvaras/estprog/cap3.html

http://atenea.ipvg.cl/~rgarrido/metodologia/cap2-1.html

http://www-gris.ait.uvigo.es/~belen/isi/doc/tutorial/unidad7p.html

http://www.cs.clemson.edu/html_docs/SUNWspro/pascal/user_guide/pascalug_progconstruct.doc.html#161

http://www.di-mare.com/adolfo/p/convpas.htm

http://199.111.112.137/labs/lab26001.html

http://www3.htl-hl.ac.at/doc/packages/fpk/ref/node57.html

Bibliografía de la Unidad

Bibliografía

Apéndice. Elaboración de un mapa conceptual

Los alumnos del Sistema de Universidad Abierta (SUA), a diferencia de los del escolarizado, estudian por su cuenta las asignaturas del plan de estudios correspondiente. Para asimilar el contenido de éstas, requieren consultar y estudiar la bibliografía específica que se les sugiere en cada unidad, actividad nada sencilla, pero indispensable para que los alumnos puedan desarrollar las actividades de aprendizaje y prepararse para los exámenes. Por tanto, un recurso educativo del que pueden valerse los estudiantes es el mapa conceptual.

¿Qué es un mapa conceptual?

- ✓ Es un **resumen o apunte gráfico**.
- ✓ Es un esquema gráfico en **forma de árbol, que muestra la relación existente entre los aspectos esenciales estudiados**, relativos a una unidad de una asignatura o de una asignatura completa, o bien, de un capítulo de un libro o un libro completo.
- ✓ Es una **estructura jerárquica en cuya parte superior** se ubica el aspecto de **mayor nivel de implicación o "término conceptual"**, de éste se derivan otros de **menor grado de implicación** que se relacionan de manera subordinada, por lo que se localizan en niveles inferiores y así sucesivamente en orden descendente, como se observa en el ejemplo de mapa conceptual de *Introducción a la teoría general de la Administración*.

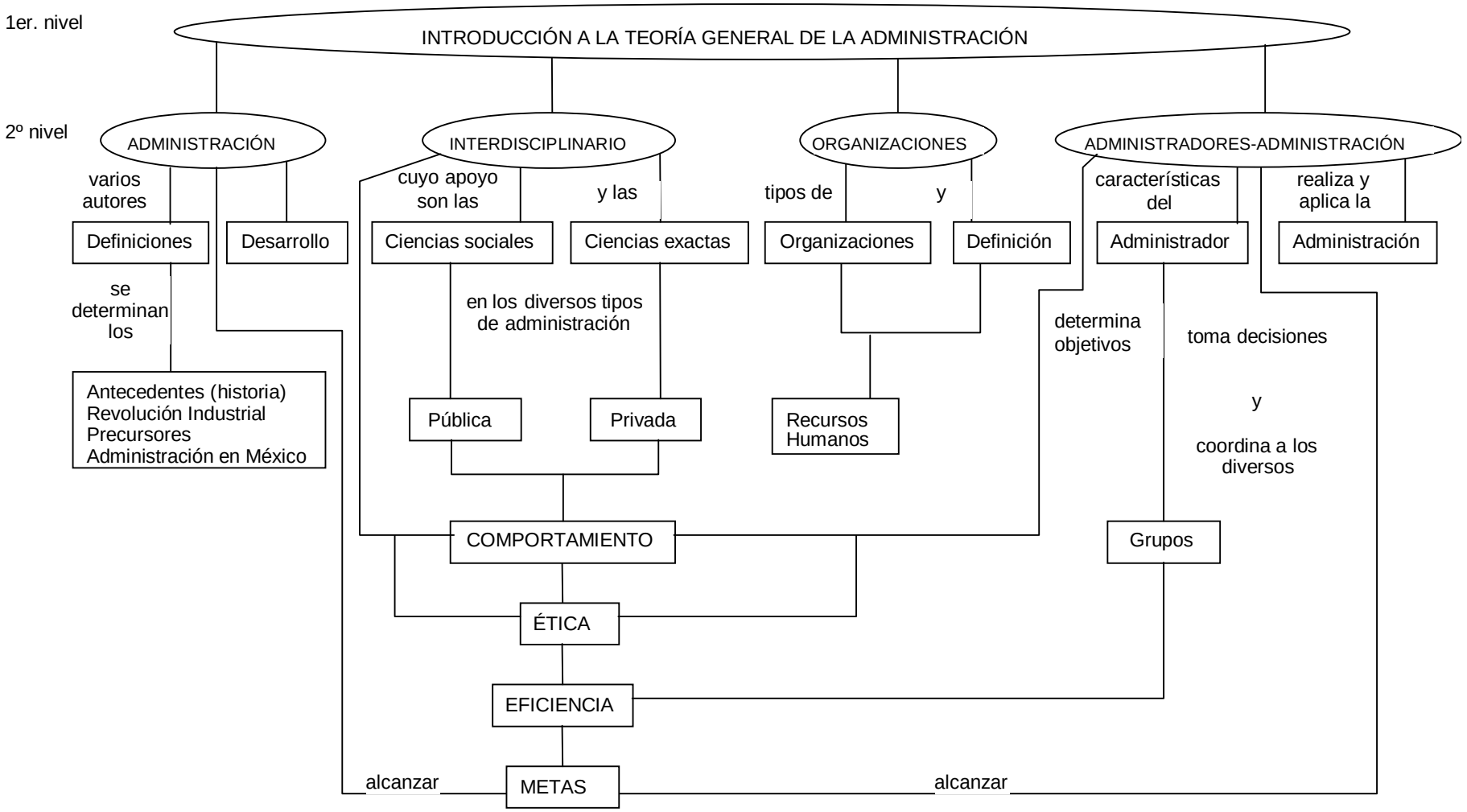
¿Qué ventajas tiene para el alumno un mapa conceptual?

- ✓ Cuando el alumno estudia nuevos contenidos, la construcción de un mapa conceptual le permite **reflexionarlos, comprenderlos y relacionarlos**, es decir, **reorganiza y reconstruye** la información de acuerdo con su propia lógica de entendimiento.
- ✓ Al encontrar las conexiones existentes entre los aspectos esenciales o "términos conceptuales" (clave) del contenido estudiado, el alumno aprenderá a **identificar la información significativa** y a dejar de lado la que no es relevante.
- ✓ El alumno aprende a identificar las ideas principales que el autor de un libro de texto expone, argumenta o analiza; así como a jerarquizarlas y relacionarlas con otros conocimientos que ya posee.
- ✓ La elaboración de un mapa conceptual ayuda a los estudiantes a reproducir con mucha aproximación el contenido estudiado.
- ✓ La construcción de un mapa conceptual estimula en el alumno el **razonamiento deductivo**.

¿Cómo se elabora o construye un mapa conceptual?

1. Realice una primera lectura del capítulo del libro que se le indica en la bibliografía específica sugerida. Preste atención a la introducción y a las notas que el autor hace acerca de los temas y subtemas, porque le ayudarán a comprender la estructura del capítulo; además revise los esquemas, las tablas, las gráficas o cualquier ilustración que se presente. Esta lectura le permitirá tener **una idea general** del contenido del capítulo.
2. Realice una **lectura analítica** del contenido del capítulo, léalo por partes guiándose por la división que el propio autor hace de los temas y subtemas, que por lo general, es más o menos extensa según el tema de que se trate y su complejidad.

3. Lea las ideas contenidas en los párrafos, **analícelos completamente**, ya que en ellos el autor define, explica y argumenta los aspectos esenciales del capítulo; también describe sus propiedades o características, sus causas y efectos, da ejemplos y, si se requiere, demuestra su aplicación.
4. Al analizar las ideas contenidas en los párrafos, **identifique los "términos conceptuales" o aspectos esenciales** acerca de los cuales el autor proporciona información específica.
5. Elabore un **listado de los principales "términos conceptuales"**. Identifique el papel que juega cada uno de ellos y **ordénelos de los más generales e inclusivos a los más específicos o menos inclusivos**.
6. Liste para cada "término conceptual" lo que el autor aborda: definición, propiedades o características, causas y efectos, ejemplos, aplicaciones, etcétera.
7. Coloque los "términos conceptuales" con los aspectos que en ellos se señalan, **en forma de árbol**. **Encierre** en un círculo o rectángulo cada término. Coloque el de mayor inclusión **en el nivel superior** y el resto, **ordénelo de mayor a menor inclusión**. Verifique que la jerarquización sea correcta.
8. Relacione los "términos conceptuales" **mediante líneas** y, si es necesario, **use flechas que indiquen la dirección** de las relaciones. Verifique que las relaciones horizontales y verticales sean correctas, así como las relaciones cruzadas (aquellas que se dan entre "términos conceptuales" ubicados opuestamente, pero que se relacionan entre sí).
9. Construya **frases breves o palabras de enlace** que establezcan o hagan evidente las relaciones entre los "términos conceptuales".
10. Analice el ejemplo del mapa conceptual de *Introducción a la teoría general de la Administración*. Identifique los niveles, "los términos conceptuales", los aspectos que de ellos se derivan, las relaciones horizontales, verticales y cruzadas.



Ejemplo de mapa conceptual de Introducción a la teoría general de la Administración (Profra. Rebeca Novoa)

NOTAS

Tutorial para la asignatura de Introducción a la programación es una edición de la Facultad de Contaduría y Administración. Se terminó de imprimir en mayo de 2003. **Tiraje:** 150 ejemplares. **Responsable:** L. A. y Mtra. Gabriela Montero Montiel, Jefa de la División de Universidad Abierta. **Edición a cargo de:** L. A. Francisco Hernández Mendoza y L. C. Aline Gómez Angel. **Revisión a cargo de:** Lic. María del Carmen Márquez González y L. C. Nizaguié Chacón Albarrán.

Dr. Juan Ramón de la Fuente
Rector

Lic. Enrique del Val Blanco
Secretario General

Mtro. Daniel Barrera Pérez
Secretario Administrativo

Lic. Alberto Pérez Blas
Secretario de Servicios a la Comunidad

Dra. Arcelia Quintana Adriano
Abogada General

Dr. José Narro Robles
Coordinador General de Reforma Universitaria

●

C.P.C. y Maestro Arturo Díaz Alonso
Director

L.A.E. Félix Patiño Gómez
Secretario General

Dr. Ignacio Mercado Gasca
Jefe de la División de Estudios de Posgrado

C.P. Eduardo Herrerías Aristi
Jefe de la División de Contaduría

L.A. y Maestro Adrián Méndez Salvatorio
Jefe de la División de Administración

Ing. y Mtra. Graciela Bribiesca Correa
Jefa de la División de Informática

L.A. y Maestro Jorge Ríos Szalay
Jefe de la División de Investigación

L.Ps. y Mtro. Fco. Javier Valdez Alejandre
Jefe de la División de Educación Continua

L.A. y Mtra. Gabriela Montero Montiel
Jefa de la División de Universidad Abierta

L.C. José Lino Rodríguez Sánchez
Secretario de Intercambio Académico

L.A. Carmen Nolasco Gutiérrez
Secretaria de Planeación Académica

L.A. Rosa Martha Barona Peña
Secretaria de Personal Docente

L.A. Gustavo Almaguer Pérez
Secretario de Divulgación y Fomento Editorial

L.A. Hilario Corona Uscanga
Secretario de Relaciones

L.C. Adriana Padilla Morales
Secretaria Administrativa

L.A. María Elena García Hernández
Secretaria de Planeación y Control de Gestión

L.E. José Silvestre Méndez Morales
Subjefe de la División de Estudios de Posgrado

Dr. Sergio Javier Jasso Villazul
Coordinador del Programa de Posgrado en Ciencias de la Administración

L.A., L.C. y Mtro. Rafael Rodríguez Castelán
Subjefe de la División de Estudios Profesionales

L.C. y Mtro. Juan Alberto Adam Siade
Subjefe de la División de Investigación

L.A. y Maestro Eric Manuel Rivera Rivera
Jefe de la División Juriquilla

C.P. Rafael Silva Ramírez
Asesor de la Dirección

L.A. Balfred Santaella Hinojosa
Jefe de Administración Escolar