

Capítulo 1. Concepto de Algoritmo.

1.1 Objetivo Educativo

El Alumno:

- Conocerá los conceptos de lenguaje y algoritmo computacional.
- Podrá diferenciar los niveles de lenguajes que utiliza un computador.
- Conocerá las principales características de un algoritmo.

1.2 Definición de Lenguaje

Lenguaje: Es una serie de símbolos que sirven para transmitir uno o más mensajes (ideas) entre dos entidades diferentes. A la transmisión de mensajes se le conoce comúnmente como **comunicación**.

La **comunicación** es un proceso complejo que requiere una serie de reglas simples, pero indispensables para poderse llevar a cabo. Las dos principales son las siguientes:

1. Los mensajes deben correr en un sentido a la vez.
2. Debe forzosamente existir 4 elementos: Emisor, Receptor, Medio de Comunicación y Mensaje.

1.3 Lenguajes de Programación

Un Lenguaje de Programación: Es un conjunto de símbolos, caracteres y reglas (programas) que le permiten a las personas comunicarse con la computadora.

Los lenguajes de programación tienen un conjunto de instrucciones que nos permiten realizar operaciones de entrada/salida, cálculo, manipulación de textos, lógica/comparación y almacenamiento/recuperación.

Los lenguajes de programación se clasifican en:

Lenguaje Máquina: Son aquellos cuyas instrucciones son directamente entendibles por la computadora y no necesitan traducción posterior para que la CPU pueda comprender y ejecutar el programa. Las instrucciones en lenguaje máquina se expresan en términos de la unidad de memoria más pequeña el bit (dígito binario 0 o 1).

Lenguaje de Bajo Nivel (Ensamblador): En este lenguaje las instrucciones se escriben en códigos alfabéticos conocidos como mnemotécnicos para las operaciones y direcciones simbólicas.

Lenguaje de Alto Nivel: Los lenguajes de programación de alto nivel (BASIC, Pascal, Cobol, Fortran, etc.) son aquellos en los que las instrucciones o sentencias a la computadora son escritas con palabras similares a los lenguajes humanos (en general en inglés), lo que facilita la escritura y comprensión del programa.

1.4 Definición de Algoritmo

La palabra algoritmo se deriva de la traducción al latín de la palabra árabe alkhwarizmi, nombre de un matemático y astrónomo árabe que escribió un tratado sobre manipulación de números y ecuaciones en el siglo IX.

Definición 1: Un algoritmo es una serie de pasos organizados que describe el proceso que se debe seguir, para dar solución a un problema específico.

Definición 2: Un algoritmo se puede definir como una secuencia finita de instrucciones cada una de las cuales tiene un significado claro y puede ser efectuada con una cantidad finita de esfuerzo en una longitud de tiempo también finito.

1.4.1 Características de los Algoritmos

Las características más relevantes de los algoritmos son:

- **Finito:** Un algoritmo debe siempre terminar después de un número finito de pasos.
- **Definido:** Cada paso de un algoritmo debe ser definido en forma precisa, estableciendo las acciones que van a efectuarse clara y rigurosamente en cada caso.
- **Entradas:** El algoritmo tiene cero o más entradas, es decir cantidades que se entregan inicialmente al algoritmo antes de su ejecución.
- **Salidas:** Un algoritmo tiene una o más salidas, es decir cantidades que tienen una relación específica respecto a las entradas.
- **Efectivo:** Generalmente, también se espera que un algoritmo sea efectivo. Esto significa que todas las operaciones que se realizan en el algoritmo deben ser lo suficientemente básicas de modo que puedan en principio ser llevadas a cabo en forma exacta y en un período de tiempo finito por una persona usando lápiz y papel (rutear).

En la práctica, para evaluar un buen algoritmo se considera el tiempo que requiere su ejecución, esto puede ser expresado en términos del número de veces que se ejecuta cada paso. Otros criterios de evaluación pueden ser la **adaptabilidad** del algoritmo al computador, su **simplicidad** y elegancia, etc. Algunas veces se tienen varios algoritmos para solucionar el mismo problema, y se debe decidir cuál es el mejor. Esto último conduce al "Análisis de Algoritmos". Dado un algoritmo es determinar sus características de desempeño.

1.4.2 Lenguajes Algorítmicos

Es una serie de símbolos y reglas que se utilizan para describir de manera explícita un proceso, estos lenguajes algorítmicos pueden ser:

- (a) **Gráficos:** Es la representación gráfica de las operaciones que realiza un algoritmo (diagrama de flujo).
- (b) **No Gráficos:** Representa en forma descriptiva las operaciones que debe realizar un algoritmo (pseudocódigo).

Capítulo 2. Técnicas para la formulación de algoritmos

2.1 Objetivo Educativo:


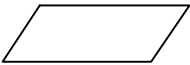

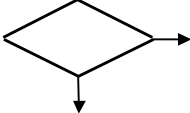





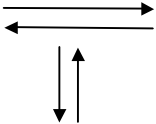
El alumno:

Será capaz de diferenciar los métodos de representación y formulación de algoritmos, así como de conocer las características más importantes de cada técnica. Las dos herramientas utilizadas comúnmente para diseñar algoritmos son:

- Diagrama de Flujo
- Pseudo código

2.2 Diagrama de Flujo

Un diagrama de flujo es la representación gráfica de un algoritmo. También se puede decir que es la representación detallada en forma gráfica de como deben realizarse los pasos en la computadora para producir resultados. Esta representación gráfica se da cuando varios símbolos (que indican diferentes procesos en la computadora), se relacionan entre si mediante líneas que indican el orden en que se deben ejecutar los procesos. Los símbolos utilizados han sido normalizados por el instituto norteamericano de normalización (ANSI).

<u>SÍMBOLO</u>	<u>DESCRIPCIÓN</u>
	Indica el inicio y el final de nuestro diagrama de flujo.
	Indica la entrada y salida de datos.
	Símbolo de proceso y nos indica la asignación de un valor en la memoria y/o la ejecución de una operación aritmética.
	Símbolo de decisión indica la realización de una comparación de valores.
	Se utiliza para representar los subprogramas.
	Conector dentro de pagina. Representa la continuidad del diagrama dentro de la misma pagina.
	Conector fuera de pagina. Representa la continuidad del diagrama en otra pagina.
	Indica la salida de información por impresora.
	Indica la salida de información en la pantalla o monitor.
	Líneas de flujo o dirección. Indican la secuencia en que se realizan las operaciones.

2.3 Recomendaciones para el diseño de Diagramas de Flujo

- Se deben usar solamente líneas de flujo horizontales y/o verticales.
- Se debe evitar el cruce de líneas utilizando los conectores.
- Se deben usar conectores solo cuando sea necesario.
- No deben quedar líneas de flujo sin conectar.
- Se deben trazar los símbolos de manera que se puedan leer de arriba hacia abajo y de izquierda a derecha.
- Todo texto escrito dentro de un símbolo deberá ser escrito claramente, evitando el uso de muchas palabras.

2.4 Pseudo código

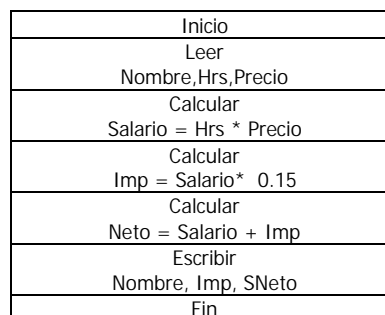
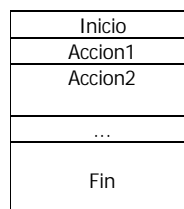
Mezcla de lenguaje de programación y español (o inglés o cualquier otro idioma) que se emplea, dentro de la programación estructurada, para realizar el diseño de un programa. En esencial, el pseudo código se puede definir como un lenguaje de especificaciones de algoritmos. Es la representación narrativa de los pasos que debe seguir un algoritmo para dar solución a un problema determinado. El pseudo código utiliza palabras que indican el proceso a realizar.

2.5 Ventajas de utilizar un Pseudo código a un Diagrama de Flujo

- Ocupa menos espacio en una hoja de papel.
- Permite representar en forma fácil operaciones repetitivas complejas.
- Es muy fácil pasar de pseudo código a un programa en algún lenguaje de programación.
- Si se siguen las reglas se puede observar claramente los niveles que tiene cada operación.

2.6 Diagramas estructurados (Nassi-Schneiderman)

El diagrama estructurado N-S también conocido como diagrama de Chapin es como un diagrama de flujo en el que se omiten las flechas de unión y las cajas son contiguas. Las acciones sucesivas se pueden escribir en cajas sucesivas y como en los diagramas de flujo, se pueden escribir diferentes acciones en una caja. Un algoritmo se representa en la sig. forma:



Capítulo 3. Metodología para Solución de Problemas por Computador.

3.1 Objetivo Educativo

El Alumno:

- (a) Deberá lograr diferenciar las especificaciones de un problema e implementar una o varias soluciones.
- (b) Conocerá las etapas principales que incorpora la creación de un programa o sistema informático.

3.2 Definición del Problema

Esta fase está dada por el enunciado del problema, el cual requiere una definición clara y precisa. Es importante que se conozca lo que se desea que realice la computadora; mientras esto no se conozca o entienda del todo no tiene mucho caso continuar con la siguiente etapa.

3.3 Análisis del Problema

Una vez que se ha comprendido lo que se desea del computador, es necesario definir:

- 3. Los datos de entrada.
- 4. Cual es la información que se desea producir (salida).
- 5. Los métodos y fórmulas que se necesitan para procesar los datos.

Una recomendación muy practica es el que nos pongamos en el lugar del computador y analicemos que es lo que necesitamos que nos ordenen y en que secuencia para producir los resultados esperados.

3.4 Diseño del Algoritmo

Las características de un buen algoritmo son las siguientes:

- (a) Debe poseer un punto particular de inicio.
- (b) Debe ser definido, no debe permitir dobles interpretaciones.
- (c) Debe ser flexible, soportando la mayoría de variantes que se puedan presentar en la definición del problema.
- (d) Debe ser finito en tamaño y tiempo de ejecución.

3.5 Codificación

La codificación es la operación de escribir la solución del problema (de acuerdo a la lógica del diagrama de flujo o pseudo código), en una serie de instrucciones detalladas ó en un código reconocible por la computadora. La serie de instrucciones detalladas se le conoce como código fuente, el cual se escribe en un lenguaje de programación o lenguaje de alto nivel.

3.6 Prueba y Depuración

Los errores humanos dentro de la programación de computadoras son muchos y aumentan considerablemente con la complejidad del problema. El proceso de identificar y eliminar errores, para dar paso a una solución sin errores se le llama depuración.

La depuración o prueba resulta una tarea tan creativa como el mismo desarrollo de la solución, por ello se debe considerar con el mismo interés y entusiasmo.

Resulta conveniente observar los siguientes principios al realizar una depuración, ya que de este trabajo depende el éxito de nuestra solución.

3.7 Documentación

Es la guía o comunicación escrita en sus variadas formas, ya sea en enunciados, procedimientos, dibujos o diagramas.

A menudo un programa escrito por una persona, es usado por otra. Por ello la documentación sirve para ayudar a comprender o usar un programa o para facilitar futuras modificaciones (mantenimiento).

La **documentación** se divide en tres partes:

- Documentación Interna
- Documentación Externa
- Manual del Usuario

Documentación Interna: Son los comentarios o mensaje que se añaden al código fuente para hacer mas claro el entendimiento de un proceso.

Documentación Externa: Se define en un documento escrito los siguientes puntos:

- Descripción del Problema (Enunciado).
- Nombre del Autor (Analista, Programador).
- Algoritmo (Diagrama de flujo o Pseudo código).
- Diccionario de Datos (Descripción de variables).
- Código Fuente (Programa).

Manual del Usuario: Describe paso a paso la manera como funciona el programa, con el fin de que el usuario obtenga el resultado deseado.

3.8 Mantenimiento

Se lleva acabo después de terminado el programa, cuando se detecta que es necesario hacer algún cambio, ajuste o complementación al programa para que siga trabajando de manera correcta. Para poder realizar este trabajo se requiere que el programa este correctamente documentado.

Capítulo 4. Entidades Primitivas para el Desarrollo de Algoritmos.

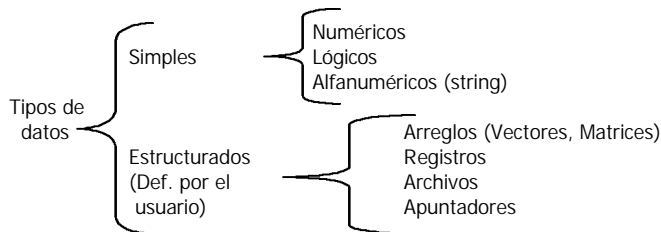
4.1 Objetivo Educativo

El Alumno:

- (a) Deberá lograr diferenciar los tipos de datos existentes, pero principalmente los simples.
Deberá ser capaz de trabajar con fórmulas compuestas por expresiones, operandos y operadores aritméticos, relacionales y lógicos.
- (b) Deberá conocer y trabajar con identificadores, sean estos constantes o variables.

4.2 Tipos de datos

Todos los datos tienen un tipo asociado con ellos. Un dato puede ser un simple carácter, tal como 'b', un valor entero tal como 35. El tipo de dato determina la naturaleza del conjunto de valores que puede tomar una variable.



4.2.1 Tipos de Datos Simples

Datos Numéricos: Permiten representar valores escalares de forma numérica, esto incluye a los números enteros y los reales. Este tipo de datos permiten realizar operaciones aritméticas comunes.

Datos Lógicos: Son aquellos que sólo pueden tener dos valores (verdadero o falso) ya que representan el resultado de una comparación entre otros datos (numéricos o alfanuméricos).

Datos Alfanuméricos (String): Es una secuencia de caracteres alfanuméricos que permiten representar valores identificables de forma descriptiva, esto incluye nombres de personas, direcciones, etc. Es posible representar números como alfanuméricos, pero estos pierden su propiedad matemática, es decir no es posible hacer operaciones con ellos. Este tipo de datos se representan encerrados entre comillas.

Ejemplo:

"Instituto Tecnológico"
"2002"

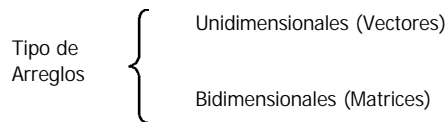
4.2.2 Tipos de Datos Estructurados

Arreglos: son un conjunto finito de valores escalares, ordenados y consecutivos, denominados conjunto de índices del vector, en otro conjunto D de datos variables de un mismo tipo, conjunto base del vector.

I	D
i1	d1
i2	d2
i3	d3
...	...
in	dn

Un *Arreglo* es una estructura de datos que almacena bajo el mismo nombre (variable) a una colección de datos del mismo tipo. Los arreglos se caracterizan por:

- Almacenan los elementos en posiciones contiguas de memoria.
- Tienen un mismo nombre de variable que representa a todos los elementos. Para hacer referencia a esos elementos es necesario utilizar un índice que especifica el lugar que ocupa cada elemento dentro del archivo.



4.2.3 Vectores

Es un arreglo de "N" elementos organizados en una dimensión donde "N" recibe el nombre de longitud o tamaño del vector. Para hacer referencia a un elemento del vector se usa el nombre del mismo, seguido del índice (entre corchetes), el cual indica una posición en particular del vector. Por ejemplo:

Vec[x]

Donde:

Vec Nombre del arreglo
 X Numero de datos que constituyen el arreglo

Representación gráfica de un vector

Vec[1]	7
Vec[2]	8
Vec[3]	9
Vec[4]	10

Llenado de un Vector

```
Hacer para I = 1 a 10
Leer vec[I]
Fin-para

Hacer mientras I <= 10
Leer vec[I]
Fin-mientras

I=1
Repetir
Leer vec[I]
I = I + 1
Hasta-que I>10
```

4.2.4 Matriz

Es un arreglo de M * N elementos organizados en dos dimensiones donde "M" es el numero de filas o renglones y "N" el numero de columnas.

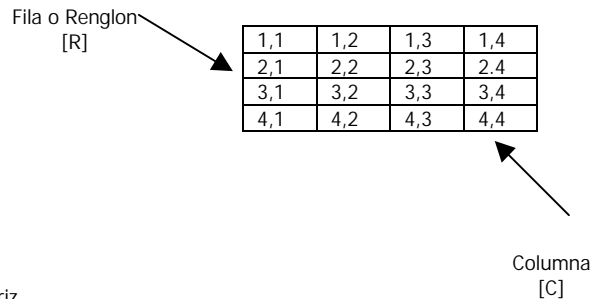
Para representar una matriz se necesita un nombre de matriz acompañado de dos índices.

Mat [R,C]

Donde R indica el renglón y C indica la columna, donde se encuentra almacenado el dato.

Representación gráfica de una matriz

Mat [R,C]



Llenado de una matriz

Por renglones

```
Hacer para R = 1 a 5
  Hacer para C = 1 a 5
    Leer Mat [R,C]
  Fin-para C
Fin-para R
```

Por columnas

```
Hacer para C = 1 a 5
  Hacer para R = 1 a 5
    Leer Mat [R,C]
  Fin-para R
Fin-para C
```

Nota: Para hacer el llenado de una matriz se deben de usar dos variables para los índices y se utilizan 2 ciclos uno para los renglones y otro para las columnas; a estos ciclos se les llama ciclos anidados (un ciclo dentro de otro ciclo).

4.3 Expresiones

Las expresiones son combinaciones de constantes, variables, símbolos de operación, paréntesis y nombres de funciones especiales. Por ejemplo:

$$a+(b + 3)/c$$

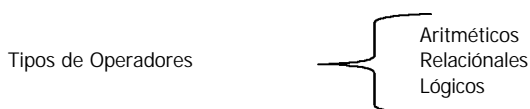
Cada expresión toma un valor que se determina tomando los valores de las variables y constantes implicadas y la ejecución de las operaciones indicadas.

Una expresión consta de operadores y operandos. Según sea el tipo de datos que manipulan, se clasifican las expresiones en:

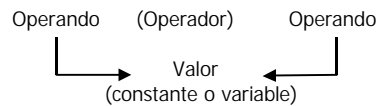
- Aritméticas
- Relacionales
- Lógicas

4.4 Operadores y Operandos

Operadores: Son elementos que se relacionan de forma diferente, los valores de una o mas variables y/o constantes. Es decir, los operadores nos permiten manipular valores.



A) Operadores Aritméticos: Los operadores aritméticos permiten la realización de operaciones matemáticas con los valores (variables y constantes). Los operadores aritméticos pueden ser utilizados con tipos de datos enteros o reales. Si ambos son enteros, el resultado es entero; si alguno de ellos es real, el resultado es real.



Operadores Aritméticos

+	Suma
-	Resta
*	Multiplicación
/	División
Mod	Modulo (residuo de la división entera)

Ejemplos:

Expresión	Resultado
$7 / 2$	3.5
$12 \text{ mod } 7$	5
$4 + 2 * 5$	14

Prioridad de los Operadores Aritméticos

Todas las expresiones entre paréntesis se evalúan primero. Las expresiones con paréntesis anidados se evalúan de dentro a fuera, el paréntesis mas interno se evalúa primero.

Dentro de una misma expresión los operadores se evalúan en el siguiente orden.

- 1.- ^ Exponenciación
- 2.- *, /, mod Multiplicación, división, modulo.
- 3.- +, - Suma y resta.

Los operadores en una misma expresión con igual nivel de prioridad se evalúan de izquierda a derecha.

Ejemplos:

$4 + 2 * 5 = 14$	$46 / 5 = 9.2$
$23 * 2 / 5 = 9.2$	
$3 + 5 * (10 - (2 + 4)) = 23$	$3 + 5 * (10 - 6) = 3 + 5 * 4 = 3 + 20 = 23$
$3.5 + 5.09 - 14.0 / 40 = 5.09$	$3.5 + 5.09 - 3.5 = 8.59 - 3.5 = 5.09$
$2.1 * (1.5 + 3.0 * 4.1) = 28.98$	$2.1 * (1.5 + 12.3) = 2.1 * 13.8 = 28.98$

B) Operadores Relacionales:

- Se utilizan para establecer una relación entre dos valores.
- Compara estos valores entre si y esta comparación produce un resultado de certeza o falsedad (verdadero o falso).
- Los operadores relacionales comparan valores del mismo tipo (numéricos o cadenas)
- Tienen el mismo nivel de prioridad en su evaluación.
- Los operadores relacionales tiene menor prioridad que los aritméticos.

4.4.1.1 Operadores Relacionales

>	Mayor que
<	Menor que
> =	Mayor o igual que
< =	Menor o igual que
< >	Diferente
=	Igual

Ejemplos lógicos:

Si $a = 10$ $b = 20$ $c = 30$

$a + b > c$ Falso
 $a - b < c$ Verdadero
 $a - b = c$ Falso
 $a * b < > c$ Verdadero

Ejemplos no lógicos:

$a < b < c$
 $10 < 20 < 30$
 $"T" < 30$ (no es lógico porque tiene diferentes operandos)

C) Operadores Lógicos:

Estos operadores se utilizan para establecer relaciones entre valores lógicos y pueden ser resultado de una expresión relacional.

Operadores Lógicos

And Y
 Or O
 Not Negación

4.4.1.2 Operador And

Operando1	Operador	Operando2	Resultado
T	AND	T	T
T		F	F
F		T	F
F		F	F

4.4.1.3 Operador Or

Operando1	Operador	Operando2	Resultado
T	OR	T	T
T		F	T
F		T	T
F		F	F

4.4.1.4 Operador Not

Operando	Resultado
T	F
F	T

Ejemplos:

$(a < b)$ and $(b < c)$
 $(10 < 20)$ and $(20 < 30)$
 T and T
 ↳ T ◀

4.4.1.5 Prioridad de los Operadores Lógicos

Not
 And
 Or

4.4.1.6 Prioridad de los Operadores en General

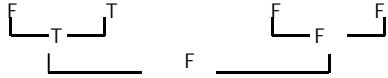
- 1.- ()
- 2.- ^
- 3.- *, /, Mod, Not

- 4.- +, -, And
- 5.- >, <, >=, <=, < >, =, Or

Ejemplos:

a = 10 b = 12 c = 13 d = 10

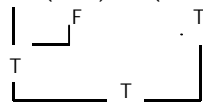
1) ((a > b) or (a < c)) and ((a = c) or (a >= b))



2) ((a >= b) or (a < d)) and ((a >= d) and (c > d))



3) not (a = c) and (c > b)



4.5 Identificadores

Los *identificadores* representan los datos de un programa (constantes, variables, tipos de datos). Un identificador es una secuencia de caracteres que sirve para identificar una posición en la memoria de la computadora, que nos permite acceder a su contenido.

Ejemplo: Nombre
 Num_hrs
 Calif2

Reglas para formar un Identificador

Debe comenzar con una letra (A a Z, mayúsculas o minúsculas) y no deben contener espacios en blanco. Letras, dígitos y caracteres como la subraya (_) están permitidos después del primer carácter. La longitud de identificadores puede ser de hasta 8 caracteres.

Constantes y Variables

Constante: Una constante es un dato numérico o alfanumérico que no cambia durante la ejecución del programa.

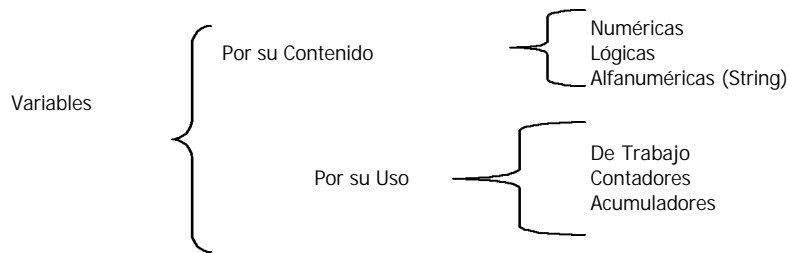
Ejemplo: pi = 3,1416

Variable: Es un espacio en la memoria de la computadora que permite almacenar temporalmente un dato durante la ejecución de un proceso, su contenido puede cambia durante la ejecución del programa. Para poder reconocer una variable en la memoria de la computadora, es necesario darle un nombre con el cual podamos identificarla dentro de un algoritmo.

Ejemplo: área = pi * radio ^ 2

Las variables son : radio, área y constate es pi

Clasificación de las Variables



4.5.1.1.1 Por su Contenido

Variables Numéricas: Son aquellas en las cuales se almacenan valores numéricos, positivos o negativos, es decir almacenan números del 0 al 9, signos (+ y -) y el punto decimal. Ejemplo:

iva=0,15 pi=3,1416 costo=2500

Variables Lógicas: Son aquellas que solo pueden tener dos valores (verdadero o falso) estos representan el resultado de una comparación entre otros datos.

Variables Alfanuméricas: Esta formada por caracteres alfanuméricos (letras, números y caracteres especiales). Ejemplo:

letra='a' apellido='lopez' direccion='Av. Libertad #190'

4.5.1.1.2 Por su Uso

Variables de Trabajo: Variables que reciben el resultado de una operación matemática completa y que se usan normalmente dentro de un programa. Ejemplo:

suma=a+b/c

Contadores: Se utilizan para llevar el control del número de ocasiones en que se realiza una operación o se cumple una condición. Con los incrementos generalmente de uno en uno. Ejemplo: c=c+1

Acumuladores (Sumadores): Forma que toma una variable y que sirve para llevar la suma acumulativa de una serie de valores que se van leyendo o calculando progresivamente. Ejemplo: Sumador = Sumador + variable

Capítulo 5. Estructuras Algorítmicas

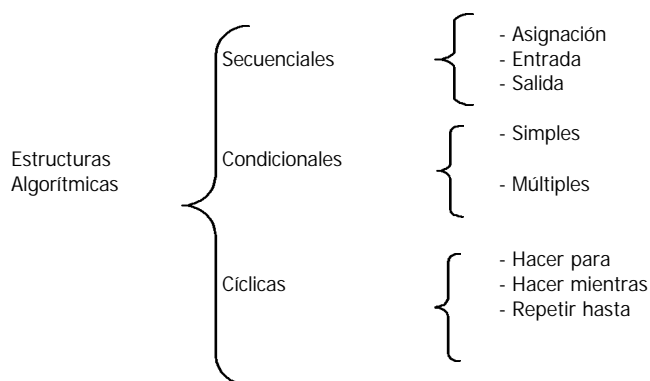
5.1 Objetivo Educativo

El alumno:

Conocerá las diferentes estructuras algorítmicas como componentes básicos de los programas y aplicara la combinación de ellas para el desarrollo de algoritmos mas complejos.

5.2 Estructuras Algorítmicas

Las estructuras de operación de programas son un grupo de formas de trabajo, que permiten, mediante la manipulación de variables, realizar ciertos procesos específicos que nos lleven a la solución de problemas. Estas estructuras se clasifican de acuerdo con su complejidad en:



5.3 Estructuras Secuenciales

La estructura secuencial es aquella en la que una acción (instrucción) sigue a otra en secuencia. Las tareas se suceden de tal modo que la salida de una es la entrada de la siguiente y así sucesivamente hasta el fin del proceso. Una estructura secuencial se representa de la siguiente forma:

```

Inicio
  Accion1
  Accion2
  .
  .
  AccionN
Fin
  
```

Asignación: La asignación consiste, en el paso de valores o resultados a una zona de la memoria. Dicha zona será reconocida con el nombre de la variable que recibe el valor. La asignación se puede clasificar de la siguiente forma:

- **Simples:** Consiste en pasar un valor constante a una variable ($a=15$)
- **Contador:** Consiste en usarla como un verificador del número de veces que se realiza un proceso ($a=a+1$)
- **Acumulador:** Consiste en usarla como un sumador en un proceso ($a=a+b$)
- **De trabajo:** Donde puede recibir el resultado de una operación matemática que involucre muchas variables ($a=c+b*2/4$).

Lectura: La lectura consiste en recibir desde un dispositivo de entrada (p.ej. el teclado) un valor. Esta operación se representa en un pseudocódigo como sigue:

Leer a, b
Donde "a" y "b" son las variables que recibirán los valores

Escritura: Consiste en mandar por un dispositivo de salida (p.ej. monitor o impresora) un resultado o mensaje. Este proceso se representa en un pseudocódigo como sigue:

Escribe "El resultado es:", R
Donde "El resultado es:" es un mensaje que se desea aparezca y R es una variable que contiene un valor.

Capítulo 6. Técnicas de Diseño.

6.1 Objetivo Educativo

El Alumno:

- (a) Conocerá las características técnicas de diseño más empleadas.

6.2 Programación Estructurada

El objetivo de la Programación Estructurada es organizar y disciplinar el diseño de programas y el proceso de codificación con la finalidad de evitar la mayoría de los errores de lógica y de facilitar la detección de los errores que permanezcan.

La programación se concentra en uno de los factores más propensos a los errores, la lógica. La programación estructurada tiene 3 importantes características:

- El Diseño Top-Down.
- La Programación Modular.
- La Codificación Estructurada.

6.2.1 Diseño Top-Down

El método más conveniente para resolver un problema de mediana complejidad es dividirlo o descomponerlo en subproblemas más simples. Es decir, si se tiene un problema P se trata de determinar un conjunto de problemas más simples P1, P2, ..., Pn, tales que solucionando primero P1, luego P2, ...,etc., signifique resolver P.

El mismo método puede aplicarse a cada uno de los sub problemas anteriores, es decir, decir, determinar una secuencia de sub problemas P1.1, P1.2, P1.3, ..., etc, cuyas soluciones impliquen la solución de P1. La descomposición de un problema en sub problemas aporta detalles de como resolver el problema. El proceso debe continuar hasta que la solución de un sub problema pueda expresarse en instrucciones de un lenguaje de programación.

El método llamado de "Refinamientos Sucesivos" o también de Desarrollo Top-Down de Programas, consiste, en general, en concentrarse inicialmente en QUE hacer, postergando los detalles de COMO hacerlo. Este proceso de refinamientos sucesivos conducirá a expresar la solución del problema en una forma que el computador entienda.

Se utilizará un pseudocódigo o pseudolenguaje, lenguaje ficticio semejante a los lenguajes de programación, para expresar la lógica del programa de una manera altamente legible. El pseudocódigo permitirá establecer la lógica del programa claramente ignorando las restricciones de la máquina.

En resumen, el Diseño Top-Down consisten básicamente en producir una serie de versiones del programa:

- Versión Inicial. La cual es una concepción informal pero precisa del problema.
- Versiones Intermedias. Las cuales son soluciones en base a instrucciones generales y abstractas.
- Versión Final. Esta solución contiene instrucciones para su implementación en un lenguaje determinado.

6.2.2 Programación Modular.

Es una técnica de desarrollo de programas que se caracteriza por dar énfasis a la definición de módulos, de manera tal que el programa final sea una combinación jerarquizada de estos módulos. Cada módulo debe cumplir una función bien específica (cohesión). Una vez que un problema grande es dividido en unidades más pequeñas, lógicas y más manejables, el programa resultante es más fácil de comprender y de leer. Esta es una técnica común para controlar la complejidad del programa (dividir y conquistar).

Si el diseño top-down para el problema ha sido realizado, esta ya ha sido dividido en sub problemas que constituirán posibles módulos.

Al usar la Programación Modular deben considerarse los siguientes objetivos:

- a) Debemos ser capaces de convencernos de que el módulo del programa está correcto independiente del contexto en el cual se usará.
- b) Debemos ser capaces de agrupar módulos para formar programas más grandes sin ningún conocimiento previo de la fórmula de trabajo interno de cada módulo. Las subrutinas científicas o programas utilitarios son ejemplos de módulos exitosos.

Independencia Entre Módulos.

En un programa modular, cada módulo, por ejemplo cada sub rutina en FORTRAN, es independiente de los otros. Esto implica, por supuesto que un módulo puede ser cambiado o modificado sin afectar a los otros módulos (acoplamiento). En todo caso, debe quedar claro que el concepto de independencia no es absoluto. Lo que se desea establecer es si un aspecto del programa cambia, ¿cuánto afectará a un módulo determinado?

Un módulo debe considerar los siguientes elementos.

- a) Un algoritmo para resolver el problema.
- b) Un conjunto de valores permitidos de entrada (dominio).
- c) Un conjunto de posibles valores de salida (rango).
- d) Un conjunto de efectos laterales, por ejemplo, definir las acciones a realizar en caso de errores en el dominio.

Ventajas de la Modularidad.

Se pueden considerar las siguientes:

- a) Facilita la modificación de los programas, ya que los procesos están separados.
- b) Facilita la comprensión del programa, pues solamente se requiere entender la parte del programa que interesa.
- c) Mejor organización, esto se refleja en procesos que son efectivamente rutinas claras y diferenciadas.
- d) Posibilita la programación en grupo, de modo que cada programador pueda desarrollar un módulo.
- e) Facilita la programación misma, al poder tratar cada módulo separadamente.

6.2.3 Codificación estructurada.

Las personas tienden a ser procesadores secuenciales, es decir, a leer y procesar las cosas secuencialmente. Por lo tanto, a mayor cantidad de quiebres de secuencia (sentencias GOTO) mayor es la dificultad que se tiene para seguir la lógica del programa.

La codificación estructurada es un método para escribir programas con un alto grado de estructuración. Permite escribir programas que son más fáciles de entender, probar y modificar. Un programa, no importa su tamaño y complejidad, se puede escribir usando un pequeño conjunto de estructuras básicas de codificación.

Teorema de la codificación estructurada.

Los autores C. BOHM y G. JACOPINI demostraron que cualquier programa se puede escribir solo con 3 tipos de sentencias:

- a) Una sentencia de acción o de secuencia, que no modifica el flujo de ejecución del programa.
- b) Una sentencia condicional (IF-THEN-ELSE) que según sea el valor de la comparación o relación ejecutará una u otra alternativa.
- c) Una sentencia de iteración que permita efectuar una operación cero o más veces mientras una expresión lógica sea verdadera (DO-WHILE).

Aunque teóricamente es posible escribir cualquier programa usando solamente las 3 estructuras básicas analizadas anteriormente, al programar se encuentra que es conveniente expandir un poco dicho repertorio de estructuras. Las sentencias DO-UNTIL (REPEAT) y CASE son comúnmente las dos estructuras incorporadas a las anteriores.

6.2.4 Resumen programación estructurada.

La Programación Estructurada se puede definir como un conjunto de técnicas de diseño de programas que se caracterizan por hacer un uso sistemático de la abstracción y que permiten obtener programas confiables, fáciles de comprender y modificar. También se puede definir como la actitud de escribir el código con la intención de comunicarse más bien con las personas que con las máquinas.

Los elementos básicos a considerar en la Programación Estructurada son

- a) El código se construye utilizando 3 elementos básicos
 - Sentencia de acción o de secuencia.
 - Sentencia condicional.
 - Sentencia de iteración.
- b) El uso de la instrucción GOTO se evita siempre que sea posible. En particular, el peor uso del GOTO es aquel que implica un salto a una sentencia previa en el programa.
- c) El código debe ser escrito con un estilo de programación aceptable (legibilidad de los programas).
- d) Cada módulo debe tener solo un punto de entrada y solo un punto de salida.
- e) Segmentación física del código en el listado para realzar la legibilidad. Las sentencias ejecutables para un módulo deben, en lo posible, ajustarse a una página de listado.
- f) El código representa una solución simple y directa del problema.

6.3 ESTILO DE PROGRAMACION (SIMPLICIDAD Y LEGIBILIDAD).

Esta orientado a programación que permitan obtener programas correctos mantenibles y legibles. Es importante considerar que son las personas las que deben leer y comprender los programas para corregirlos, mantenerlos y modificarlos.

Sugerencias para la Simplicidad en el Programa.

- Usar las técnicas proporcionadas por la Programación Estructurada.
- Evitar el mal uso de las instrucciones del lenguaje de programación.
- No escribir programas que se modifiquen a si mismos en tiempo de ejecución.

Sugerencias para la Legibilidad de los Programas

- Comentarios a considerar:
- a) Descripción del trabajo que efectúa el programa.
 - b) Descripción de variables, lista de sub programas que utiliza, fecha en que se escribió, autor, etc.
 - Líneas en blanco (espaciamiento vertical)
 - Espacios en la línea de codificación.
 - Identificación y numero de secuencia.
 - Selección de nombres de variables.
 - Nombres de archivos
 - Uso de abreviaciones estándar.
 - División de palabras (nombres de variables, literales, etc.).
 - Ubicación de declaraciones (una declaración por línea es suficiente y permite una mayor legibilidad y facilidad de modificaciones en el programa).
 - Ordenar las listas alfabéticamente (facilidad para encontrar un nombre de dato determinado en la lista).
 - Uso de paréntesis en expresiones.
 - Formatear el programa fuente (indentación)

Capítulo 7. Estudio de lenguajes de programación

7.1 ¿Por qué estudiar LP?

Hay muchos, pero realmente utilizamos pocos

7.1.1 Seis razones

- Mejorar la habilidad para desarrollar algoritmos eficaces (capacidad v/s experiencia)
- Mejorar el uso del lenguaje de programación disponible (herramienta v/s conocimiento)
- Acrecentar el propio vocabulario con construcciones útiles sobre programación (solución óptima v/s herramienta óptima)
- Hacer posible una mejor elección del lenguaje de programación (problema v/s herramienta)
- Facilitar el aprendizaje de un nuevo lenguaje (comprensión general → aprendizaje ilimitado)
- Facilitar el diseño de un nuevo lenguaje (comprensión de diseño de aplicaciones v/s comprensión y desarrollo de un nuevo lenguaje)

El costo de implementación depende de la herramienta o lenguaje utilizado.

7.2 Historia

Los lenguajes inicialmente fueron utilizados con fines militares, fue así como en los:

'50 → A-O, Speed coding, Fortran'60 → Ada, C, Pascal, Prolog, Smalltalk
'70 → C++, ML

7.2.1 Lenguajes basados en el cálculo numérico

Se implementaron lenguajes para solucionar cálculos numéricos o matemáticos, llamaban calculadoras matemáticas.

- Jovial
- Fortran (traductor de fórmulas, versiones II, IV, 66, 70, 90) (Meta: eficiencia en IBM).
- Algol (lenguaje algorítmico)(matemática pura) (Meta: Solución en matemática pura y abarcar variadas arquitecturas)

Se presentó un caos en la orientación y reglas de programación, GAMM y ACM crean Algol que respondía a una estandarización mundial y luego con la intervención de Backus y Naur; y el estudio previo de Chomsky, se crea la BNF(Backus Naur Form) o sea "Teoría Formal de la gramática al mundo de los Lenguajes de Programación".

7.2.2 Lenguajes para negocios

- Flow matic (1955)
- CBL (lenguaje común para negocios, 1959) del Depto. Defensa de USA
- Cobol (lenguaje común orientado a negocios)

7.2.3 Lenguajes para inteligencia artificial

- IPL (lenguaje de procesamiento de información)
- Lisp (procesamiento de listas → IBM)
- Comit (Yague del Mit)
- Snobol (laboratorio Bell de AT&T)

7.2.4 Lenguajes para sistemas operativos

- PL/I, BCPL, B → Multiplics → Unix
- C → a B se le agregó tipos, estructura de datos, operaciones → Unix

7.2.5 Lenguajes para internet

- SGML (señalización gramática meta lenguaje)

- HTML (hiper texto meta lenguaje)
- Lenguaje de Scripts (vbscript, javascript, pearl) para interactividad con usuario, base de datos, internet, etc.

7.3 Aplicabilidad de LP

7.3.1 Problemas de aplicabilidad

- Altos Costos de maquinas v/s bajos costos de programación.
- Altos costos de programación v/s bajos costos de maquinas.
- Tendencia de exclusividad por área para los lenguajes.
- Des – uso de lenguajes antiguos por otros o nuevas tendencias.

7.3.2 Definición de nuevos lenguajes por

(a) Capacidad de los lenguajes

- Costos
- Sistemas operativos
- Hardware
- Memoria y procesador (rapidez de la maquina)

Aplicaciones

- Antiguas tendencias: militares, calculos, negocios
- Nuevas tendencias: juegos, bd, red, pc's

Métodos de programación

- Entorno de interfaz gráfica
- Pequeños trozos y metodos (generación de codigo)

Métodos de implementación

- Nuevos metodos o estilos de programación cambian los diseños para implementar de acuerdo a nuevas características

Estudios teóricos

- Fortalezas y debilidades de los lenguajes

Estandarización

- Aumentar o fomentar la compatibilidad del lenguaje según equipo, plataforma o base de datos.

7.4 Un buen LP

7.4.1 Atributos Generales

- a) Claridad, sencillez y unidad: sintaxis, conceptos (sencillas combinaciones en el código)
- b) Ortogonalidad: evaluar cualquier expresión o combinación, para esto menos casos especiales que recordar
- c) Naturalidad: adoptar facilmente la estructura a los tipos de aplicaciones.
- d) Apoyo a la abstracción: utilización de clases
- e) Facilidad para verificar los programas: puntos de quiebre, visualización de variables, compilación en linea.
- f) Entorno de programación: interfaz gráfica.
- g) Portabilidad de programas: compatibilidad y transporte de programas.
- h) Costo de uso:
 - Ejecución (tiempo y diagnostico)
 - Traducción (tiempo y ejecutable optimizado)
 - Creación, prueba y uso (minimizar esfuerzo de programación)
 - Mantenimiento(errores, s.o., hw, nuevas necesidades)