Índice

1. Resumen sintaxis pseudocódigo

1.1. Tipos de datos

En pseudocódigo hay varios tipos de datos:

- Simples
  - Numéricos.
    - Entero.
    - Real.
  - Lógico.
  - Carácter.
- Estructurados
  - Vectores (empiezan en 1 en pseudocódigo, cuidadito)
  - Estructuras o registros.
  - Matrices.

1.2. Declaración de variables y constantes

De esta forma declaramos nuestras variables y constantes de cualquier tipo de dato.

```
const
   real:PI = 3.1416

var
   entero:Suspensos
```

1.3. Variables locales y globales

Con la nueva modificación del pseudocódigo las variables y/o constantes globales se colocan antes de la palabra reservada **Principal**(como ya se hacía). Y las locales luego del principal y antes del inicio. Tal que así:

```
Algoritmo nombreAlgoritmo
      const
         sección de definición de constantes.
      tipos
         sección de definición de tipos.
      var
         sección de definición de variables globales.
   Principal
      var
         sección de defnición de variables locales a Principal.
      Inicio
         inicialización de variables...
         instrucciones del algoritmo principal...
   fin_principal
      definición de funciones y procedimientos.
fin_algoritmo
```

1.4. Operadores

Operadores (Lógicos)	Definición
=	Compara dos elementos y devuelve verdadero si son iguales
≠ o <>	Compara dos elementos y devuelve verdadero si son diferentes
y	AND lógico.Funciona como en el lenguaje natural.
o	OR lógico. Funciona como en el lenguaje natural.
no	NOT lógico. Funciona como en el lenguaje natural.

Operadores(aritméticos)	Definición
+	Suma
-	Resta
*	Multiplicación
/	División entera(solo pilla el cociente sin decimales) Ej.: $5/2 = 2$
div	División real(con decimales)
mod	Resto de la división entera. Ej.: $5 \bmod 2 = 1$

1.5. Sentencias de control

1.5.1. Estructuras selectivas

Simple

```
si condición entonces
   instrucciones...
fin_si
```

Doble

```
si condición entonces
   instrucciones Bloque A...
si_no
   instrucciones Bloque B...
fin_si
```

Múltiple

```
segun_sea expresión hacer
   1:
      instrucciones Bloque 1...
   2:
      instrucciones Bloque 2...
   3:
      instrucciones Bloque 3...
   ....
   n:
      instrucciones Bloque n...
   en_otro_caso:
      instrucciones...
fin_segun
```

1.5.2. Repetitivas

Mientras

```
mientras (condición) hacer:
   instrucciones...
fin_mientras
```

Repetir

```
repetir
   instrucciones...
hasta_que (condición)
```

Nota: Cuidadito que esta estructura se repite mientras la condición es FALSA y sale cuando la condición es verdadera.

Desde

```
desde i⟵ Vi hasta Vf hacer
   instrucciones...
fin_desde
```

- V_i es el valor inicial.
- V_f el valor final.

Nota: Internamente el bucle **desde** aumenta en 1 la variable `i`, aunque si queremos aumentarlo de otra manera, detras del **hacer** debemos poner un `i ⟵ i + n` , siendo `n` el aumento que se le va a dar a la variable `i` en cada paso.

1.6. Funciones y procedimientos

Las funciones son pequeños trozos de código que devuelven un valor a donde fueron llamadas. Los procedimientos son pequeños trozos de código que no devuelven nada.

Sintaxis Funciones:

```
especificación de la función
tipo del resultado función nombre función ( lista de parámetros formales E o S o E/S)
   var
      variables locales a la funcion.
   inicio:
      instrucciones
      devolver(expresión resultado)
fin_función
```

Sintaxis Procedimientos:

```
especificación del procedimiento
procedimiento nombre procedimiento ( lista de parámetros formales E o S o E/S)
   var
      variables locales al procedimiento.
   inicio:
      instrucciones
fin_procedimiento
```

1.7. Tipos de datos estructurados

1.7.1. Vectores (popularmente conocido como arrays o arreglos)

Para declarar un vector usamos la siguiente sintaxis:

```
vector [tamaño] de tipo de dato : identificador del tipo vector
```

Para declararlo lo hacemos igual que con las variables pero ya será con nuestro tipo, ejemplo:

```
tipo
     vector [30] de entero : Mes
var
     Mes : miMes
```

Para acceder a los elementos colocamos el identificador de nuestra variable de tipo vector y le agregamos unos corchetes con la posición del elemento al que queremos acceder o modificar dentro. Ejemplo:

```
miMes[1]⟵ 20
```

IMPORTANTE: Los vectores en pseudocódigo comienzan en la posición 1.

1.7.2. Cadenas

Las cadenas son solo vectores de caracteres pero por su amplio uso se declaran directamente tal que así:

```
cadena : identificador del tipo cadena
```

Para asignar una cadena utilizaríamos el operador de asignacion y la cadena a asignar entre comillas. Para acceder a un carácter concreto se hace como en los vectores. Hay unas cuantas funciones que nos ayudan en pseudocódigo con su manejo:

```
longitud(cadena) // Devuelve la longitud de la cadena pasada
compara(cadena1, cadena2) // Devuelve 0 si las cadenas son iguales, y
                          // distinto de 0 en caso contrario.
concatena(cadena1, cadena2) // Devuelve las cadenas 1 y 2 unidas
cadena1 + cadena2 // Otra forma de concatenar cadenas
```

1.7.3. Matrices

Para declarar una matriz usamos la siguiente sintaxis:

```
matriz [tamaño1, tamaño2, ... , tamañoN] de tipo de dato : identificador del tipo matriz
```

Ejemplo:

```
tipo
     matriz [10,30] de real : ventasSupermercadosMes
var
     ventasSupermercadosMes : misVentasPorSupermercado
```

Para acceder a los elementos de la matriz se utiliza la misma sintaxis que para los vectores, solo que habra N indices, para N dimensiones, ejemplo para la matriz anterior (que es de dos dimensiones):

```
misVentasPorSupermercado[2][5]⟵ 23,21
```

1.7.4. Registros o estructuras

Para declarar un vector usamos la siguiente sintaxis:

```
tipo
      registro : nombreRegistro
            tipo1 : identificadorCampo1
            tipo2 : identificadorCampo1
            tipo3 : identificadorCampo1
            tipo4 : identificadorCampo1
            .....
            tipoN : identificadorCampoN
      fin_registro
```

Ejemplo para representar los datos de un DNI:

```
tipo
      registro : DNI
            cadena : nombre
            cadena : apellidos
            entero : edad
            cadena : Direccion
            cadena : nombrePadre
            cadena : nombreMadre
            cadena : numeroDNI
      fin_registro
```

El acceso a los distintos campos se realiza con la siguiente sintaxis(para el ejemplo anterior):

```
var
      DNI : miDNI

miDNI.nombre⟵"Manolo"
miDNI.edad⟵19
miDNI.apellidos⟵"Español Español"
....
```

1.8. Ficheros

Para declarar un fichero usamos la siguiente sintaxis:

```
tipo
      archivo de tipo: tipoFichero
```

Para definirlo se hace igual que con todo lo anterior. Hay varias funciones que nos posibilitan el manejo de archivos:

```
abrir(varFichero, modo, nombreFichero)
```

Siendo

- `nombreFichero` la variable de tipo fichero definida anteriormente.
- `varFichero` el nombre del fichero que se encuentra en la memoria masiva.
- `modo` el modo de acceso al fichero **lectura** o **escritura.**

La función **`escribir`** que escribe `elemento` en el archivo.

```
escribir(varFichero, elemento)
```

La función **leer** que lee del fichero y lo guarda en **elemento**.

```
leer(varFichero, elemento)
```

La función **feof** que comprueba si estamos en el final del fichero.

```
feof(varFichero)
```

La función **cerrar** que cierra el fichero.

```
cerrar(varFichero)
```

IMPORTANTE: ¡¡¡Para hacer algo con un archivo hay que abrirlo primero!!! ¡¡¡Una vez terminemos de usar un archivo hay que cerrarlo!!!

2. Teoría

2.1. Objetivos de la programación

- **Corrección:** Antes de desarrollar un programa debe especificarse con toda claridad cuál es el funcionamiento correcto del mismo.
- **Claridad:** Es fundamental que sus descripciones sean claras y fácilmente legibles.
- **Eficiencia:** una tarea de tratamiento de información puede ser programada de muy diferentes maneras. Los programas eficientes aprovecharán mejor los recursos disponibles.

2.2. Fases de creación de un programa

1. Planteamiento y análisis del problema
   - Definición
   - Representación de los datos
     - Entradas
     - Salidas
2. Diseño del Algoritmo
   - Descomposición en subproblemas
   - Diseño descendente
   - Refinamientos sucesivos
   - Optimización y Depuración
3. Implementación
   - Elección del lenguaje
   - Adaptación al problema
   - Conocimientos
   - Disponibilidad de rutinas

- Herramientas de desarrollo
- Entorno de la aplicación
- Codificación (normas de estilo)

4. Ejecución
   - Traducción
   - Prueba y Depuración

2.3. Estructuras repetitivas

Sustitución de unas estructuras por otras:

- Podemos sustituir siempre una estructura desde por una estructura repetir o por una estructura mientras.
- Una estructura repetir o mientras, sin embargo, se puede sustituir por una estructura desde sólo cuándo se conoce de antemano el número de veces que van a ejecutarse las acciones del bucle.
- Una estructura mientras puede sustituirse por una estructura repetir cuando no altere el resultado el hecho de que las acciones del bucle se ejecuten al menos una vez.
- Una estructura repetir siempre puede sustituirse por una estructura mientras. En el peor de los casos, también habría que escribir el conjunto de instrucciones del bucle antes del mismo, para que se ejecuten siempre al menos una vez.
- Estas afirmaciones pueden variar en función de cómo estén definidas las diferentes estructuras en cada lenguaje de programación