Pseudocódigo

Jesús Rodríguez Heras

16 de febrero de 2017

Índice

1. Introducción

El pseudocódigo es un lenguaje natural en español sujeto a determinadas reglas y con estructura similar a la de un programa, lo que facilita su traducción a lenguaje de alto nivel, por ejemplo: C. Debido a que no se puede implementar ningún algoritmo con pseudocódigo, su objetivo principal se basa en diseñar dichos algoritmos que pueden tener una o varias implementaciones en los lenguajes de programación de alto nivel.

1.1. Estructura del pseudocódigo

La estructura general del pseudocódigo es la siguiente:

```
1  //Comentarios que faciliten la comprension del algoritmo como
2  //datos de entrada y salida.
3  Algoritmo Nombre_algoritmo
4  const
5    //Seccion de definicion de constantes
6  tipos
7    //Seccion de definicion de tipos creados por el programador
8  var
9    //Seccion de declaracion de variables
10 inicio
11   //Inicializacion de variables
12   Accion 1
13   Accion 2
14   .
15   .    //Cuerpo del algoritmo
16   Accion n
17 fin_algoritmo
```

1.2. Variables y expresiones

1.2.1. Variables y constantes

Una variable es un objeto con un nombre significativo en el programa que puede variar su valor durante la ejecución del programa. Será declarada en la sección `var`.

```
1 var
2 entero: suma
```

Una constante es un objeto que contiene un valor que no varía durante la ejecución de un programa y será declarada en la sección `const`.

```
1 const
2 PI = 3.1415
```

1.2.2. Asignaciones y expresiones aritméticas

Las asignaciones se realizan usando el carácter "←". Se utiliza para dar valores a las variables. Por ejemplo:
`notas←8`
Si se realiza una siguiente asignación, el valor de la variable será el de la ultima asignación y el anterior desaparece.

En cuanto a las expresiones aritméticas tenemos:

Operador	Significado
+	Suma
-	Resta
*	Multiplicación
/	División entera
div	División real
mod	Resto (módulo)

Tabla 1: Expresiones aritméticas.

Las reglas de prioridad de dichas expresiones son muy simples. Primero se evalúan las operaciones que se encuentran entre paréntesis (si hay paréntesis anidados, se empieza por los más internos) y luego se evalúan la multiplicación y la división, y, por último, la suma y la resta. Si aparece más de un operador de la misma propiedad se avanzará de izquierda a derecha.

1.2.3. Expresiones lógicas

Se evalúan constantes, variables y expresiones de tipo lógico usando operadores lógicos y relacionales.

Operador	Significado
<	Menor que
>	Mayor que
=	Igual que
$\leq$	Menor o igual que
$\geq$	Mayor o igual que
$\neq$	Distinto de

Tabla 2: Expresiones lógicas.

Las reglas de prioridad de estas expresiones para datos de tipo numérico son las que se emplean en matemáticas. Para datos de tipo carácter se ordenan según su código ASCII.

1.2.4. Lectura y escritura

Son las operaciones de entrada y salida básicas.
Por ejemplo:
`leer (a)` Lee un valor de la entrada estándar (teclado) y lo almacena en la variable `a`.
`escribir (a)` Escribe lo que haya en la variable `a`.

2. Programación estructurada

2.1. Características de la programación estructurada

La programación estructurada es el conjunto de técnicas que incorporan:
- Recursos abstractos.
- Diseño descendente.
- Estructuras básicas: secuenciales, selectivas, repetitivas, etc.

2.2. Estructura secuencial

La estructura secuencial es el descenso del programa instrucción a instrucción en forma de cascada. Un ejemplo en el que podemos ver la secuencia de instrucciones es el siguiente:

```
1  Algoritmo suma_producto
2    Principal
3    var
4      entero: A, B, S, P
5    inicio
6      escribir("Introduce␣dos␣numeros")
7      leer (A, B)
8      S <- A+B
9      P <- A*B
10     escribir (S, P)
11   fin_principal
12 fin_algoritmo
```

Como podemos observar que en este ejemplo, se tienen que ir cumpliendo las instrucciones una a una para que el programa termine. Si alguna de las instrucciones falla, el programa dejaría de funcionar.
La programación secuencial establece que para cada inicio de un bloque y/o estructura debe haber un final de la misma. Por lo tanto, para cada `Principal` hay un `fin_principal`, para cada `si` un `fin_si`, etc.

2.3. Estructura selectiva

En este tipo de estructuras, la ejecución de un conjunto de instrucciones dependerá del resultado de la evaluación previa de una determinada condición como verdadero o falso.
Podemos diferenciar tres tipos de estructuras selectivas:

2.3.1. Simple

Las instrucciones que están a continuación del condicional (cláusula `si`) solo se ejecutarán en caso de que éste resulte verdadero en la evaluación. Si no lo es, sencillamente, se salta dichas instrucciones.
Por ejemplo: Decir si un número es o no mayor que cero.

```
1  Algoritmo mayor_que_cero
2    Principal
3    var
4      entero: numero
5    inicio
6      escribir ("Introduce un numero:")
7      leer (numero)
8      leer(numero)
9      si (numero>0) entonces
10       escribir ("Numero mayor que cero.")
11     fin_si
12   fin_principal
13 fin_algoritmo
```

En este ejemplo, si el número introducido fuese menor o igual a cero, el programa terminaría y no escribiría nada.

2.3.2. Doble

Si el condicional es cierto se ejecutan las instrucciones que están justo después de la evaluación del mismo (cláusula `si`). Si fuera falso, se ejecutarían las instrucciones que están después, en la cláusula `si_no`.
Por ejemplo: Decir si un número es par o impar.

```
1  Algoritmo par_impar
2    Principal
3    var
4      entero: n
5    inicio
6      escribir ("Introduce un numero:")
7      leer (n)
8      si (n mod 2 = 0) entonces
9        escribir("PAR")
10     si_no
11       escribir("IMPAR")
12     fin_si
13   fin_principal
14 fin_algoritmo
```

En este ejemplo, si el número introducido es par, se ejecuta la cláusula `si` al evaluarse como verdadera la condición. Si el número es impar, se ejecuta la cláusula `si_no` al evaluarse como falsa la condición.

2.3.3. Múltiple

Este tipo de estructura selectiva evalúa una condición que, si coincide con alguno de los casos establecidos, realiza las instrucciones que se encuentren en dicho caso.
Por ejemplo: Asignar un número a los días de la semana.

```
1  Algoritmo Nombre_dias_semana
2    Principal
3    var
4      entero: dia
5    inicio
6      escribir ("Introduzca un numero de dia")
7      leer(dia)
8      segun_sea (dia) hacer
9        1: escribir("LUNES")
10       2: escribir("MARTES")
11       3: escribir("MIERCOLES")
12       4: escribir("JUEVES")
13       5: escribir("VIERNES")
14       6: escribir("SABADO")
15       7: escribir("DOMINGO")
16       en_otro_caso
17         escribir("Dia incorrecto")
18     fin_segun
19   fin_principal
20 fin_algoritmo
```

En este ejemplo, se evalúa el número introducido por el usuario, y si coincide con aluno de los siete casos establecidos, escribirá el día de la semana correspondiente con el número introducido. En caso de que el número introducido no se encuentre entre los casos contemplados, el programa devolverá la frase `Día incorrecto` correspondiente a la cláusula `en_otro_caso`.

2.4. Estructuras repetitivas

Este tipo de estructuras permiten repetir una o varias instrucciones varias veces en función del resultado de la evaluación de una condición. Son los llamados bucles, y se llama iteración a cada repetición de la secuencia de instrucciones del bucle.
Existen tres tipos de estructuras repetitivas:

2.4.1. Mientras

En este tipo de estructura, el cuerpo del bucle se ejecuta mientras que la condición del mismo se evalúe como verdadera. Debido a su estructura, la condición del bucle es evaluada antes de entrar por primera vez, es decir, al inicio de cada iteración.
Por ejemplo: Multiplicar mediante sumas sucesivas.

```
Algoritmo producto_sumas
  Principal
  var
    entero: a, b, cont, prod
  inicio
    escribir("Introduzca dos numeros:")
    leer(a, b)
    cont <- 0
    prod <- 0
    mientras (cont < B) hacer
      prod <- prod + A
      cont <- cont + 1
    fin_mientras
    escribir (prod)
  fin_principal
fin_algoritmo
```

Como se puede ver en el ejemplo, antes de entrar en el cuerpo del bucle, se evalúa la condición de entrada del mismo. Y así durante todas las iteraciones hasta que la condición sea evaluada como falsa y salga del bucle para continuar por la instrucción que le sigue al `fin_mientras`.

2.4.2. Repetir

En este tipo de estructura, la condición del bucle se encuentra al final del cuerpo del mismo por lo que las instrucciones que pertenezcan al cuerpo del bucle se ejecutarán como mínimo una vez, hasta llegar a la condición. Estas instrucciones se reiterarán hasta que se evalúe como verdadera la condición del bucle.
Por ejemplo: Hallar el factorial de un número.

```
Algoritmo factorial
  Principal
  var
    entero: cont, num
    real: fact
  inicio
    escribir ("Introduzca un numero: ")
    leer (num)
    fact <- 1
    cont <- 1
    repetir
      fact <- fact * cont
      cont <- cont + 1
    hasta_que (cont > num)
    escribir ("Factorial", fact)
  fin_principal
fin_algoritmo
```

En este ejemplo, el cuerpo del bucle se reiterará hasta que la variable `cont` tenga un valor mayor que la variable `num`. Cuando se cumpla esa condición sigue con la instrucción escribir.

2.4.3. Desde

Esta estructura ejecuta las instrucciones del cuerpo del bucle un número determinado de veces, controlado automáticamente por un contador (normalmente denominado i). Debido a su control automático, esta estructura es ideal para recorrer vectores y matrices (que veremos más adelante). Por ejemplo: Sumar hasta cien.

```
1  Algoritmo suma_cien
2    Principal
3    var
4      entero: suma, i
5    inicio
6      suma <- 0
7      desde i <- 1 hasta 100 hacer
8        suma <- suma + i
9      fin_desde
10     escribir (suma)
11   fin_principal
12 fin_algoritmo
```

En este ejemplo, una vez que se recorre el bucle por completo, el valor de suma es 100, que es escrito mediante la instrucción escribir. En cuanto a la variable `i`, podemos ver que está declarada como entero, pero no se inicializa hasta que no se encuentra en la condición del bucle. Ésta es una característica de la estructura desde que no la tienen las otras estructuras reptetitivas.

2.4.4. Aclaración de las estructuras repetitivas

Debemos señalar ciertos aspectos una vez vistas todas las estructuras repetitivas.
- **Siempre** podemos sustituir una estructura `desde` por una estructura `repetir` o por una estructura `mientras`.
- Una estructura `repetir` o `mientras`, solo podemos sustituirlas por una estructura `desde` si sabemos 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 del bucle el hecho de que en la estructura `mientras`, como mínimo se ejecutarán una vez todas las instrucciones del cuerpo del bucle.

2.5. Estructuras anidadas

Debemos tener en cuenta que tanto las estructuras selectivas como las repetitivas se pueden anidar, es decir, pueden estar contenidas unas dentro de otras. Podemos hacer esto siempre que una esté dentro de la otra y que no se salga.
Lo veremos con el siguiente ejemplo, donde podemos encontrar una estructura selectiva dentro de dos bucles, uno dentro de otro.

```
Algoritmo primos
  Principal
  var
    logico: encontrado
    entero: i, divisor
  inicio
    desde i <- 2 hasta i <- 100 hacer
      encontrado <- falso
      divisor <- 2
      mientras (divisor <= sqrt(i) and encontrado = falso) hacer
        si (i mod divisor = 0) entonces
          encontrado <- verdadero
        fin_si
        divisor <- divisor + 1
      fin_mientras
      si (encontrado = falso) entonces
        escribir (i)
      fin_si
    fin_desde
  fin_principal
fin_algoritmo
```

En este ejemplo donde se escriben todos los número primos entre 2 y 1000 podemos ver como las estructuras se anidan unas dentro de otras pero ninguna interfiere con la otra.

3. Abstracción operacional

Para el diseño de un algoritmo se puede dividir el problema en subproblemas, cada uno de ellos atendido por un subalgoritmo (procedimiento o función).
En todo subalgoritmo se pueden distinguir dos partes fundamentales:
- La especificación: ¿Qué hace el algoritmo?
- La implementación: ¿Cómo lo hace?

La especificación de un subalgoritmo se divide en tres cláusulas:
- **Cabecera:** Proporciona información sintáctica para proceder a la ejecución del subalgoritmo.
- **Precondición:** Condiciones que deben cumplir los valores de entrada.
- **Postcondición:** Efecto producido por el subalgoritmo y condiciones que se cumplen al finalizar el mismo.

3.1. Subalgoritmos

3.1.1. Funciones

Es un subalgoritmo que tiene valores de entrada llamados argumentos y devuelve un único valor denominado resultado.
La declaración de funciones en pseudocódigo sigue la siguiente estructura:

```
<especificacion de la funcion>
<tipo del resultado>funcion<nombre funcion>(<lista de parametros formales>
[declaraciones locales]
  inicio
  <instrucciones> //cuerpo
  devolver (<expresion resultado>)
fin_funcion
```

<lista_de_parametros >: Será la lista de parámetros formales de la siguiente forma:
(E|S|E/S tipoA: parámetro1, E|S|E/S tipoB: parámetro2,...
Siendo, E: Entrada, S: Salida, E/S: Entrada y Salida.
<nombre_función>: Identificador valido para la función.
<instrucciones>: instrucciones que constituyen el cuerpo de la función, deberán contener una única expresión devolver (<expresión>).
<tipo_de_resultado>:
tipo de dato correspondiente al resultado de la función.

Para llamar a la función sería de la siguiente forma:
`nombre_función (lista de parámetros actuales)`

Por ejemplo: Cálculo del cuadrado de un número.

```
Algoritmo elevar_cuadrado
  Principal
  var
    real : numero, res
  inicio
    escribir ("Introduzca un numero:")
    leer (numero)
    res <- cuadrado(numero)
    escribir("El cuadrado de", numero, "es", res)
  fin_principal
fin_algoritmo

//Cabecera: real cuadrado(E real: x)
//Precondicion: recibe x real
//Postcondicion: devuelve el cuadrado de x (x2)
real funcion cuadrado(E real: x)
  inicio
    devolver (x*x)
fin_funcion
```

En este ejemplo se llama a la función `cuadrado` que calcula y devuelve el cuadrado de un número.

3.1.2. Procedimientos

Es un subalgoritmo que solo tiene valores de salida y no devuelve nada.
La declaración de procedimientos en pseudocódigo sigue la siguiente estructura:

```
<especificacion del procedimiento>
procedimiento<nombre_procedimento>(<lista de parametros formales>)
[declaraciones locales]
  inicio
    <instrucciones> //cuerpo
fin_procedimiento
```

<lista_de_parámetros>: Será la lista de parámetros formales de la siguiente forma:
(E|S|E/S tipoA: parámetro1,E|S|E/S tipoB: parámetro2).
Siendo, E: Entrada, S: Salida, E/S: Entrada y Salida.
<nombre_procedimento>: Identificador válido para el procedimiento.
<instrucciones>: instrucciones que constituyen el cuerpo del procedimiento (observamos que no existe una sentencia devolver como sucedía en las funciones).

Para llamar a la función sería de la siguiente forma:
`nombre_función (lista de parámetros actuales)`

Por ejemplo: Calculo de la división de dos números enteros calculando cociente y resto.

```
Algoritmo divide
  Principal
  var
    entero: divdo, divi
  inicio
    repetir
      escribir("Introduce␣los␣valores␣del␣dividendo␣y␣del␣divisor:␣")
      leer(divdo, divi)
    hasta_que (divdo >= divi y divi>0 y divdo>0)
    division(divdo, divi)
  fin_principal
fin_algoritmo

//Cabecera: division(E entero: dividendo,E entero: divisor)
//Precondicion: dividendo > = divisor y ambos mayores que 0
//Postcondicion: escribe en pantalla el cociente y el resto
procedimiento division (E entero: dividendo, E entero : divisor)
  var
    entero: cociente, resto
  inicio
    cociente <- dividendo / divisor
    resto <- dividendo mod divisor
    escribir (cociente, resto)
fin_procedimiento
```

En este ejemplo se llama al procedimiento `división` que calcula el cociente y el resto de la división de dos números.

3.2. Ámbito y persistencia de variables

El ámbito es el fragmento de código en el cual una variable puede ser referenciada. Podemos distinguir dos ámbitos:

3.2.1. Ámbito local

Son las variables declaradas dentro del `Principal` o dentro de los subalgoritmos como `procedimientos` o `funciones`. Pueden estar declaradas en la sección de definición de variables (`var`) o como parámetro formal y pueden ser referenciadas dentro de ese subalgoritmo.
El uso de variables locales conserva la independencia de los subalgoritmos y hace un uso más eficiente de la memoria.

3.2.2. Ámbito global

Son las variables declaradas fuera del principal y de cualquier subalgoritmo y puede ser referenciada desde cualquier punto del cuerpo del `Principal` o subalgoritmo.
Por ejemplo:

```
1  Algoritmo ejemplo
2    var
3      entero: a, b
4    Principal
5    inicio
6      a <- 2
7      b <- 3
8      proc1(a)
9    Fin_principal
10 fin_algoritmo
11
12 procedimiento proc1(E entero: x)
13   var
14     entero: b
15   inicio
16     b <- x+a
17     escribir(b)
18 fin_procedimiento
```

En este ejemplo, `a` y `b` están definidas fuera del `Principal`, por lo que son variables de ámbito global, a su vez, `a` es el parámetro real (o actual) de la llamada al procedimiento `proc1`, concretamente, con el valor real 2. La variable `x` es un parámetro formal y a su vez, variable local de `proc1`.
`b` es otra variable local de `proc1`, distinta de la `b` global. En la instrucción `b <- x+a` estamos asignando a la variable local `b` el resultado de sumar el valor de la variable local `x` con el valor de la variable global `a`. Debemos tener en cuenta que tanto `Principal` como el procedimiento `proc1` podrían modificar el valor de la vriable global `a`.

3.2.3. Persistencia de variables

La persistencia de una variable es la duración de la variable en memoria. Podemos distinguir dos tipos:
- **Persistendia indefinida:** Permanece en memoria durante toda la ejecución de un programa.
- **Persistencia dinámica:** Cuando la variable solo existe durante la ejecución de un fragmento de código (función o procedimiento).
Generalmente suele asociarse una persistencia indefinida al ámbito global y una persistencia dinámica al ámbito local.
En algunos lenguajes de programación es posible modificar el ámbito y la persistencia de las variables a través de determinadas instrucciones siendo posible que existan las cuatro combinaciones posibles: ámbito global y persistencia indefinida, ámbito global y persistencia dinámica, ámbito local y persistencia indefinida, y, ámbito local y persistencia dinámica.

3.3. Correspondencia entre argumento y parámetro formal

El paso de parámetros es el método mediante el cual los distintos subalgoritmos de un algoritmo pueden comunicarse, compartir valores y variables sin perder su independencia.
Existe una correspondencia automática entre los parámetros formales y los parámetros actuales que puede ser de dos tipos:
- **Posicional:** De izquierda a derecha entre parámetros formales ya actuales.
- **Por nombre:** Se indica de forma explícita en la llamada al subalgoritmo cómo ha de realizarse la correspondencia. Por ejemplo: `porc(a=>x, b=>3)`
El tipo de paso de parámetros puede ser por valor o por referencia:

3.3.1. Paso por valor

En el paso por valor se produce una copia del valor de los parámetros acutales (constantes, variables o expresiones) en los formales (variables locales al procedimiento inicializadas con el valor que el parámetro actual les pasa en la llamada).
Los parámetros pasados por valor serán siempre parámetros de entrada y llevarán la letra `E` en su definición. A través de estos parámetros no se devuelve ningún tipo de información al algoritmo desde el cual se realizó la llamada.
Cuando el subalgoritmo devuelve el control al algoritmo que realizó la llamada, los parámetros actuales conservarán el valor que tenían antes de realizar la llamada.

3.3.2. Paso por referencia

En el paso por referencia, el parámetro formal recibe una referencia (dirección de memoria) del parámetro actual, lo que obliga a que dichos parámetros sean siempre variables.
Como tanto los parámetros formales como los actuales comparten la misma dirección de memoria, todo cambio realizado en el parámetro formal tendrá repercusión en el parámetro actual.
Estos parámetros siempre serán de salida o entrada/salida y llevarán en su definición las letras `S` o `E/S`.
Gracias a este paso de parámetros se consigue que el subalgoritmo devuelva valores al algoritmo que realizó la llamada.
En el siguiente ejemplo podemos ver dos pasos por valor y dos pasos por referencia.

```
Algoritmo divide
  Principal
  var
    entero: dividendo, divisor, cociente, resto
  inicio
    escribir("Introduce los valores del dividendo y divisor: ")
    repetir
      leer (dividendo, divisor)
    hasta que (dividendo >= divisor)
    division (dividendo, divisor, cociente, resto)
    escribir (cociente, resto)
  fin_principal
fin_algoritmo

//Cabecera: division(E entero: divi,E entero: d, S entero: c,
            S entero: r)
//Precondicion: divi >= divi y ambos mayores que 0
//Postcondicion: devuelve a traves de los parametros de salida
        c y r, el cociente y el resto de dividir divi entre d
procedimiento division (E entero: divi, E entero: d, S entero: c,
                        S entero: r)
  Inicio
    c <- divi div d
    r <- divi mod d
fin_procedimiento
```

En este ejemplo podemos ver como los parámetros formales `divi` y `d` son pasados por valor y parámetros formales `c` y `r` son pasados por referencia al procedimiento `division` que devuelve al programa principal el valor del cociente y del resto tras la ejecución del mismo.

3.3.3. Efectos laterales

Los efectos laterales son los cambios o modificaciones que se producen en las variables de un subalgoritmo como consecuencia de la instrucción de otro subalgoritmo.
Si la comunicación entre un subalgoritmo y otro no se realiza mediante el paso de parámetros, se pueden provocar efectos laterales como cambiar el contenido de una variable global.
Si se necesita una variable local en un subalgoritmo se utiliza localmente, nunca una variable global, salvo que se quiera que se modifique el valor de la misma y deberá ser pasado por referencia (`E/S` o `S`).
Dichos efectos laterales están considerados como una mala técnica de programación debido a que disminuye la legibilidad del código y dificulta la depuración y detección de errores.

4. Tipos de datos

4.1. Tipos de datos estructurados

Un tipo de dato estructurado o una estructura de datos es un conjunto de datos que se trata como una sola unidad pero que a su vez permite referenciar independientemente cada uno de sus componentes.
La mayoría de los lenguajes de programación permiten al programador definir estructuras de daos cuyos elementos son a su vez tipos de datos estructurados.
La elección del tipo de dato adecuado dependerá de la complejidad del problema a resolver y del lenguaje de programación utilizado para su implementación y los definiremos en la sección `tipo`.
Las estructuras de datos se clasifican en dos tipos:

4.1.1. Momento de reserva de memoria

- **Estáticas:** El espacio de memoria ocupado por la estructura es fijo y se define en tiempo de compilación por lo que no puede ser modificado mediante la ejecución del programa.
- **Dinámicas:** El espacio de memoria ocupado por la estructura es dinámico, se define en tiempo de ejecución y además puede ser modificado durante la ejecución del programa en función de las necesidades del mismo.

4.1.2. Tipos de datos que forman la estructura

- **Homogéneas:** Estructuras que están compuestas por componentes del mismo tipo. Por ejemplos, vectores y matrices.
- **Heterogéneas:** Estructuras que están compuestas por elementos de distinto tipo. Por ejemplo, registros.

4.2. Vectores y matrices

Los vectores y matrices son estructuras de datos homogéneas y puede ser estático o dinámico según convenga para la resolución del problema.

4.2.1. Vectores. Matrices unidimensionales

Es un conjunto finito del mismo tipo que están relacionados consecutivamente y pueden ser identificados de forma independiente.
Es importante tener en cuenta que los vectores en pseudocódigo empiezan por la posición 1, sin embargo en los lenguajes de programación como C, empiezan por la posición 0.
La definición de un vector es la siguiente:

```
1 Tipo
2   vector [tamano] de <tipo de dato> : <identificador_del_tipo_vector>
```

En cuanto a las operaciones que podemos realizar con un vector tenemos:
- **Asignación:** `V[2]<-5`
En algunos lenguajes tenemos la opción de realizar asignaciones entre vectores.

- **Recorrido de un vector:** Se recorre el vector posición a posición con la finalidad de introducir o leer datos de él.
Un ejemplo de cómo recorrer un vector para introducir datos en él es el siguiente:

```
desde i <- 1 hasta 7 hacer
  escribir ("Inserte el elemento", i)
  leer (V[i])
fin_desde
```

Y un ejemplo para leer los datos del vector y mostrarlos por pantalla es el siguiente:

```
desde i <- 1 hasta 7 hacer
  escribir (V[i])
fin_desde
```

Cabe mencionar que en en estos dos ejemplos, solo hemos visualizado el bucle que lo recorre sin mostrar el algoritmo completo con la única razón de darle claridad.

Por último veamos un ejemplo del trabajo con vectores:
Diseña un algoritmo que almacena en un vector de 20 elementos los valores correspondientes a las notas de 20 alumnos de una clase. Además el algoritmo debe calcular la nota media y escribirla por pantalla.

```
Algoritmo media_vector
  const
    MAX = 20
  tipo
    vector [MAX] de real : Nota
  Principal
  var
    Nota: V
    real: media
  inicio
    leer_vector(V)
    media <- calcula_media(V)
    escribir ("La media es", media)
  fin_principal
Fin_algoritmo

//Cabecera: leer_vector(S Nota: V).
//Precondicion: V es una variable de tipo Nota.
//Postcondicion: inicializa V con los datos introducidos por el usuario.
procedimiento leer_vector(S Nota: V)
  var
    entero: i
  inicio
    desde i <- 1 hasta MAX hacer
      escribir("Introduzca el elemento: ", i)
      leer(V[i])
    fin_desde
```

```
fin_procedimiento

//Cabecera: real calcula_media(E Nota: V)
//Precondicion: V es una variable de tipo Nota que debe estar inicializada
//Postcondicion: devuelve la media de todos los elementos del vector.
real funcion calcula_media(E Nota: V)
  var
    real: suma, media
  inicio
    suma <- 0
    desde i <- 1 hasta MAX hacer
      suma <- suma + V[i]
    fin_desde
    media <- suma/MAX
    devolver media
fin_funcion
```

En este ejemplo podemos ver como el algoritmo principal tiene dos subalgoritmos para resolver el problema planteado en los cuales se trabaja con los elementos de un vector llamado `Nota`.

4.2.2. Matrices multidimensionales

Una matriz multidimensional es una estructura homogénea, en la cual para hacer referencia a un elemento necesitamos dos o más índices, dependiendo de su dirección. Lo más frecuente es usar matrices bidimensionales (también llamadas tablas por su símil a la hora de establecer dos índices: fila y columna).
Básicamente, una matriz multidimensional se puede entender como un vector de vectores.
La definición de una matriz multidimensional es la siguiente:

```
tipo
  matriz [tam_1, tam_2, ..., tam_n] de <tipo de dato>:
                          <identificador_del_tipo_matriz>
```

En cuanto a las operaciones que podemos realizar con una matriz multidimensional tenemos:
- **Asignación:** `M[2][3]<-8`
- **Recorrido de una matriz:** Se recorre la matriz posición a posición con la finalidad de introducir o leer datos de ella.

Un ejemplo de cómo recorrer una matriz para introducir datos en ella es la siguiente:

```
desde i <- 1 hasta 3 hacer
  desde j <- 1 hasta 10 hacer
    M[i][j] <- 0
  fin_desde
fin_desde
```

En este ejemplo, al igual que con los vectores, solo hemos visualizado el bucle que recorre la matriz sin mostrar el algoritmo completo con la única razón de aportar mayor claridad.

Por último veamos un ejemplo del trabajo con matrices multidimensionales:
Diseña un algoritmo que suma dos matrices de dimensión `fxc`, siendo `f` y `c` introducidos por teclado y menores o iguales que 100.

```
Algoritmo suma_matrices
  const
    MAX = 100
  tipo
    matriz [MAX,MAX] de entero: Tabla
  Principal
  var
    entero: i,j,f,c
    Tabla: A,B,S
  inicio
    escribir("Introduzca el numero de filas y columnas de la matriz: ")
    leer(f,c)
    leer_matriz(A,f,c)
    leer_matriz(B,f,c)
    calcula_suma(A,B,f,c,S);
    escribir_matriz(S)
  fin_principal
Fin_algoritmo

//Cabecera: leer_matriz(S Tabla: M, E entero: f, E entero: c).
//Precondicion: M es una variable de tipo Tabla, f y c son dos
              //variables enteras que deben estar inicializadas.
//Postcondicion: M se inicializa a los valores introducidos
                  //por el usuario.
procedimiento leer_matriz (S Tabla: M, E entero: f, E entero: c )
  var
    entero i, j
  inicio
    desde i <- 1 hasta f hacer
      desde j <- 1 hasta c hacer
        escribir("Introduzca el elemento de la posicion ", i, j)
        leer(M[i][j])
      fin_desde
    fin_desde
fin_procedimiento

//Cabecera: calcula_matriz(E Tabla: A, E Tabla: B, E entero: fil,
                                        E entero: col, S Tabla: C).
//Precondicion: A, B y C son matrices de tipo Tabla, fil y col
       //son dos variables enteras que deben estar inicializadas.
//Postcondicion: C contiene la matriz suma de A y B.
procedimiento calcula_suma(E Tabla: A, E Tabla: B, E entero: fil,
                                         E entero: col, S Tabla: C)
  var
    entero i, j
  inicio
    desde i <- 1 hasta fil hacer
```

```
48        desde j <- 1 hasta col hacer
49          C[i][j] <- A[i][j]+B[i][j]
50        fin_desde
51      fin_desde
52  fin_procedimiento
53
54  //Cabecera: escribir_matriz(E Tabla: M, E entero: f, E entero: c).
55  //Precondicion: M es una variable de tipo Tabla, f y c son dos
56  //variables enteras. Todas las variables deben estar inicializadas.
57  //Postcondicion: Se visualiza el contenido de la matriz.
58  procedimiento escribir_matriz (E tabla: M, E entero: f, E entero: c )
59    var
60      entero i, j
61    inicio
62      desde i <- 1 hasta f hacer
63        desde j <- 1 hasta c hacer
64          escribir(M[i][j])
65        fin_desde
66      fin_desde
67  fin_procedimiento
```

En este ejemplo podemos ver como el algoritmo principal tiene tres subalgortimos para resolver el problema planteado en los cuales se trabaja con los elementos de una matriz llamada `Tabla`.

4.3. Cadenas de caracteres

Una cadena de caracteres es una secuencia de caracteres consecutivos. Existen lenguajes de programación que incorporan el tipo cadena como uno de sus tipo básicos (`string`).
Otros lenguajes no incorporan esta posibilidad y es el programador quien debe definir y utilizar las cadenas de caracteres como vectores de tipo carácter y la longitud de la cadena (vector) será el número de caracteres que contiene.
La definición de una cadena de caracteres se realiza en la sección `var` así `cadena: cad`.

4.3.1. Operaciones con cadenas de caracteres

Las operaciones más comunes en las cadenas de caracteres son:
- **Asignación:** Podemos asignar una variable de tipo cadena a una constante de cadena:
`cad <- "Hola a todos."`.
- **Lectura/escritura:** Podemos utilizar las operaciones de lectura y escritura con cadenas de caracteres: `leer(cad)` o `escribir(cad)`.
- **Cálculo de longitud:**Función que devuelve el número de caracteres que contiene la cadena:
`n <- longitud(cad)`.
- **Comparación:** Realiza comparaciones entre caracteres. Para ello utiliza el código ASCII y devuelve: un número negativo si la cadena1 es mayor que la cadena2; un número positivo si la cadena1 es mayor que la cadena2; o cero, si la cadena1 es igual que la cadena2:
`n <- compara(cad1,cad2)`
- **Concatenación:** Consiste en unir varias cadenas en una: `cad3 <- concatena(cad1,cad2)`.

4.4. Registros

Un registro es una estructura de datos formada por un conjunto de elementos que van a contener información relativa a un mismo objeto. Los elementos que constituyen un registro se llaman campos y cada campo puede ser de un tipo diferente al resto.
En un registro también se pueden introducir tipos creados por el propio usuario.
La definición de un registro se realiza en la sección `tipo` y es la siguiente:

```
tipo
  registro: nombre_reg
    tipo 1: id_campo 1
    tipo 2: id_campo 2
    . . .
    tipo n: id_campo n
  fin_registro
```

Por ejemplo: para representar la información relativa a un estudiante matriculado en una asignatura concreta, se puede emplear un registro con los campos: nombre, apellidos, edad, DNI y nota.

```
tipo
  registro: reg_estudiante
    cadena: apellidos, nombre
    entero: edad
    entero: dni
    entero: nota
  fin_registro
  Principal
  var
    reg_estudiante: estudiante
```

Una vez creado el tipo `reg_estudiante` y definido como `estudiante`, cada vez que nos refiramos a ese registro lo haremos con `estudiante`. Para seleccionar el campo *identificador_campo* del registro *nombre_reg* se utiliza el formato *nombre_reg.identificador_campo*.

Siempre que tengamos dos registros del mismo tipo se puede realizar la operación de asignación.
Cada campo de un registro puede usarse en cualquier tipo de expresión, ser asignado, pasado como argumento en llamadas a subalgoritmos o ser devuelto como resultado de una función.
También debemos saber que un registro también puede contener dentro de sí a otros registros y vectores o matrices.
Por ejemplo: si queremos conocer el mes de la fecha de nacimiento del estudiante `i`, el acceso al registro en el cuerpo del algoritmo sería `estudiantes[i].nacimiento.mes`.

```
const
  N=100
tipo
  registro: reg_fecha
    entero: dia
    cadena: mes
    entero: ano
  fin_registro
```

```
  registro: reg_estudiante
    cadena: apellidos, nombre
    entero: edad
    entero: dni
    entero: nota
    reg_fecha: nacimiento
  fin_registro
  vector [N] de reg_estudiante : Testudiantes
Principal
var
  TEstudiantes: estudiantes
```

4.5. Ficheros

Los ficheros son una estructura de datos no acotada y almacenada en memoria masiva o externa. Aporta la ventaja de ser permanente y que la mayor limitación que tiene es la del soporte donde se encuentra.
Para trabajar con un fichero almacenado en memoria masiva, definiremos una variable de tipo fichero en la memoria principal a la que le asociaremos dicho fichero mediante la operación de apertura del mismo.
La declaración de un fichero se realiza en las secciones `tipo` y `var` de la siguiente forma:

```
tipo
  archivo de tipo: tipo_fich
var
  tipo_fich: id_var
```

Aquí lo podemos ver en un ejemplo:

```
tipo
  archivo de entero: tipo_fich
Principal
var
  tipo_fich: fich_notas_act, fich_notas_ant
```

En este ejemplo se ha definido un tipo fichero que va a corresponder a ficheros cuyos elementos son enteros, y en la sección `var` se han declarado dos variables, que serán posteriormente asociadas a ficheros mediante la función de apertura.

4.5.1. Apertura de un fichero

La función `abrir(var_fichero)` asocia una variable de tipo fichero con un archivo almacenado en memoria secundaria. El formato de la operación de apertura es el siguiente:
`abrir(var_fichero, modo, nombre_fichero)`
- **nombre_fichero:** Es el nombre del fichero que se encuentra almacenado en memoria masiva.
- **var_fichero:** Es la variable definida previamente a la que se le asociará el fichero físico. A partir de este momento cualquier referencia a lfichero se hará a través de esta variable.
- **modo:** Indica el modo de acceso al fichero, es decir, si la operación que se desea realizar es de lectura, escritura, o lectura/escritura.

Si el fichero es abierto en modo escritura, el sistema comprueba si existe un fichero con ese nombre situando el indicador de posición al inicio del fichero y cada vez que se realice una operación de escritura, el indicador irá avanzando a la siguiente posición. Si no existe, entonces, se crea uno vacío con ese nombre. Si existe, lo borra y crea uno nuevo.
Si el fichero es abierto en modo lectura o lectura/escritura, el indicador de posición se situará al inicio del mismo y avanzará a la posición siguiente cada vez que se realice una operación de lectura. Si el fichero solo se abre en modo lectura no podrá ser modificado.
La función `feof(var_vichero)` comprueba si se ha llegado al final del fichero y la función `cerrar(var_fichero)` cierra y guarda el fichero.

Veamos un ejemplo del trabajo con ficheros: Diseña un algoritmo que lea caracteres y los escriba en un fichero. Posteriormente lea del fichero para escribir en pantalla.

```
1  Algoritmo lee_escribe_fichero
2    tipo
3      archivo de caracteres: tipo_fich
4    Principal
5    var
6      tipo_fich: fichero
7    inicio
8      lectura (fichero)
9      escritura(fichero)
10   Fin_principal
11 fin_algoritmo
12
13 //Cabecera: lectura(E/S tipo_fich f)
14 //Precondicion:
15 //Postcondicion: escribe en f los caracteres introducidos por el usuario.
16 Procedimiento lectura (E/S tipo_fich f)
17   var
18     caracter: c
19   inicio
20     abrir (f, "escritura", "fichero_caracteres.txt")
21     escribir("Introduzca␣los␣caracteres.␣Caracter␣'*'␣para␣terminar")
22     mientras (leer(c) != '*') hacer
23       escribir(f,c)
24     fin_mientras
25     cerrar(f)
26 fin_procedimiento
27
28 //Cabecera: escritura (E tipo_fich f)
29 //Precondicion:
30 //Postcondicion: lee todo el contenido de f y lo escribe en pantalla.
31 Procedimiento escritura (E tipo_fich f)
32   var
33     caracter : c
34   inicio
```

```
35      abrir (f, "lectura", "fichero_caracteres.txt")
36      mientras (no feof(f)) hacer
37        leer(f,c)
38        escribir(c)
39      fin_mientras
40      cerrar(f)
41  fin_procedimiento
```

En este ejemplo podemos ver como el algoritmo principal tiene dos subalgoritmos para resolver el problema que coinciden con la escritura y la lectura de datos del fichero.