Departamento de Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

Apuntes para la asignatura

Informática

Facultad de Ciencias (Matemáticas)

http://www.lcc.uma.es/personal/pepeg/mates

Tema 5. Estructuras iterativas

Bibliografía

- *Programación 1*. José A. Cerrada y Manuel Collado. Universidad Nacional de Educación a Distancia.
- *Programming with TopSpeed Modula-2*. Barry Cornelius. Addison-Wesley.
- *Fundamentos de programación*. L. Joyanes. McGraw-Hill.

Introducción

Las estructuras de selección, estudiadas en el tema anterior, no son suficientes para describir cualquier algoritmo. Es habitual que ciertas partes de un algoritmo deban repetirse varias veces con objeto de resolver un problema. La *repetición* es un concepto importante a la hora de describir algoritmos. Los lenguajes de programación disponen de una serie de sentencias que permiten repetir varias veces algunos segmentos del programa. A este tipo de sentencias se les denomina *sentencias iterativas*. En este tema, estudiamos las sentencias iterativas que posee el lenguaje de programación *Modula-2*.

5.1 Bucles

Un término muy utilizado en el argot informático es *bucle*. Un bucle es un segmento de algoritmo que se repite varias veces. Podemos distinguir dos tipos de bucles:

- *Bucles deterministas*: son aquellos para los cuales el número de repeticiones es conocido a priori.

  ❏ Ejemplo: Supongamos que tenemos un ordenador que sólo es capaz de sumar una unidad a un número. Escribir un algoritmo que lea un número del teclado, le sume diez y muestre el resultado por pantalla.

  Un posible algoritmo que utiliza un bucle determinista para resolver este problema es el siguiente:

  1. Leer un número desde el teclado
  2. Repetir 10 veces

     2.1. Sumar uno al valor del número
  3. Escribir el valor del número

  ❏

- *Bucles no deterministas o indeterministas*: son aquellos para los cuales no se conoce a priori el número de veces que se van a repetir. El bucle se repite hasta que se alcanza cierta condición. La condición es conocida a priori, pero no sabemos cuántas veces es necesario repetir el bucle para que la condición se alcance.

  ❏ Ejemplo: Escribir un algoritmo que lea números desde teclado hasta que se lea un valor cero. El algoritmo debe calcular la suma total de los números leídos y mostrar el resultado por pantalla.

  Un algoritmo que utiliza un bucle no determinista para resolver este problema es el siguiente:

  1. Guardar el valor cero en la variable suma
  2. Leer un número desde el teclado
  3. Mientras que el valor del número leído sea distinto de cero

     3.1. Sumar el valor del número leído a la variable suma

     3.2. Volver a leer el número desde el teclado
  4. Escribir el valor de la variable suma

En este caso, no sabemos a priori cuántas veces se repetirán los pasos 3.1 y 3.2. Sin embargo, sabemos que dichos pasos dejarán de repetirse en el momento en que se lea un valor cero desde el teclado. Conocemos a priori la condición pero no el número de repeticiones.

❑

5.2 Estructuras iterativas en Modula-2

Modula-2 dispone de cuatro sentencias iterativas distintas. Pasemos a estudiar cada una de ellas.

5.2.1 La sentencia FOR

La sentencia FOR permite repetir, un número de veces conocido a priori, una serie de instrucciones. La sentencia FOR es el modo más adecuado de expresar bucles definidos en *Modula-2*.

La forma más simple que toma esta sentencia es la siguiente:

```
FOR Variable := Valor1 TO Valor2 DO
  Acciones
END
```

Las palabras FOR, TO, DO y END son palabras reservadas del lenguaje y delimitan cada una de las partes de la sentencia. La variable que aparece a la izquierda del signo de asignación se llama *variable de control* del bucle. El valor de esta variable se modifica en cada repetición del bucle. El conjunto de acciones que aparecen entre las palabras DO y END constituyen el *cuerpo del bucle*.

El significado de la sentencia FOR es el siguiente: se ejecutan las acciones entre las palabras DO y END tantas veces como indique la expresión (`Valor2-Valor1+1`). La primera vez que se ejecuten estas acciones, el valor de la variable de control es `Valor1`. La segunda vez que se ejecuten las acciones, el valor de la variable de control es `Valor1+1`. El valor de la variable de control es incrementado en una unidad tras cada repetición, de modo que la última vez que se ejecuten las acciones el valor de la variable de control es `Valor2`.

❑ Ejemplo: Escribir un programa en *Modula-2* que lea un número `Num` del teclado y calcule el sumatorio de los números naturales menores o iguales a `Num ( 1+2+3+ ... + (Num-1)+Num )`. El programa debe mostrar el resultado del sumatorio por pantalla.

Supongamos que no conocemos la siguiente equivalencia:

$$\sum_{i=1}^{N} i = \frac{N(N-1)}{2}$$

Para calcular el sumatorio anterior podemos utilizar una variable `Suma` cuyo valor inicial sea cero. Primero, sumaremos a esta variable el valor uno. A continuación, el valor dos. Repetiremos este paso hasta sumar el valor `Num`. El algoritmo que calcula el sumatorio es:

```
Suma := 0;
Suma := Suma + 1;
Suma := Suma + 2;
...
Suma := Suma + (Num-1);
Suma := Suma + Num;
```

o de un modo más breve:

1. Asignar a la variable Suma el valor cero.
2. Repetir Num veces (variando el valor de i desde uno hasta Num)
   2.1. Sumar i al valor de la variable Suma

El modo más adecuado de expresar el paso 2 de este algoritmo en *Modula-2* es usando la sentencia FOR. Se obtiene el siguiente programa:

```
MODULE Suma;
FROM IO IMPORT RdCard, RdLn, WrCard, WrStr;
VAR
  Num, Suma, i :CARDINAL;
BEGIN
  WrStr("Dame un número: ");
  Num := RdCard(); RdLn;

  Suma := 0;
  FOR i := 1 TO Num DO
    Suma := Suma + i
  END;

  WrStr("La suma de los primeros ");
  WrCard(Num,0);
  WrStr(" números naturales es ");
  WrCard(Suma,0)
END Suma.
```

❑

Podemos especificar que la variable de control se incremente en un valor distinto a uno tras cada repetición de las acciones dentro del bucle. Para ello se utiliza la palabra reservada BY seguida del valor a sumar a la variable de control.

❑ Ejemplo: Escribir un programa en *Modula-2* que lea del teclado un número impar Num y calcule el sumatorio de los números naturales impares menores o iguales a Num (1+3+ ... + (Num-2)+Num).

Para calcular el sumatorio anterior podemos utilizar una variable Suma cuyo valor inicial sea cero. Sumaremos primero a esta variable el valor uno. A continuación el valor tres. Repetiremos este paso hasta sumar el valor Num. El algoritmo que calcula el sumatorio es:

1. Asignar a la variable Suma el valor cero.
2. Repetir (variando el valor de i desde uno hasta Num de dos en dos)
   2.1. Sumar i al valor de la variable Suma

Se obtiene el siguiente programa:

```
MODULE Suma;
FROM IO IMPORT RdCard, RdLn, WrCard, WrStr;
VAR
  Num, Suma, i :CARDINAL;
BEGIN
  WrStr("Dame un número: ");
  Num := RdCard(); RdLn;

  IF NOT ODD(Num) THEN
    WrStr("El número no es impar")
  ELSE
    Suma := 0;
    FOR i := 1 TO Num BY 2 DO
      Suma := Suma + i
    END;
```

```
      WrStr("La suma de los primeros ");
      WrCard(Num,0);
      WrStr(" números impares naturales es ");
      WrCard(Suma, 0)
   END
END Suma.
```

❑

El valor que aparece tras la palabra reservada BY puede ser también negativo. En dicho caso, la variable de control del bucle es decrementada en el valor indicado tras cada repetición del bucle.

La sintaxis BNF para la sentencia FOR es la siguiente:

Sentencia_FOR	**::=**	FOR Identificador_de_Variable := Expresión_Inicial TO Expresion_Final **[** BY Paso **]** DO Secuencia_de_Sentencias END
Expresión_Inicial	**::=**	Expresión
Expresión_Final	**::=**	Expresión
Paso	**::=**	Expresión_Constante

Obsérvese que la expresión que aparece tras la palabra BY debe ser una expresión constante (no puede contener ni variables ni operadores no predefinidos).

El diagrama de flujo que representa la sentencia FOR depende del signo de la expresión constante `Paso`:

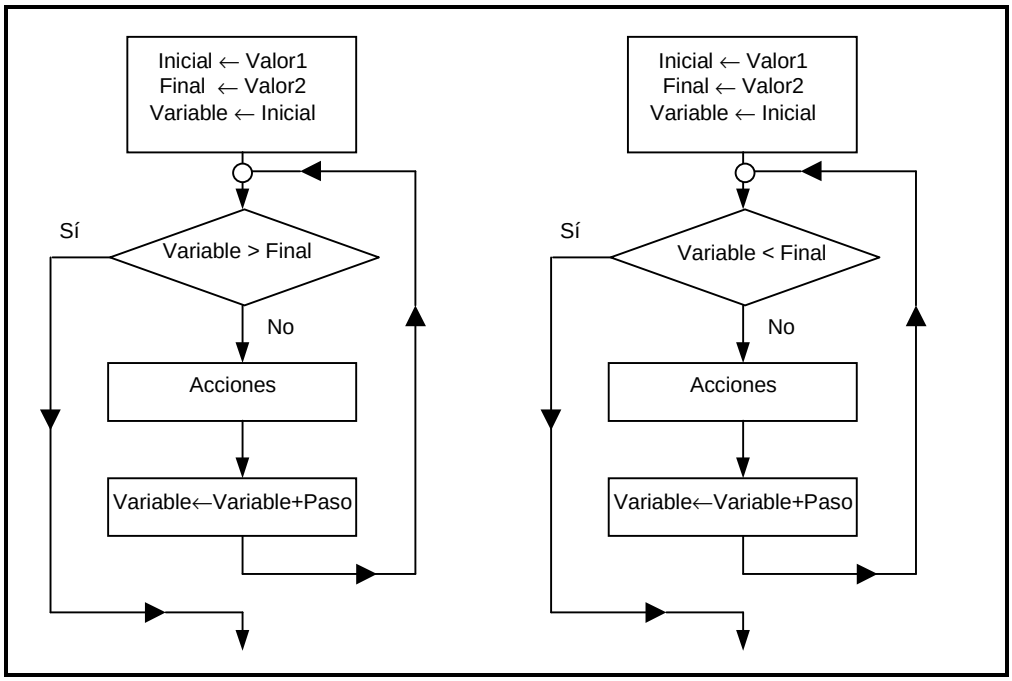


Figura 1. Bucle FOR con paso positivo. **Figura 2**. Bucle FOR con paso negativo.

La condición de salida del bucle es que el valor de la variable de control sea estrictamente mayor (o menor en el caso de que el `Paso` sea negativo) que el valor de la expresión que aparece tras la palabra TO. Adviértase que dicha expresión se calcula *una sola vez*, antes de comprobar la condición del bucle por primera vez. En cada repetición del bucle, la condición es

comprobada antes de ejecutar el cuerpo del bucle. La variable de control es actualizada tras ejecutar el cuerpo del bucle, antes de volver a comprobar la condición.

❏ Ejemplo: El cuerpo del siguiente bucle se ejecuta cero veces, ya que la condición de salida es cierta la primera vez.

```
FOR i := 1 TO 0 DO
  WrCard(i,0); WrLn
END
```

❏

❏ Ejemplo: El cuerpo del siguiente bucle se ejecuta dos veces (con valores para la variable `i` de uno y tres) ya que la condición de salida es cierta la tercera vez.

```
FOR i := 1 TO 4 BY 2 DO
  WrCard(i,0); WrLn
END
```

❏

❏ Ejemplo: El cuerpo del siguiente bucle se ejecuta diez veces (con valores para la variable `i` entre uno y diez) ya que el valor para la expresión que aparece tras la palabra TO es calculado una única vez y no se vuelve a calcular tras cada repetición.

```
Fin := 5;
FOR i := 1 TO 2*Fin DO
  WrCard(i,0); WrLn;
  Fin := 15
END
```

❏

Algunas consideraciones sobre la sentencia FOR son las siguientes:

- El tipo de la variable de control puede ser básico (excepto REAL y LONGREAL), enumerado o subrango (ver tema sobre tipos definidos por el programador).

  ❏ Ejemplo: El siguiente bucle muestra por pantalla todas las letras mayúsculas (excepto la letra eñe mayúscula que está fuera del rango [‘A’..’Z’] en la tabla ASCII).

  ```
  FOR Letra := ‘A’ TO ‘Z’ DO
    WrChar(Letra)
  END
  ```

  ❏

- Los tipos de las expresiones `Valor1` y `Valor2` deben ser compatibles con el tipo de la variable de control.
- El tipo de la expresión constante `Paso` debe ser entero.
- La variable de control no puede ser:
  1) Una componente de una variable estructurada (ver tema sobre *arrays*).
  2) Una variable anónima (ver tema sobre *punteros*).
  3) Una variable importada de otro módulo (ver tema sobre *módulos*).
  4) Un parámetro formal (ver tema sobre *subprogramas*).
- El programador no debe escribir en el cuerpo del bucle ninguna sentencia que modifique el valor de la variable de control.

- El valor de la variable de control está indefinido tras la finalización del bucle FOR.

  ❏ Ejemplo: No sabemos qué valor se escribe por pantalla tras del siguiente bucle.

```
FOR i := 1 TO 10 DO
  ...
END;
WrInt(i, 0);
```

❏

5.2.2 La sentencia WHILE

La sentencia WHILE permite repetir, mientras que sea cierta una condición, una serie de instrucciones. La sentencia WHILE debe utilizarse exclusivamente para expresar bucles indefinidos.

La forma que toma esta sentencia es la siguiente:

```
WHILE Condición DO
  Acciones
END
```

Las palabras WHILE, DO y END son palabras reservadas del lenguaje y delimitan cada una de las partes de la sentencia. La condición que aparece entre las palabras WHILE y DO es denominada *condición de permanencia* del bucle. La condición de permanencia debe ser una expresión de tipo booleano. El conjunto de acciones que aparecen entre las palabras DO y END constituyen el *cuerpo del bucle*.

El diagrama de flujo correspondiente a esta sentencia es el siguiente:

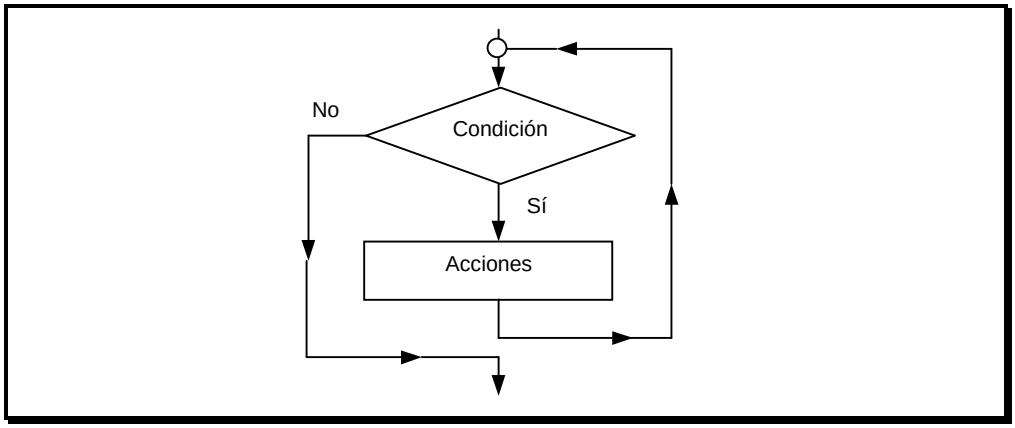


Figura 3. La sentencia WHILE.

Obsérvese que la condición se vuelve a evaluar, para cada repetición del bucle, antes de ejecutar el cuerpo. Se permanece dentro del bucle mientras que la condición sea cierta (es por esto que la condición se llama de permanencia). Podría ocurrir que el cuerpo del bucle no llegara a ejecutarse, en caso de que la condición fuese falsa la primera vez que se comprueba.

La sintaxis de la sentencia WHILE, utilizando la notación BNF, es:

Sentencia_WHILE	::=	WHILE Condición DO Secuencia_de_Sentencias END
Condición	::=	Expresión

❏ Ejemplo: Escribir un programa que, dado un número natural *Tope*, calcule el mínimo valor de *N* tal que $0+1+2+...+(N-1)+N \geq Tope$.

Un posible algoritmo para resolver este problema es el siguiente:

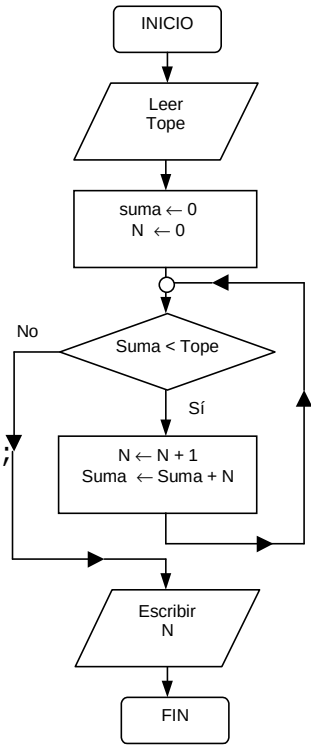
1. Leer del teclado el valor de la variable `Tope`
2. Guardar el valor cero en la variable `Suma`
3. Guardar el valor cero en la variable `N`
4. Mientras que el valor de la variable `Suma` sea menor al de la variable `Tope`
   4.1. Sumar uno a la variable `N`
   4.2. Sumar N a la variable `Suma`
5. Escribir el valor de la variable `N`

Este algoritmo da lugar al siguiente diagrama de flujo y su correspondiente programa en *Modula-2*:

```
MODULE Suma;
FROM IO IMPORT RdCard, RdLn, WrCard, WrStr;
VAR
  Tope, N, Suma :CARDINAL;
BEGIN
  WrStr("Dame el tope a sumar: ");
  Tope := RdCard(); RdLn;

  Suma := 0;
  N    := 0;
  WHILE Suma < Tope DO
    N    := N+1;
    Suma := Suma + N
  END;

  WrStr("El mínimo N que cumple la condición es ");
  WrCard(N,0)
END Suma.
```



❏

5.2.3 La sentencia REPEAT

La sentencia REPEAT permite repetir una serie de instrucciones hasta que cierta condición sea cierta. La sentencia REPEAT debe utilizarse exclusivamente para expresar bucles indefinidos.

La forma que toma esta sentencia es la siguiente:

```
REPEAT
   Acciones
UNTIL Condición
```

Las palabras REPEAT y UNTIL son palabras reservadas del lenguaje y delimitan cada una de las partes de la sentencia. La condición que aparece tras la palabras UNTIL es denominada *condición de salida* del bucle. La condición de salida debe ser una expresión de tipo booleano. El conjunto de acciones que aparecen entre las palabras REPEAT y UNTIL constituyen el *cuerpo del bucle*.

El diagrama de flujo correspondiente a esta sentencia es el siguiente:

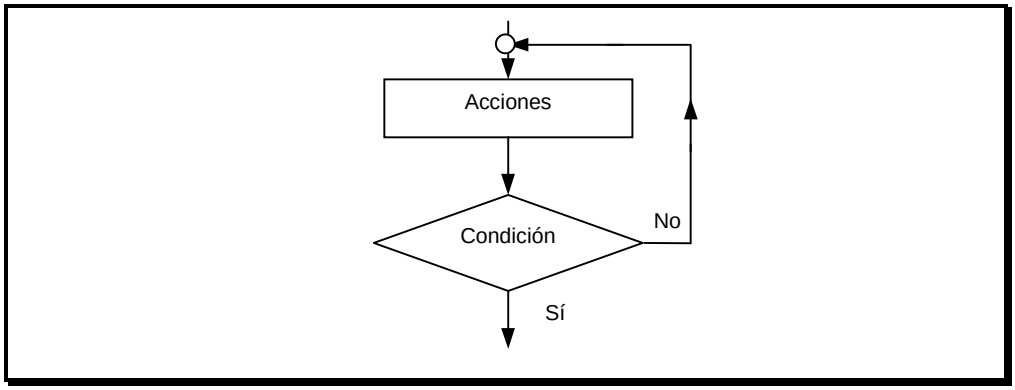


Figura 4. La sentencia REPEAT.

Obsérvese que la condición se vuelve a evaluar, para cada repetición del bucle, después de ejecutar el cuerpo. Se sale del bucle una vez que la condición sea cierta (es por esto que la condición se llama de salida). El cuerpo del bucle se ejecuta al menos una vez.

La sintaxis de la sentencia REPEAT, utilizando la notación BNF, es:

```
Sentencia_REPEAT  ::=  REPEAT
                           Secuencia_de_Sentencias
                       UNTIL Condición
```

❑ Ejemplo: Escribir un programa que calcule una aproximación al número *e* utilizando la siguiente igualdad:

$$e = \sum_{i=0}^{\infty} \frac{1}{i!} = 1 + \frac{1}{1!} + \frac{1}{2!} + \cdots$$

Dado que es imposible sumar infinitos términos, el programa dejará de sumar términos una vez que el valor del último término sumado sea menor a una diezmilésima.

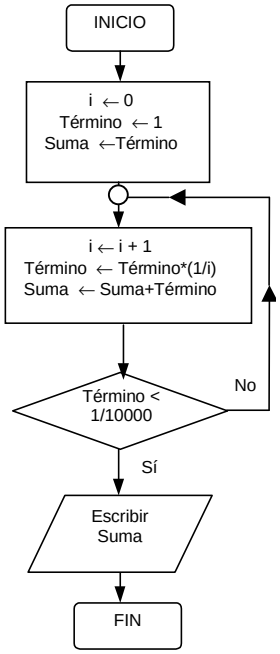
Llamemos término *i*-ésimo al término 1/*i*!. Es interesante observar que, conocido el valor del término *i*-ésimo, basta con multiplicar éste por 1/(*i*+1) para obtener el valor del término siguiente.

En el algoritmo utilizaremos tres variables. La variable `i` indicará, en todo momento, el orden del término que estemos sumando. La variable `Termino` almacenará siempre el valor del término actual. Por último, la variable `Suma` irá acumulando la suma de términos.

El diagrama de flujo correspondiente al algoritmo y su correspondiente programa en *Modula-2* son los siguientes:

```
MODULE NumeroE;
FROM IO IMPORT WrLngReal, WrStr;
CONST
  DIEZMILESIMA = 1.0/10000.0;
  PRECISION    = 10;
  ANCHO        = 0;
VAR
  i, Termino, Suma :LONGREAL;
BEGIN
  i       := 0.;
  Termino := 1.;
  Suma    := Termino;
  REPEAT
    i       := i+1.;
    Termino := Termino * (1.0/i);
    Suma    := Suma + Termino
  UNTIL Termino < DIEZMILESIMA;

  WrStr("La aproximación al número e es ");
  WrLngReal(Suma,PRECISION,ANCHO)
END NumeroE.
```



❑

5.2.4 Las sentencias LOOP y EXIT

La sentencia LOOP permite repetir indefinidamente una serie de instrucciones.

La forma que toma esta sentencia es la siguiente:

```
LOOP
  Acciones
END
```

Las palabras LOOP y END son palabras reservadas del lenguaje y delimitan el *cuerpo del bucle*.

El diagrama de flujo correspondiente a esta sentencia es el siguiente:

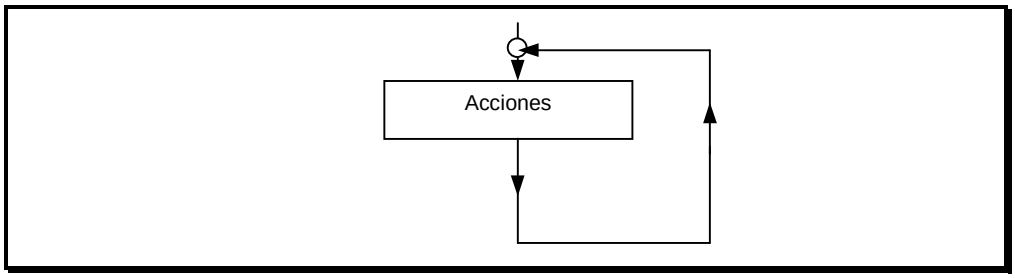


Figura 5. La sentencia LOOP.

Obsérvese que el cuerpo del bucle es ejecutado indefinidamente. Nunca se pasa a la instrucción siguiente al bucle.

La sentencia EXIT puede ser utilizada para salir de un bucle LOOP. La ejecución de una sentencia EXIT, dentro del cuerpo de un bucle LOOP, hace que el bucle finalice y que la ejecución del programa continúe con la instrucción siguiente a la palabra reservada END.

El modo habitual de combinar las sentencias LOOP y EXIT es el siguiente:

```
LOOP
   Acción A;
   IF Condición THEN
      EXIT
   END;
   Acción B
END
```

El diagrama de flujo correspondiente a esta construcción es:

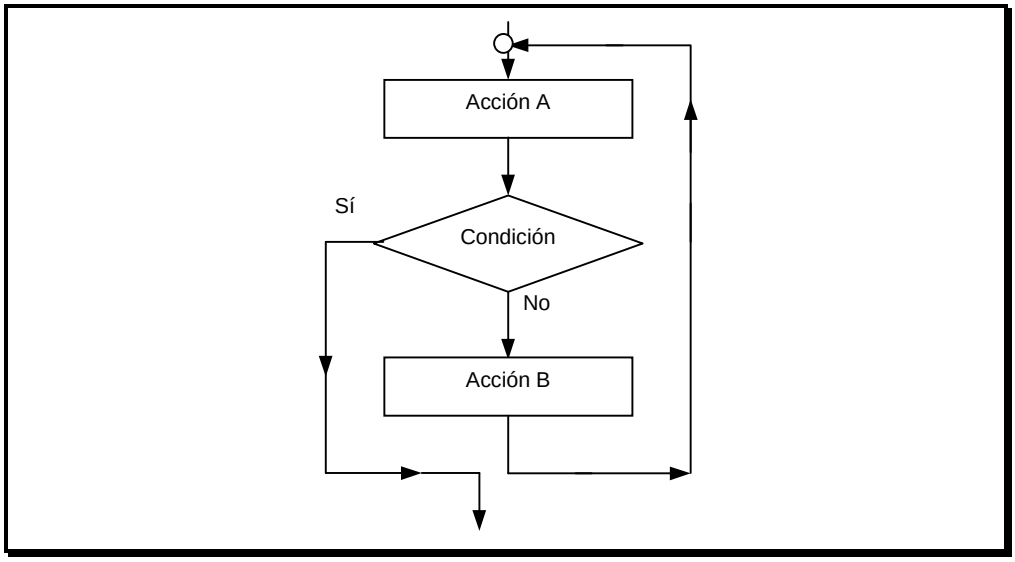


Figura 6. Las sentencias LOOP y EXIT.

La sintaxis de las sentencias LOOP y EXIT, utilizando la notación BNF, es:

Sentencia_LOOP	**::=**	LOOP Secuencia_de_Sentencias END
Sentencia_EXIT	**::=**	EXIT

Algunas consideraciones sobre estas sentencias son:

- La sentencia EXIT sólo puede aparecer dentro del cuerpo de un bucle LOOP. No se puede usar la sentencia EXIT para salir de un bucle FOR, WHILE o REPEAT.
- Cuando una sentencia EXIT aparece en un bucle LOOP anidado dentro de otro bucle LOOP, la ejecución de dicha sentencia sólo hará que abandone el bucle LOOP más interno.
- Si la comprobación de la condición de salida se hace justo al principio del bucle LOOP, se debe sustituir el bucle LOOP por uno WHILE equivalente:

```
LOOP
   IF Condición THEN
      EXIT
   END;
   Acciones
END
```

≡

```
WHILE NOT Condición DO
   Acciones
END
```

- Si la comprobación de la condición de salida se hace justo al final del bucle LOOP, se debe sustituir el bucle LOOP por uno REPEAT equivalente:

```
LOOP
  Acciones                      REPEAT
  IF Condición THEN     ≡         Acciones
    EXIT                        UNTIL Condición
  END;
END
```

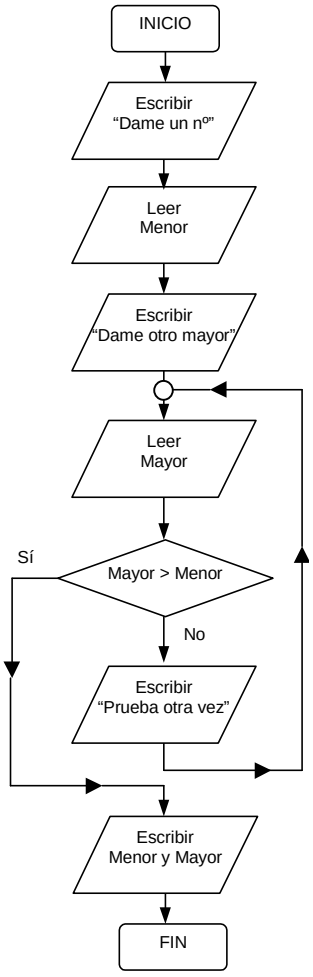
- De los dos puntos anteriores, se deduce que *sólo* se debe utilizar las combinación de las sentencias LOOP y EXIT cuando la condición de salida está situada en un punto intermedio del cuerpo del bucle.
- *Modula-2* no limita el número de sentencias EXIT que pueden aparecer dentro de un bucle LOOP. El uso de esta característica del lenguaje hace que los programas resultantes sean difíciles de entender. Además de ser un mal estilo de programación, el uso de esta característica del lenguaje no es necesaria. Por todo esto, es muy importante evitar que aparezca más de una sentencia EXIT dentro de cada bucle LOOP.

❏ Ejemplo: Escribir un programa que lea dos números enteros del teclado, de modo que el segundo sea mayor que el primero.

Utilizaremos dos variables enteras (`Menor` y `Mayor`). Leeremos un valor en la variable `Menor` y repetiremos el proceso de leer otro número en la variable `Mayor` hasta que éste sea estrictamente mayor al primero.

El diagrama de flujo correspondiente al algoritmo y su correspondiente programa en *Modula-2* son los siguientes:

```
MODULE Numeros;
FROM IO IMPORT WrStr, WrInt, RdInt, RdLn;
VAR
  Menor, Mayor : INTEGER;
BEGIN
  WrStr("Dame un número: ");
  Menor := RdInt(); RdLn;
  WrStr("Dame un número mayor: ");
  LOOP
    Mayor := RdInt(); RdLn;
    IF Mayor > Menor THEN
      EXIT
    END;
    WrStr("Debe ser mayor. Prueba de nuevo")
  END
  WrStr("Los números son: ");
  WrInt(Menor,0);
  WrStr(" y ");
  WrInt(Mayor,0)
END Numeros.
```



❏

5.3 Diseño de bucles

5.3.1 Anidamiento de bucles

Modula-2 no impone ningún tipo de restricciones sobre el tipo de sentencias que pueden aparecer en el cuerpo de un bucle. En concreto, dentro del cuerpo de un bucle puede aparecer otro bucle. A esto se le llama *anidamiento de bucles*.

❑ Ejemplo: Escribir un programa que calcule el factorial de un número leído del teclado sin usar la operación de multiplicación.

Desarrollamos primero un algoritmo que calcula el producto de dos números enteros y positivos `i` y `f`. Para ello, sumamos f veces el valor de i:

$$i \times f = \underbrace{i + i + \cdots + i}_{f \text{ veces}}$$

Para realizar el cálculo anterior, hay que repetir la acción sumar `i` un número predeterminado de veces (`f` veces). El modo más adecuado de expresar esto es mediante una sentencia FOR:

```
MODULE Producto;
FROM IO IMPORT RdCard, RdLn, WrCard, WrStr;
VAR
  i, f, Producto, j :CARDINAL;
BEGIN
  WrStr("Dame un número: ");
  i := RdCard(); RdLn;
  WrStr("Dame otro: ");
  f := RdCard(); RdLn;

  Producto := 0;
  FOR j := 1 TO f DO
    Producto := Producto + i
  END;

  WrStr("El producto es ");
  WrCard (Producto, 0)
END Producto.
```

Para calcular el factorial de un número `n` debemos calcular la siguiente expresión:

$$n! = \underbrace{n \times (n-1) \times (n-2) \times \cdots 2 \times 1}_{n \text{ productos}}$$

Para realizar el cálculo anterior, hay que repetir `n` veces la acción multiplicar por `i` (variando el valor de `i` desde uno hasta `n`). De nuevo, el modo más adecuado de expresar esto es mediante un bucle FOR:

```
MODULE Fact;
FROM IO IMPORT RdCard, RdLn, WrCard, WrStr;
VAR
  n, i, f : CARDINAL;
BEGIN
  WrStr("Dame un número: ");
  n := RdCard(); RdLn;

  f := 1;
  FOR i := 1 TO n DO
    f := f * i
  END;
```

```
  WrStr("El factorial es ");
  WrCard (f, 0)
END Fact.
```

Podemos utilizar el primer algoritmo para calcular el producto `f * i`. Combinando los dos algoritmos, obtenemos un algoritmo que calcula el factorial de un número sin usar la operación de multiplicación:

```
MODULE FactSum;
FROM IO IMPORT RdCard, RdLn, WrCard, WrStr;
VAR
  n, i, j, f, Producto :CARDINAL;
BEGIN
  WrStr("Dame un número: ");
  n := RdCard(); RdLn;

  f := 1;
  FOR i := 1 TO n DO

    Producto := 0;
    FOR j = 1 TO f DO
      Producto := Producto + i      ≡  f := f * i
    END;
    f := Producto

  END;

  WrStr("El factorial es ");
  WrCard (f, 0)
END FactSum.
```

Nótese que ha sido necesario anidar dos bucles FOR para resolver el problema.

❏

❏ Ejercicio: Dibuja el diagrama de flujo correspondiente a este algoritmo. Sigue los pasos que se realizarían para calcular 3! usando el algoritmo anterior.

❏

Al diseñar un algoritmo con bucles anidados, tanto el comienzo como el final del bucle anidado deben estar dentro del bucle exterior. Así, el siguiente anidamiento de bucles es correcto:

```
WHILE Condición DO
  Acciones
  REPEAT
    Acciones
  UNTIL Condición
  Acciones
END
```

El siguiente anidamiento no es correcto, ya que el bucle REPEAT está anidado en el bucle WHILE y debe acabar antes que éste último:

```
WHILE Condición DO
  Acciones
  REPEAT
    Acciones
END
  UNTIL Condición
  Acciones
```

Otra consideración importante es que a la hora de anidar bucles FOR, es imprescindible utilizar una variable de control distinta para cada uno de ellos. Por ejemplo,

```
FOR i := 1 TO 3 DO
  FOR j := 1 TO 2 DO
    WrChar('*')
  END
END
```

Obsérvese que este trozo de programa escribe seis asteriscos por pantalla.

5.3.2 Bucles infinitos

Un *bucle infinito* es un bucle que nunca acaba.

La ejecución de un programa que contiene un bucle infinito no acaba. En *TopSpeed Modula-2*, el programador puede interrumpir un programa de este tipo pulsando simultáneamente las teclas *Ctrl* y C.

A veces, el bucle infinito es debido a que el programador olvidó alguna sentencia en el cuerpo del bucle.

❏ Ejemplo: El siguiente bucle no acaba ya que falta una sentencia en el cuerpo que incremente el valor de la variable `N`:

```
Suma := 0;
N := 0;
WHILE N <= 100 DO
  Suma := Suma + N
END
WrCard(Suma,0)
```

❏

En otras ocasiones, el bucle infinito se debe a que es imposible alcanzar la condición de finalización del bucle. Este tipo de situaciones se suelen deber a un error de diseño del programa.

❏ Ejemplo: El siguiente bucle no acaba ya que la variable `N` nunca alcanzará un valor negativo:

```
Suma := 0;
N := 100;
WHILE N >= 0 DO
  Suma := Suma + N;
  N := N + 1
END
WrCard(Suma,0)
```

❏

Con objeto de evitar la aparición de bucles infinitos, es importante comprobar que las condiciones de finalización de un bucle serán alcanzadas.

Relación de problemas (Tema 5)

1. Escribir un programa que lea una lista de valores reales positivos terminada por un número negativo y calcule la media de esta lista.

2. Escribe un programa que lea un número entero positivo por teclado y determine si es primo o compuesto.

3. Diseña un programa que escriba en pantalla la tabla de multiplicar de un numero del 1 al 9 introducido desde el teclado.

4. Escribir un programa que lea una lista de valores enteros positivos terminada por 0 y determine los valores máximo y mínimo.

5. Escribe un programa que calcule e imprima en pantalla los N primeros números primos, siendo N un número que se introduce por teclado.

6. Escribe un programa que calcule el valor de seno x dado por:

$$Seno\ x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

Este programa debe leer dos variables `x` y `precisión` donde `x` es un ángulo en radianes y `precisión` debe ser positiva y menor que 1. La serie debe sumarse hasta que el valor absoluto del último sumando sea menor que `precisión`, incluyendo este último sumando en la suma.

NOTA: El término con numerador x^n de la serie se puede calcular multiplicando el previo por $-\frac{x^2}{n \times (n-1)}$

7. Un billete de segunda clase en tren de Londres a París vale normalmente 80 libras. Sin embargo, existen tarifas especiales en los siguientes casos:

65 años o más: 52.60 libras

12-15 años: 55.00 libras

4-11 años: 40.20 libras

menos de 4 años: gratis

Supón que se quiere averiguar el coste total para una familia. Escribe un programa que lea la edad de cada miembro de la familia y calcule el coste total de los billetes.

8. Si x_0 es una aproximación a $\sqrt{a}$ entonces una aproximación mejor es:

$$x_1 = \frac{1}{2}\left(x_0 + \frac{a}{x_0} \right)$$

Podemos repetir esto con x_1 y obtener una nueva aproximación. Este proceso, que es una aplicación del método de Newton-Raphson, se puede generalizar a:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n} \right), n = 1,2,\cdots$$

Escribir un programa que lea los valores de *a* y *precisión*. El programa debe calcular la raíz de *a* utilizando el método descrito. El programa se detiene cuando el valor absoluto de la diferencia entre dos aproximaciones consecutivas es menor que *precisión*. El valor inicial para la iteración será *a/2*. El programa debe mostrar en pantalla todos las aproximaciones obtenidas.

9. Escribir tres programas para calcular el factorial de un número entero leído desde el teclado utilizando las sentencias FOR, WHILE y REPEAT.

10. Escribir un programa que visualice en pantalla una figura similar a la siguiente:

```
*
**
***
****
```

Siendo el número de líneas un dato que se lee al principio del programa.

11. Un número es perfecto si coincide con la suma de todos sus divisores excepto el mismo. Por ejemplo 28 es perfecto ya que sus divisores son 1, 2, 4, 7, 14 y 1+2+4+7+14 = 28. Escribe un programa que determine si un número leído del teclado es perfecto.

12. Para encontrar el máximo común divisor (mcd) de dos números enteros se puede emplear el algoritmo de Euclides: dados los números *a* y *b* (siendo $a > b$) se divide *a* por *b* obteniendo el cociente q_1 y el resto r_1. Si $r_1 \neq 0$, se divide *b* por r_1, obteniendo el cociente q_2, y el resto r_2. Si $r_2 \neq 0$ se divide r_1 por r_2 obteniendo cocientes y restos sucesivos. El proceso continúa hasta obtener un resto igual a 0. El resto anterior a éste es el máximo común divisor de los números iniciales (*a* y *b*). Escribe un programa que lea dos números enteros positivos del teclado y calcule su máximo común divisor y su mínimo común múltiplo (mcm).

NOTA: obtener el mínimo común múltiplo a partir de la siguiente igualdad:

$$\text{mcd}\,(a,b) * \text{mcm}\,(a,b) = a * b$$

13. Diseñar un programa que calcule el cociente entero y el resto correspondiente para dos números naturales dados por teclado, sin utilizar los operadores DIV y MOD.

14. Escribir un programa que lea un número N y calcule el término N-ésimo de la sucesión de Fibonacci:

$$\begin{aligned} Fibonacci(1) &= 0 \\ Fibonacci(2) &= 1 \\ Fibonacci(n) &= Fibonacci(n-1) + Fibonacci(n-2)\,, si\ n > 2 \end{aligned}$$

Realizar los ejercicios 3, 8 y 10 como práctica en el laboratorio.

Relación complementaria (Tema 5)

1. Escribir un programa que calcule y visualice el más grande, el más pequeño y la media de N números. El valor N se solicitará al principio del programa y los números serán introducidos por teclado.

2. Encontrar el número natural N más pequeño tal que la suma de los N primeros números enteros exceda una cantidad introducida por teclado.

3. El valor *exp(x)* se puede calcular mediante la siguiente serie:

$$\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Escribir un programa que lea un valor *x* de teclado y calcule el valor de *exp(x)* deteniendo el cálculo cuando el valor del último término sumado sea inferior a una diezmilésima.

4. Escribir un programa que calcule una aproximación al número π sabiendo que

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$$

para ello sumará términos de la serie anterior mientras que sus valores absolutos sean mayores a una diezmilésima y del valor obtenido despejará π.

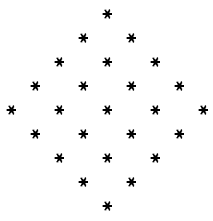
5. Diseña un programa que multiplique dos números enteros mediante el algoritmo de la multiplicación rusa. Este algoritmo multiplica por 2 el multiplicando y divide por dos el multiplicador hasta que el multiplicador toma el valor 1. Después suma todos los multiplicandos correspondientes a multiplicadores impares. Dicha suma es el producto de los números originales.

Ejemplo: 37 * 12 = 444 (multiplicador: 37, multiplicando:12)

```
37  12
18   24
 9  48
 4  96
 2  192
 1  384

12 + 48 + 384 = 444
```

6. Escribe un algoritmo que lea un número natural N y un carácter. La salida debe ser un rombo compuesto del carácter y de la anchura que especifica el número N. Por ejemplo, si N es 5 y el carácter es *, el rombo sería:



7. Escribe un algoritmo que imprima una pirámide de dígitos como la de la figura, tomando como entrada el número de filas de la misma siendo este número menor o igual que 9.

```
    1
   1 2 1
  1 2 3 2 1
 1 2 3 4 3 2 1
1 2 3 4 5 4 3 2 1
```

8. Escribir un programa que lea del teclado un número N (expresado en base 10), una base B y pase N a base B.

9. Escribe un programa que determine si la cadena “abc” aparece en una sucesión de caracteres cuyo final viene dado por un punto.