Instituto Superior Pedagógico
"José Martí"
Camagüey

Facultad de Ciencias
Departamento de Informática

APUNTES SOBRE FUNDAMENTOS DE PROGRAMACIÓN

AUTORES: MsC. Rafael Ramírez Varona.
Lic. Eduardo Pérez Suárez.

2003

INTRODUCCIÓN

Este material pretende servir de texto de consulta para aquellos estudiantes y profesores que se inician en el estudio de la informática, particularmente aquellos que se forman en los Institutos Superiores Pedagógicos del país. El mismo es el resultado de revisión, actualización, ampliación y complementación de la obra de Joelle Biondi y Gilles Clavel.

Se parte de la necesidad de abordar la lógica en la enseñanza de la Informática, no solo desde la perspectiva de las asignaturas de Lenguajes y Técnicas de Programación sino, que soporta además, la base de la disciplina de Sistemas de Aplicación.

En el material se enuncian algunos principios y consideraciones generales para abordar la resolución de problemas por computadoras unidos a una notación algorítmica en extremos simples pero consistentes y generales, que son aplicables al proceder práctico en cualquiera de las disciplinas antes referenciadas.

Aspiramos a que los docentes hallen en esta obra no sólo los contenidos del programa, sino también algunas ideas o sugerencias sobre su tratamiento metodológico, a partir del propio enfoque que se ha tenido como premisa para el desarrollo de cada temática. No obstante, consideramos que resulta una herramienta muy valiosa para el desarrollo de cualquier curso relacionado con la enseñanza de la Informática y una alternativa para impulsar su estudio. No obstante, un esfuerzo serio hubo que realizar para en un tiempo corto poder disponer de él. Los autores lo han trabajado con el máximo de esmero para que Ud., como usuario pueda disponer del texto. Esperamos le resulte de utilidad y deseamos realmente que las opiniones que puedan existir nos la hagan llegar, seguro que podrán contribuir a mejorar una próxima edición.

1- NOCIÓN DE ALGORITMO.

1.1- Noción de acción y de procesador.

Consideremos los enunciados siguientes:

E1: Elaboración de una tortilla de 6 huevos.

a) Romper 6 huevos en un plato;
b) batir la clara y la yema con un tenedor;
c) calentar el aceite en un sartén;
d) cuando el aceite está caliente, verter el contenido del plato;
e) quitar el sartén del fuego cuando la tortilla esté hecha.

E2: Cálculo de la media aritmética de números con una calculadora.

a) Pulsar “C”;
b) teclear el primer número;
c) pulsar “+”;
d) teclear el segundo número;
e) pulsar “+”;
f) teclear el tercer número;
g) pulsar “÷”;
h) teclear 3;
i) pulsar “=”. Aparece el resultado.

Los enunciados E1 y E2 describen cada uno un cierto trabajo.

Llamamos **procesador** a toda entidad capaz de entender un enunciado y de ejecutar el trabajo indicado.

En el caso de los enunciados E1 y E2, una persona que sepa leer y disponga de los utensilios necesarios puede ser un perfecto procesador. El procesador no puede ejecutar el trabajo solicitado a menos que se den las condiciones requeridas por el enunciado. En el caso del E1, los utensilios serán: un fogón, 6 huevos, un plato, un tenedor, un sartén y aceite. En el caso del enunciado E2, serán una calculadora y 3 números dados.

El conjunto de los utensilios necesarios para la ejecución de un trabajo constituye el **entorno** de dicho trabajo.

El entorno, para un procesador dado, es, por tanto, específico del trabajo a realizar; la elección hecha para efectuar este trabajo depende de los utensilios puestos a disposición del procesador. Puede haber una interdependencia entre trabajo y entorno. Ejemplo:

Un comerciante debe devolver el cambio de $50, cobrando $20,50. Consideremos los 3 entornos siguientes:

e1: el comerciante sólo dispone de monedas de $10 y $1.
e2: el comerciante sólo dispone de monedas de $5 y 50 céntimos.
e3: el comerciante dispone de monedas de $10, $5, $1 y 50 céntimos.

En el primer caso, el procesador (el comerciante) no puede, en función de su entorno (las monedas), efectuar el trabajo que se le pide (dar el cambio). En los otros dos casos, el trabajo puede ser efectuado. Si consideramos entorno e3, el comerciante elegirá la forma de dar el cambio (elección del trabajo) en función del número respectivo de monedas de cada tipo (entorno): si dispone de un gran número de monedas de 50 céntimos y pocas monedas de $1, decidirá, con el fin de equilibrar las cantidades respectivas de los diferentes tipos de monedas, deshacerse preferentemente de las monedas de 50 céntimos.

Entorno y trabajo están estrechamente ligados. Cualquiera que sea el entorno, la ejecución de un trabajo no elemental no es generalmente inmediato: supone una cierta progresión hacia el fin deseado. Así, en la realización de los trabajos descritos en los enunciados E1 y E2, se distinguen las etapas (a, b, c, d, ...), cada una de las cuales se denomina acción.

Una **acción** es un suceso que modifica el entorno.

Consideremos la acción a) del enunciado E1: romper los huevos en un plato. Antes de ejecutar la acción, el plato estaba vacío. Después de ésta, el plato contiene 6 yemas y 6 claras de huevo. Además, la ejecución de una acción puede necesitar una observación del entorno (la acción d) sólo debe ser ejecutada cuando el sartén está caliente). Por regla general, el procesador respeta la secuencia de las acciones: las ejecuta en el orden en que aparecen en el enunciado.

En un mismo trabajo, a veces, ciertas acciones podrían ser ejecutadas al mismo tiempo. La ventaja de estas ejecuciones en paralelo es una ganancia de tiempo, aunque en general necesitan varios procesadores.

Como hemos visto, el enunciado de una misma acción puede aparecer varias veces en la descripción de un trabajo: en el enunciado E2, la acción de “pulsar “+”” aparece 3 veces.

Veamos, pues, el enunciado E3, en el cual x, y, z, representan acciones.

1. x
2. y
3. repetir x 6 veces.
4. z
5. y

La acción x aparece 2 veces; la acción y, 2 veces; x será ejecutada 7 veces e y 2 veces.

1.2- Acciones primitivas; descomposición de una acción.

En los enunciados E1 y E2, hemos dado por supuesto que el procesador sabía ejecutar las acciones enumeradas. Para este procesador, éstas son acciones primitivas.

Para un procesador dado, una **acción** es **primitiva** si el enunciado de dicha acción es suficiente para poder ejecutarla sin información suplementaria.

En el ejemplo anterior (enunciado E3), la acción 3) debe ser para le procesador una acción primitiva. Por lo que ejecutará x, y, x, x, x, x, x, z, y. En el caso de que la acción 3) no sea primitiva, habrá que reemplazarla por la secuencia x, x, x, x, x, x.

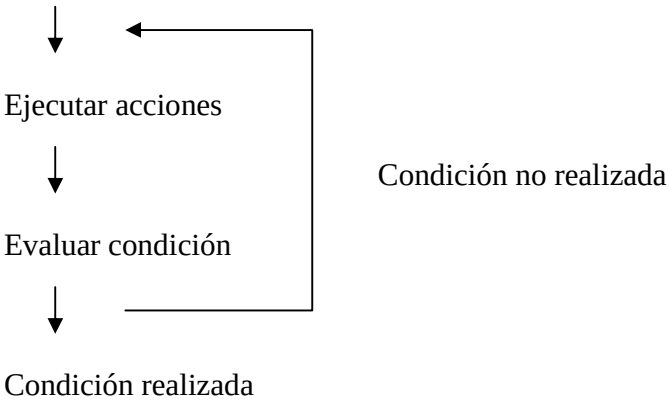
Una acción no primitiva debe ser **descompuesta** en acciones primitivas.

Por ejemplo, en el enunciado E1, si el procesador es un niño, la acción a) (romper 6 huevos en un plato) puede ser un poco explícita. Será necesario entonces descomponer esta acción. Por ejemplo:

a1: poner los 6 huevos en la superficie de trabajo.
repetir
    a2: coger un huevo de la superficie de trabajo
    a3: romperlo y verter su contenido en el plato.
    a4: tirar las cáscaras a la basura.
hasta que no queden huevos en la superficie de trabajo.

Esta descomposición utiliza un esquema repetitivo cuyo principio está esquematizado de la manera siguiente:

Repetir acciones
Hasta que condición



Si suponemos que en la descomposición de la acción a) del enunciado E1, las acciones a1, a2, a3, a4 son primitivas, que el procesador entiende el esquema *repetir...hasta que* y que es capaz de hacer la evaluación de la condición, podemos decir entonces que ha sido descompuesta en acciones primitivas.

Existen numerosos medios para descomponer una acción en acciones primitivas. El análisis descendente es uno de los métodos para obtener esta descomposición.

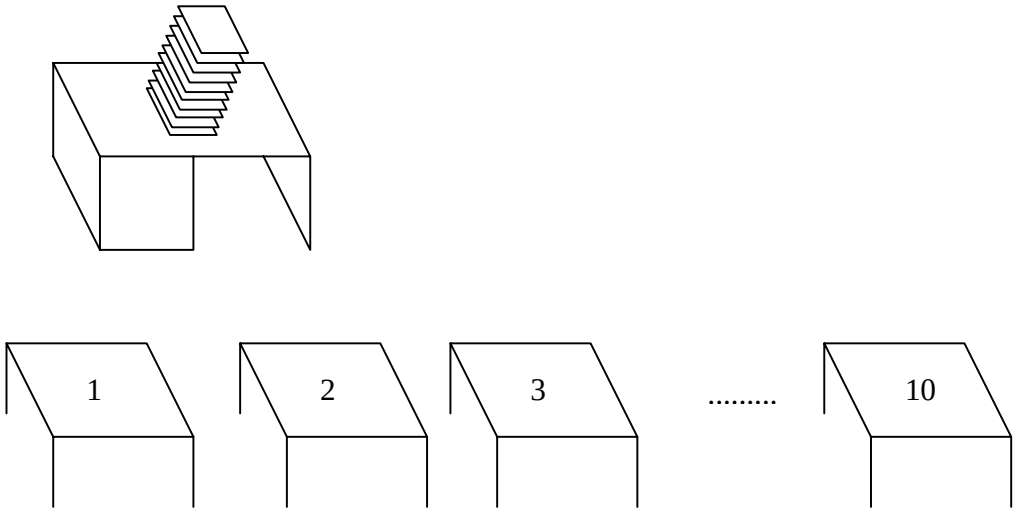
1.3- Noción de análisis descendente.

Para ilustrar este principio, lo aplicaremos a un ejemplo concreto.

Supongamos que tenemos que realizar la agrupación de un documento de 10 páginas que ha sido enviado a un servicio de fotocopiado.

Para cada una de las páginas del documento, el servicio de reproducción ha hecho alrededor de un centenar de ejemplares idénticos, la incertidumbre en el número es debida al hecho de que la máquina fotocopiadora no funciona con total perfección. Disponemos de una pila de 10 paquetes de hojas fotocopiadas, cada paquete está compuesto de un centenar de hojas que corresponden a la misma página. Los paquetes no están dispuestos en la pila por orden de página. Se quieren agrupar las fotocopias por un individuo (el procesador) que dispone de los objetos siguientes (entorno) para efectuar su tarea:

- 10 paquetes de hojas numeradas.
- un escritorio donde están apiladas, paquete por paquete, las hojas que se han de ordenar,
- una presilladora,
- 10 mesas numeradas del 1 al 10. La mesa con el número 1 deberá recibir las hojas cuyo número de página sea 1.



Las únicas funciones a realizar por el individuo (acciones primitivas) son:

- coger y poner,
- ir de una mesa a otra (pasar de la mesa 1 a la mesa de número inmediatamente superior),
- ir a una mesa de número correcto (ya que se supone que el individuo es capaz de reconocer la igualdad entre 2 números: el de la mesa y un número dado),
- reconocer la presencia o ausencia de hojas sobre una mesa,
- presillar.

El trabajo a realizar puede ser descrito por:

A { A partir de la pila sobre la mesa, agrupar las fotocopias.

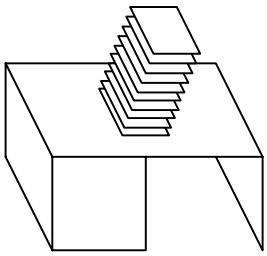
El enunciado A no representa una acción primitiva para el individuo; debe, por tanto, ser descompuesta. Esto sería:

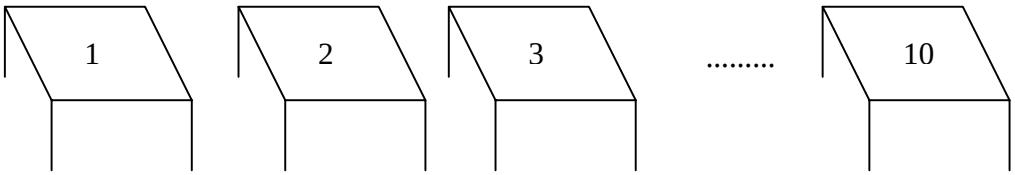
Estado inicial:

A

B- Distribuir los paquetes sobre las mesas a razón de un paquete por mesa de forma que el paquete relativo a la hoja i sea puesto sobre la mesa i.

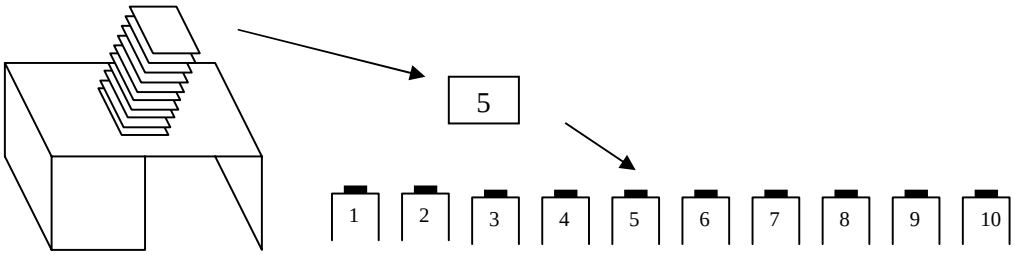
C- Reunir y presillar los ejemplares y ponerlos sobre el escritorio.



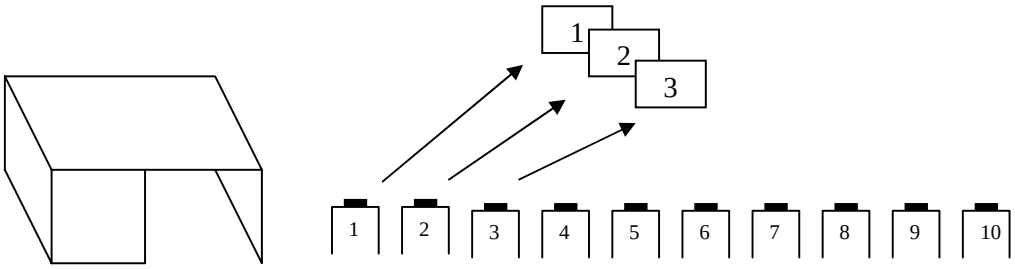


Las acciones B y C no son primitivas, por lo que se ha de continuar el análisis descendente. B y C pueden ser descompuestas independientemente una de otra (estas acciones son relativamente independientes, aunque una debe preceder a la otra en la ejecución: podrían entonces ser descompuestas por dos personas diferentes).

Etapa B:



Etapa C:



Se descompone B y C de la siguiente forma:

A

B

repetir

B1: coger el paquete situado encima de la pila sobre el escritorio.

B2: ir a la mesa cuyo número es el mismo que el de las hojas del paquete.

B3: poner el paquete sobre la mesa.

hasta que (B4) se acabe la pila que hay sobre el escritorio. ejemplares y ponerlos sobre el escritorio.

C

repetir

C1- coger las hojas necesarias para un ejemplar.

C2: presillar el ejemplar.

C3: ponerlo sobre el escritorio.

hasta que (C4) una de las mesas quede sin hojas.

Si el procesador entiende el esquema “repetir...hasta que”, entonces no vale la pena continuar la descomposición de B: las acciones B1, B2 y B3 son primitivas y el procesador es capaz de reconocer la presencia o ausencia de hojas sobre una mesa y, por tanto, determinar si la condición B4 se cumple. Para la descomposición de C, C2 y C3 son primitivas y la condición C4 es entendida por el procesador. La

acción C1, en cambio, no es una acción primitiva, por lo que hay que descomponerla.

Para realizar C1, basta con ir a la primera mesa, coger una hoja y pasar a la mesa siguiente, coger una hoja y pasar a la mesa siguiente, etc...Vemos entonces aparecer una repetición de la secuencia:

- ir a la mesa siguiente,
- coger una hoja,

y se puede expresar una descomposición de C1 como la siguiente:

C1 {
C'1: colocarse en la mesa No 1.
C'2: coger una hoja de la mesa No 1.

repetir

C'3: ir a la mesa siguiente.
C'4: coger una hoja de esta mesa y ponerla en la mano bajo las otras.

hasta que la mesa sea la No 10.
}

Las acciones C'1 a C'4 son primitivas, el análisis está acabado. La secuencia final, que llamaremos algoritmo, es la siguiente:

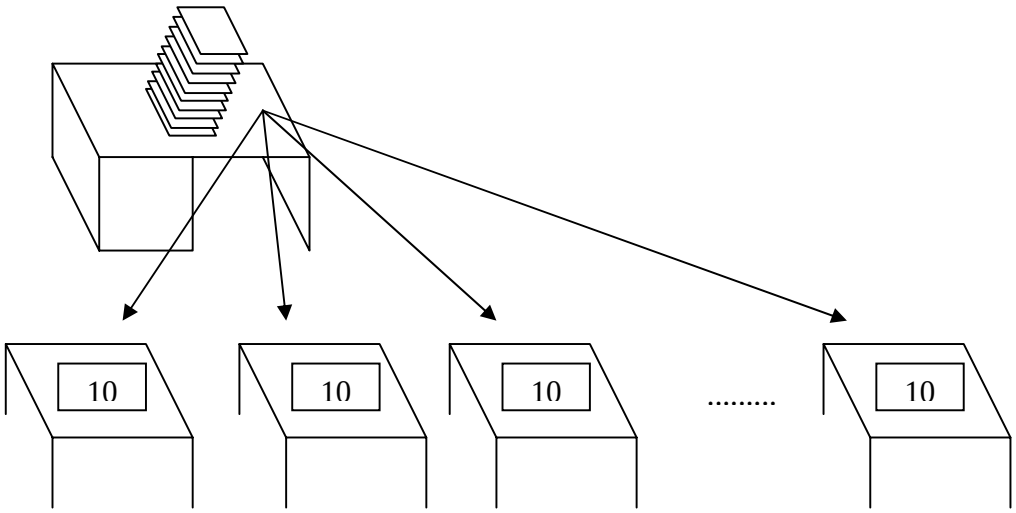
```
inicio
    repetir
        B1: coger el paquete situado encima de la pila sobre el escritorio.
        B2: ir a la mesa cuyo número es el mismo que el de las hojas del paquete.
        B3: poner el paquete sobre la mesa.
        repetir hasta se acabe la pila que hay sobre el escritorio
            C'1: colocarse en la mesa No 1.
            C'2: coger una hoja de la mesa No 1.
        repetir
            C'3: ir a la mesa siguiente.
            C'4: coger una hoja de esta mesa y ponerla en la mano bajo las otras.
        hasta que la mesa sea la No 10.
        C2: presillar el ejemplar.
        C3: ponerlo sobre el escritorio.
    hasta que (C4) una de las mesas quede sin hojas.
fin
```

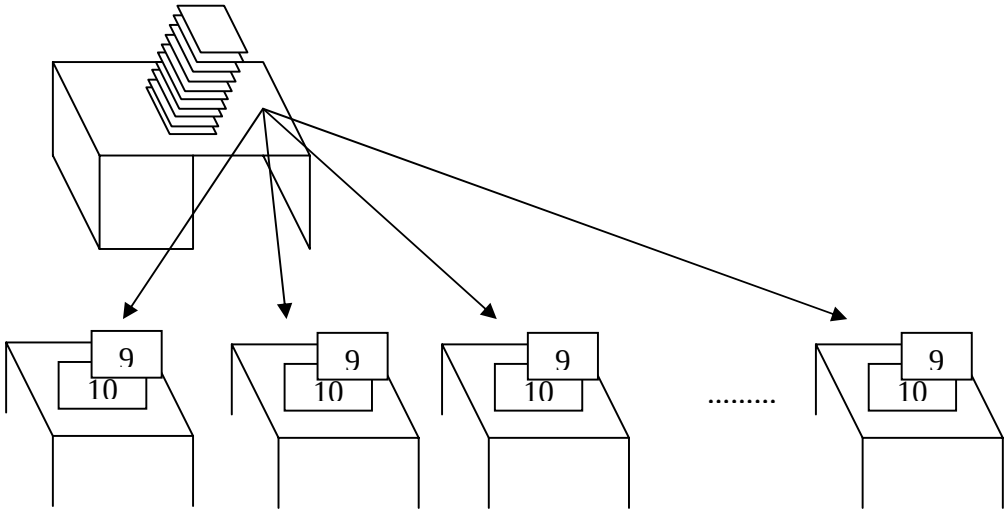
La secuencia de acciones así obtenida utiliza sucesivamente dos esquemas "repetir...hasta que" y el segundo esquema repetitivo contiene otro esquema

“repetir...hasta que” (esquemas anidados). El enunciado del problema ha sido puesto en forma de secuencia de acciones primitivas. A partir de este ejemplo ya podemos dar una primera definición de la noción de algoritmo:

Dado un procesador bien definido y un tratamiento a ejecutar por dicho procesador, un **algoritmo** del tratamiento es el enunciado de una secuencia de acciones primitivas que realizan este tratamiento.

La palabra algoritmo proviene del nombre de un matemático árabe (El Khowarismi), originario de la antigua ciudad de Khowarism, hoy Khiva, situada en la antigua URSS. En la definición se dice “un algoritmo” y no “el algoritmo”: por regla general, las soluciones no son únicas. Así para la agrupación de fotocopias pudiera utilizarse otro método:





1. Apilamos sobre el escritorio los paquetes en el orden 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 (el paquete número 10 encima de la pila).

2. Para cada paquete de la pila (en el orden 10, 9, 8, 7, 6, 5, 4, 3, 2, 1), se efectúa el proceso siguiente:

   - coger el paquete;
   - repartir una hoja de dicho paquete sobre cada una de las mesas;
   - poner el paquete al lado de la pila.

   Tenemos así sobre la cada mesa una hoja 10, después una 9, etc. Cuando lo hemos hecho con todos los paquetes, tenemos sobre cada mesa un ejemplar sin presillar y sobre el escritorio, los paquetes apilados en el orden (10, 9, 8, 7, 6, 5, 4, 3, 2, 1).

3. Presillamos los 10 ejemplares y los ponemos en el escritorio.
4. Giramos la pila del escritorio y volvemos a empezar...etc...hasta que uno de los paquetes se acabe.

Como al anterior, este método permite realizar el trabajo requerido. Para elegir entre uno u otro, habría que determinar cuál es el más eficaz. La eficacia de un algoritmo puede ser definida de varias formas según las características del entorno o del procesador. En algunos casos, el algoritmo más eficaz será aquel que ejecute el trabajo lo más rápidamente posible. En otros casos, será el que resulte más económico y haga intervenir el entorno menos complejo.

1.4- Objeto de la programación.

Sea un trabajo a efectuar con ayuda de un procesador dado: el objetivo de la programación es establecer una secuencia de acciones que puedan ser ejecutadas por el procesador y que realicen este trabajo.

La programación está constituida por dos fases:

a) La resolución del problema, es decir, la determinación de un algoritmo que realice el tratamiento;
b) La adaptación del algoritmo al procesador. Se realiza por la codificación del algoritmo en el lenguaje específico del procesador utilizado.

Debido a que la mayoría de los ordenadores no pueden ejecutar directamente los algoritmos en su formas literal, es necesario codificarlos en un lenguaje apropiado denominado lenguaje de programación. Generalmente este trabajo no presenta mayores dificultades y hace intervenir principalmente mecanismos de traducción que requieren muy poco de la inteligencia de la persona que codifica. Los lenguajes de concepción reciente se aproximan bastante al formalismo que proponemos: los mecanismos de codificación son particularmente reducidos.

2- ACCIONES Y OBJETOS ELEMENTALES.

2.1- Formalización del entorno de un problema.

Para avanzar un poco más en el estudio de las bases de la programación, es necesario aprender a formalizar el entorno de un problema. Para ello, tenemos que definir un número de reglas que nos permitirán describir sin ambigüedad y con precisión los constituyentes del universo de un problema.

2.1.1- Objetos de un entorno.

Retomemos algunas de las acciones formuladas en el epígrafe anterior:

a : colocarse en la mesa No 1;
b : tomar el paquete encima de encima de la pila;
c : ir a la mesa siguiente;
d : teclear el primer número;
e :teclear “3”;
f :pulsar la tecla “+”.

Se trata de acciones que hacen intervenir distintos objetos:

– la mesa No 1;
– el paquete encima de encima de la pila;
– la mesa siguiente;
– el primer número;
– “3”;
– la tecla “+”.

Si intentamos extraer de estos ejemplos algunas características de la noción de objeto, vemos que cada objeto tiene un nombre que se utiliza para designarlo sin ambigüedad. Para citar los distintos objetos hemos dado una lista de sus nombres.

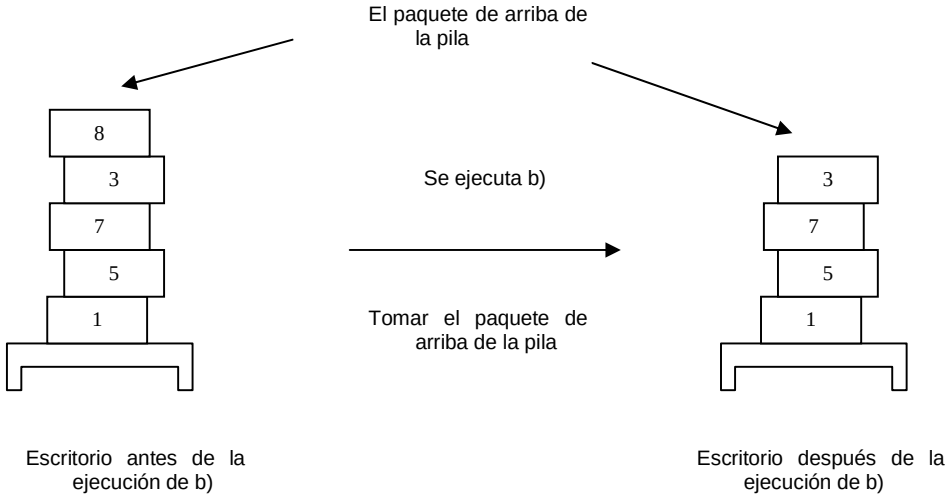
Igualmente, cada objeto tiene una utilización particular y las utilizaciones no son intercambiables: podemos dividir con la ayuda del objeto “3”, pero no podremos hacerlo con el objeto “mesa No 1”. Por tanto, diremos que cada objeto es de un tipo particular y que este tipo indica los caracteres comunes a todos los estaos posibles de dicho objeto. Por ejemplo, dos de los objetos citados, la “mesa No 1” y la mesa siguiente, son del mismo tipo “mesa”; es decir, que representan, en cada instante, una de las mesas del grupo de 10 mesas del problema. Para describir un tipo, es suficiente con enumerar los diferentes estados que un objeto de este tipo puede adoptar:

Tipo “mesa”: {mesa No 1, mesa No 2, mesa No 3, mesa No 4, mesa No 5, mesa No 6, mesa No 7, mesa No 8, mesa No 9, mesa No 10}

Esta enumeración no siempre es práctica. Si, por ejemplo, decimos que los objetos “el primer número” y “3” son de tipo numérico, está claro que, para describir este tipo, hay que enumerar todos los valores de números posibles. Podremos entonces

definir el tipo numérico como el conjunto de todos los números que el procesador es capaz de tratar.

La última característica de la noción de objeto es el valor. Diciendo anteriormente que un objeto del tipo “mesa” representa a cada instante una de las 10 mesas, hemos supuesto implícitamente esta tercera característica. Todo objeto del entorno tiene un valor susceptible de cambio a lo largo de la ejecución de una acción. Si, en el ejemplo de “agrupar las fotocopias”, en el epígrafe anterior, se designan los paquetes por el número de hoja que le corresponde y si situamos en la etapa de trabajo representada en la figura siguiente, podemos decir que el objeto “el paquete de encima de la pila” tiene por valor “el paquete 8”, antes de ejecutar b), y el “paquete 3” después de la ejecución de b).



En relación a esto, observemos una particularidad importante en el cambio del valor de un objeto: a menos que hayamos anotado antes el valor precedente, es imposible recuperarlo ahora a partir del nuevo estado del entorno obtenido después de haber ejecutado la acción que atribuye el nuevo valor. Diremos que cada nuevo valor de un objeto destruye el valor precedente. Siguiendo con la noción de valor, vemos igualmente que ciertos objetos, tales como el objeto “3”, no cambia jamás de valor.

Para resumir, podemos decir que un objeto puede describirse como una caja con una etiqueta (su nombre), de una cierta forma (su tipo) y que contiene una información (su valor).

2.1.2- Ejemplo.

Supongamos que disponemos de un objeto llamado “NUM”, que contiene un valor numérico entero ‘n’ ($n \geq 1$). Se quiere elaborar un algoritmo que determine la suma de los ‘n’ primeros enteros naturales.

Para resolver el problema, es necesario conocer el procesador que estará encargado de ejecutar el algoritmo que va a ser elaborado: es necesario saber cuáles son las acciones primitivas que este procesador puede ejecutar. Supongamos que el procesador sabe.

a) dar el valor a un objeto;
b) calcular la suma de dos valores numéricos;
c) comparar dos valores numéricos y constatar su igualdad o diferencia;
d) ejecutar un esquema repetitivo del tipo “repetir..hasta que”.

Una vez planteado el problema, para proceder a su solución es necesario, en principio, describir con precisión el entorno en el que deberá trabajar el procesador. Este entorno comprende un objeto ya descrito: ‘NUM’. El valor inicial de este objeto está determinado y va a servir para controlar el cálculo: si este valor es 3, se calculará 1+2+3=6; si fuera 5; se calcularía 1+2+3+4+5=15; etc...El valor-resultado deberá figurar en el estado final del entorno. Necesitamos, para que sirva de soporte a este valor, un nuevo objeto. Llamémosle “S”. En este objeto acumularemos los enteros sucesivos. Estos enteros también deben estar representados. Para lo cual tenemos dos posibilidades:

– utilizar n objetos que contengan a cada uno de los valores enteros necesarios (1, 2, 3,....,n);
– bien, utilizar un solo objeto, cuyo valor haremos variar para hacerle representar todos los valores de 1 a n.

Además del inconveniente de necesitar muchos objetos, la primera solución presenta la gran desventaja de que el número de objetos del entorno depende de n. La gestión de un número variable de objetos es una operación particularmente delicada. Por esta razón, elegimos la segunda solución y llamaremos “ENTERO” al objeto utilizado.

Acabamos de definir (o sea, de crear) dos objetos numéricos: S y ENTERO. Un objeto tiene siempre, a cada instante, un valor, pero admitiremos que, en el momento de creado el objeto, su valor no puede ser conocido: diremos que es <u>indeterminado</u>. Si se desea dar a este objeto un valor determinado, es necesario ejecutar una acción.

Ya podemos dar una descripción del entorno del problema:

Objetos	Función	Estado inicial	Estado final
NUM	Objeto de valor entero ≥ 1 que indica el entero máximo a acumular.	N	n
S	Objeto en el que se calcula la suma de los n primeros enteros.	Valor indeterminado	$\sum_{i=1}^{n} i$
ENTERO	Objeto que contendrá los enteros sucesivos.	Valor indeterminado	

El valor final del objeto ENTERO no está indicado. No lo conoceremos hasta que hayamos elaborado el algoritmo, y así sabremos exactamente cómo el procesador utiliza a ENTERO.

Después de haber descrito el entorno, podemos estudiar la composición del algoritmo; es decir, determinar las acciones que se han de ejecutar para trasladar el entorno del estado inicial descrito al estado final deseado. Una idea bastante simple es dar a ENTERO los valores sucesivos de enteros de 1 a n y acumular estos valores en S. El primer entero a acumular es 1; para lo que necesitaremos inicialmente dar a ENTERO el valor 1. Igual que para obtener, por acumulación, el valor en S, es necesario dar un valor determinado a este objeto. Hemos admitido anteriormente que el valor de un objeto que acaba de ser definido es siempre indeterminado. Si 's' es el valor (indeterminado) de S en la creación de este objeto, una acumulación en S de todos los enteros de 1 a n no producirá el resultado esperado, sino el valor $s + \sum_{1}^{h} i$. Por tanto, es necesario, antes de la acumulación, dar a S el valor 0. Para cada valor de ENTERO, es suficiente con:

- acumular el valor de ENTERO en S,
- pasar al entero siguiente, repetir la acumulación, etc....hasta que se hayan acumulado todos los enteros necesarios.

Podemos entonces dar una solución:

algoritmo "suma de enteros"

1. dar a s el valor 0
2. dar a entero el valor 1

    repetir
    - 3a. Sumar el valor de ENTERO y el valor de S y asignar el resultado como nuevo valor de S.
    - 3b. Añadir 1 al valor de ENTERO y asignar el resultado como nuevo valor de ENTERO.

    hasta que el valor de ENTERO sea igual al de NUM incrementado en 1

En este algoritmo, todas las acciones representadas son acciones primitivas o composiciones de primitivas (3ª es una composición de las primitivas b): calcular la suma de dos valores y a): dar un valor a un objeto). Habiendo propuesto el algoritmo, ya podemos precisar el valor final de ENTERO; el que finalizará la repetición, por tanto: n + 1.

2.1.3- Ejecución paso a paso.

La importancia en el algoritmo elaborado anteriormente no está en ser utilizado una sola vez para calcular, por ejemplo, la suma de los 10 primeros enteros. Sería, en

este caso, más rápido calcular esta suma a mano, más que componer un algoritmo y utilizar un procesador. El algoritmo, de alguna forma, permite al procesador aprender a calcular la suma de los n primeros enteros, cualquiera que sea n, y está destinado a ser ejecutado por el procesador un gran número de veces para diferentes valores del objeto NUM.

Para entender cómo el procesador ejecuta el algoritmo, seguiremos una ejecución paso a paso, es decir, acción por acción, y lo haremos con el valor 3 para el objeto NUM.

Acción ejecutada	Estado del entorno después de la ejecución de la acción		
	NUM	S	ENTERO
inicialmente	3	indefinido	indefinido
1	3	0	indefinido
2	3	0	1
3ª	3	1	1
3b	3	1	2
3c (2≠3+1), continuar			
3ª	3	3	2
3b	3	3	3
3c (3≠3+1), continuar			
3ª	3	6	3
3b	3	6	4
3c (4≠3+1), la ejecución ha terminado			

Una ejecución así “a mano”, en la que el creador del algoritmo sustituye al procesador para ejecutar el algoritmo a partir de valores iniciales dados, es llamada a menudo “traza” o “rastrear el algoritmo” por los informáticos.

Observemos que una traza no sabría aportar la prueba de que el algoritmo es exacto. La traza del algoritmo anterior muestra que, para el valor 3 del objeto NUM, el algoritmo funciona correctamente. No se ha demostrado que fuera correcto para cualquier valor inicial de NUM. La única cosa que se puede probar eventualmente a partir de una traza es el mal funcionamiento del algoritmo: si el resultado obtenido con un seguimiento paso a paso es incorrecto, entonces seguro que el algoritmo es incorrecto.

2.2- Acciones y algoritmo.

Volvamos a la definición intuitiva de las nociones de acciones y de algoritmo analizadas en el epígrafe anterior y propongamos una fórmula más específica:

Dado un entorno descrito formalmente como un conjunto de objetos, una **acción** en este entorno es un suceso de duración finita que, a partir de un estado inicial particular del entorno, tiene como consecuencia un nuevo estado bien definido.

Un problema está planteado correctamente siempre que se haya descrito:

– lo que se ha de realizar (de dónde se parte: estado inicial del entorno; dónde se quiere llegar: estado final);
– el instrumento para la realización (el procesador del que se deben conocer las acciones primitivas).

Un **algoritmo** del problema es una sucesión de operaciones que permite transformar el entorno del estado inicial dado en el estado final deseado. Una sucesión tal que cada operación sea:

– una acción primitiva,
– bien un algoritmo ya conocido y descrito.

Es importante observar que, en un algoritmo, se puede llamar a otro algoritmo mejor que volver a formular las acciones primitivas. Por ejemplo, para los algoritmos siguientes:

algoritmo X	algoritmo y
a	a
c	b
y	b
d	a

si a, b, c, y d son acciones primitivas, diremos que la acción y en el algoritmo X es una llamada del algoritmo Y. Si se ejecuta el algoritmo X, las acciones ejecutadas estarán en el orden a, c, a, b, b, a, d.

2.3- Constantes y variables.

En el algoritmo “suma de enteros”, la acción 3b): “añadir 1al valor ENTERO y asignar el resultado como nuevo valor de ENTERO” hace intervenir dos objetos: ENTERO y 1. El objeto ENTERO varía su valor de 1 en 1 durante la ejecución del algoritmo; diremos que ENETERO es una variable, y podemos resumir de la siguiente manera las principales propiedades de una variable, extraídas de los párrafos anteriores:

Una **variable** es un objeto cuyo **valor** no es invariable. Además de este valor, toda variable posee otros dos atributos:

– un **nombre** (invariable) que sirve para designarla,
– un **tipo** (invariable) que describe la utilización posible de la variable.

Cuando se refiere una variable, se ha de precisar su nombre y su tipo. Definir una variable es, de hecho, crear un objeto para el procesador. Las reglas de creación de una variable no permiten conocer con antelación el valor que tendrá en el momento de su creación. Antes de examinar estas reglas, aceptamos desde ahora que toda variable que acaba de ser definida tiene un valor indeterminado.

Volviendo a la acción 3b) citada anteriormente, analicemos ahora el objeto 1. Es un objeto particular, indispensable al entorno del algoritmo “suma de enteros”, pero cuyo valor es invariable. Este valor también es utilizado para designar el objeto, que no tiene otro nombre. Entonces diremos que 1 es una constante.

Una **constante** es un objeto de valor invariable. Es la realización de un valor de un tipo particular.

La distinción que acabamos de hacer entre la noción de constante y la de variable puede llevar a preguntarnos en qué categoría debe clasificarse un objeto como NUM, utilizado también en el algoritmo “suma de enteros” y que a lo largo de toda la ejecución conserva su valor inicial. De hecho, NUM es una variable, y ya veremos que esta variable se ha considerado como un parámetro del algoritmo. En efecto, nada impide a un usuario del algoritmo hacer variar a NUM entre dos ejecuciones de este mismo algoritmo. Así, la secuencia de acciones:

- dar a NUM el valor 10;
- ejecutar el algoritmo “suma de enteros”;
- dar a NUM el valor 15;
- ejecutar el algoritmo “suma de enteros”,

permite, con dos ejecuciones del mismo algoritmo, calcular sucesivamente dos sumas diferentes. Si, en el algoritmo “suma de enteros”, hubiéramos utilizado una constante en lugar de NUM, entonces hubiéramos fijado para siempre su valor. El algoritmo que elaborado entonces nunca habría podido ser utilizado para calcular otra suma que la determinada por la constante. Por ejemplo, de haber utilizado la constante 7 en el lugar de NUM hubiera calculado siempre la suma de los 7 primeros enteros.

2.3.1- Tipos elementales.

El tipo numérico es el conjunto de los valores numéricos que el procesador sabe tratar. Un valor numérico será escrito en su forma habitual con o sin signo (un valor sin signo está considerado como positivo): 10, 12.5, +22, -125.34 son valores numéricos correctamente escritos.

Más adelante veremos que el modo de representación de una variable siempre limita la capacidad de la misma, a la que es imposible dar valores en los que el valor absoluto es demasiado grande. Se puede entender este problema, observando que, con una máquina de escribir que marque dos cifras por

centímetro, no se pueden escribir números de más de 10 cifras en un espacio de 5 centímetros.

El tipo carácter es el conjunto de las cadenas de caracteres que se pueden formar a partir de los elementos del conjunto de caracteres (letras, cifras y signos especiales) que el procesador reconoce. Para evitar confundir una cadena de caracteres con el nombre de una variable, la representamos siempre entre dos apóstrofes. Estos dos apóstrofes delimitan el principio y final de la cadena y no se considera que formen parte de la cadena. Así: 'lista', 'APELLIDO, NOMBRE', TOTAL A FACTURAR', son tres cadenas de caracteres. La primera está formada por 5 letras minúsculas l, ni, s, t, a. La segunda contiene 15 caracteres (14 letras mayúsculas y una coma). La última contiene 16 caracteres (14 letras y dos espacios); el espacio que permite separar las palabras en una cadena también es un carácter.

Una variable de tipo carácter tiene valores que son cadenas de caracteres. En lo adelante supondremos que una variable de este tipo siempre tiene una capacidad suficiente para acoger al valor que se le deba dar.

El tipo lógico es el conjunto de dos valores lógicos, cierto y falso. Una variable de tipo lógico siempre posee uno de estos dos valores.

2.3.2- Variables compuestas.

Experimentaremos la necesidad de agrupar varios valores para caracterizar una entidad particular. Definiremos entonces una variable compuesta, asociación de varias variables elementales. Por ejemplo, si disponemos de una nota obtenida en un examen por un estudiante, para una asignatura dada, podremos colocar estas informaciones en la variable compuesta RESULTADO:

NOMBRE	ASIGNATURA	NOTA
'FERMÍN'	'INGLÉS'	6

En el ejemplo, la variable RESULTADO está compuesta de 3 variables elementales:

- NOMBRE, variable de tipo carácter, cuyo valor, en la situación representada por el esquema, es 'FERMÍN',
- ASIGNATURA, variable de tipo carácter, cuyo valor es 'INGLÉS',
- NOTA, variable de tipo numérico, cuyo valor es 6.

Diremos que las variables NOMBRE, ASIGNATURA y NOTA son subdivisiones de la variable compuesta RESULTADO. Una subdivisión puede ser ella misma una compuesta. Por ejemplo:

ESTABLECIMIENTO		
NOMBRE	DIRECTOR	DIRECCIÓN

‘floristería dalia’	‘bertha’	CALLE	CIUDAD
		‘calle del mercado’	‘PARIS’

La subdivisión DIRECCIÓN de la variable ESTABLECIMIENTO comprende las subdivisiones CALLE y CIUDAD.

La designación de una subdivisión puede entrañar a veces una ambigüedad. Por ejemplo, si suponemos que las 2 variables RESULTADO y ESTABLECIMIENTO forman parte de un mismo entorno, la acción “imprimir el valor de la variable NOMBRE” no es suficientemente explícita, ya que se ignora si la subdivisión designada es la de RESULTADO o la de ESTABLECIMIENTO. Cada vez que haya riesgo de ambigüedad, designaremos una subdivisión concatenando su nombre al de la variable a la cual está subordinada:

- RESULTADO.NOMBRE designa la subdivisión NOMBRE de RESULTADO,
- ESTABLECIMIENTO.DIRECCIÓN.CALLE designa la subdivisión CALLE de la subdivisión DIRECCIÓN de la variable ESTABLECIMIENTO.

2.4- Acción de asignación.

Después de haber aclarado la noción de objeto, vamos a elegir una fórmula precisa para describir las acciones que permiten asignar un valor a una variable.

2.4.1- Asignación interna.

En un entorno dado, para atribuir un valor a una variable que provenga de este mismo entorno, utilizaremos la notación V← E en la que:

V: es el nombre de la variable a la que el procesador debe atribuir el valor,
←: caracteriza la acción de asignación,
E: representa el valor a asignar y puede ser:

- Una constante,
- El nombre de otra variable que contenga el valor,
- Una expresión aritmética describiendo un cálculo a efectuar.

Observemos que E sólo puede ser una expresión aritmética si V es una variable de tipo numérico. Es importante tener en cuenta que las dos entidades que, en una asignación, aparecen a ambos lados del signo ← deben ser siempre del mismo tipo: por ejemplo, no se puede asignar un valor lógico a uan variable numérica.

Estudiemos las 4 acciones a, b, c y d:

a) ENTERO ← 1
b) NOMBRE ← ‘ANDRÉS’
c) TOTAL ← SUMA

d) NUM ← A + B

La acción a) asigna a la variable ENTERO el valor 1. Para que la acción b) sea correcta, es necesario que la variable NOMBRE sea de tipo carácter: la acción b) le asigna la cadena 'ANDRÉS'. En las acciones a) y b) el valor a asignar se indica por medio de una constante.

La acción c) representa una asignación en la que el valor a asignar está indicado por una variable. Las dos variables TOTAL y SUMA que figuran a cada lado del signo ← deben ser del mismo tipo. Si, antes de la ejecución de c), TOTAL y SUMA tienen los valores respectivos t y s, después de la ejecución de c) TOTAL habrá tomado el valor s y SUMA conservará su valor: sólo la variable cuyo nombre aparece a la izquierda del signo ← cambia de valor.

La acción d) asigna a la variable NUM el resultado de un cálculo aritmético. Las variables NUM, A y B deben ser de tipo numérico. La acción d) se ejecuta en dos tiempos:

- cálculo del valor de la expresión aritmética (suma de valores de A y B),
- asignación de este valor a la variable NUM.

Para formar expresiones aritméticas, utilizaremos las constantes o las variables numéricas y los operadores:

+	suma
-	resta
*	multiplicación
/	división
div	división euclidiana

La manera en que se ejecuta una asignación aritmética (1: cálculo, 2: asignación del resultado) entraña que una acción como:

ENTERO = ENTERO + 1

En la que ENTERO es una variable numérica, tenga un sentido y represente un aumento en 1 del valor de ENTERO. Una misma variable puede, por tanto, aparecer a ambos lados del signo ←; el valor que representa a la derecha del signo de asignación es el que tiene antes de la ejecución de la acción.

Desde ahora se puede constatar entonces la ventaja que aporta esta nomenclatura con relación a una formulación en lenguaje natural: rescribamos el algoritmo "suma de enteros", presentado anteriormente, expresando las asignaciones con la ayuda del formalismo elegido:

s ← 0

entero ← 1
repetir
  s ← s + ENTERO
  ENTERO ← ENTERO + 1
hasta que el valor de ENTERO sea igual al de NUM incrementado en 1.

2.4.2- Entrada de un dato.

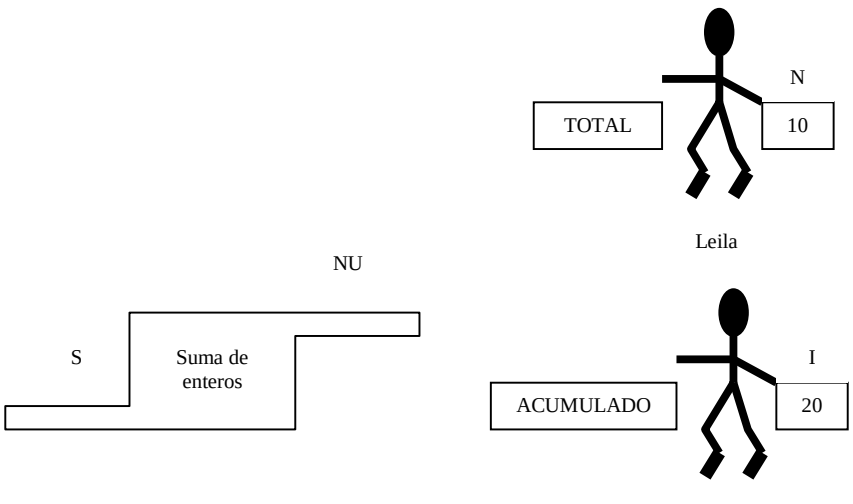
Puede ocurrir que un valor a entrar no forme parte del entorno de un trabajo: para poder registrar este valor, será necesario introducirlo en el entorno. Así, con una reserva mecanizada de avión, para la cual es necesario conocer los nombres de los pasajeros, un cliente sólo es conocido cuando "aborda" el entorno pasando por o telefoneando a la oficina de reservas. El nombre de este cliente penetra en el entorno cuando se marca en el teclado de la terminal de reserva: este nombre se convierte en el valor de una variable del entorno. Llamamos lectura a toda operación de este tipo que introduce un valor en un entorno por mediación de una aparato. Una lectura es una asignación: toma el valor y lo asigna a una de las variables del entorno. Consideraremos que toda lectura es una acción primitiva y, si v es una variable, indicaremos por: leer v a toda acción de lectura que introduce un valor en el entorno para asignarlo a v.

2.4.3- Salida de un dato.

También nos encontraremos con que, a menudo, hay que hacer "salir" un valor del entorno para comunicar una información al exterior (representación en una pantalla, impresión sobre el papel...). Se trata entonces de comunicar al exterior, por medio de un aparato, e l valor de una variable. Llamamos escritura a una operación tal que consideramos siempre como una acción primitiva. Indicamos por: escribir v a la acción que permite comunicar al exterior el valor de la variable v. Hay que insistir en el hecho de que esta acción, que transmite al exterior una copia del valor v, no modifica de ninguna forma el valor de la variable. La acción "leer v" asigna un valor a v, la acción "escribir v" no cambia el valor de v.

2.5- Parametrización de una algoritmo.

En la figura siguiente:



Se han presentado 3 personas deseosas de utilizar el algoritmo “suma de enteros”. Recordemos que este algoritmo calcula en la variable S la suma de los n primeros enteros naturales, siendo n el valor (se supone entero) de la variable NUM.

Teniendo en cuenta sus necesidades, cada una de estas 3 personas se prepara para utilizar el algoritmo con diferentes herramientas: Leila quiere obtener en una variable TOTAL la suma de los 10 primeros enteros (10 es el valor de una variable N), Miguel quiere la de los 20 primeros enteros en ACUMULADO, y Ana la de lso 8 primeros enteros en SUMA. Se pueden resumir las operaciones que serán efectuadas de la siguiente manera:

Leila	Miguel	Ana
NUM ← N	NUM ← I	NUM ← V
Ejecutar el algoritmo	Ejecutar el algoritmo	Ejecutar el algoritmo
TOTAL ← S	ACUMULADO ← S	SUMA ← S

Para simplificar estas operaciones, vamos a parametrizar el algoritmo a fin de permitirle trabajar directamente sobre las variables utilizadas por los diferentes usuarios. Para ello, rescribamos el algoritmo de la siguiente manera:

Algoritmo SUMAENTEROS (NUM, S)

inicio
    S ← 0
    ENTERO ← 1
    repetir

S ← S + ENTERO
ENTERO ← ENTERO + 1
hasta que el valor de ENTERO sea igual al de NUM incrementado en 1.
fin

¿Qué hemos cambiado con respecto a la presentación anteriormente analizada?. De entrada, y esto no tiene que ver con la noción de parametrización, hemos limitado la secuencia de acciones a ejecutar con las palabras *inicio* y *fin*: en lo sucesivo lo haremos sistemáticamente para cada algoritmo presentado. Hemos dado un nombre al algoritmo (SUMAENTEROS) acompañado de los nombres de las 2 variables(NUM y S) entre paréntesis. Con esto indicamos que NUM y S son parámetros y que el algoritmo es capaz de ejecutarse haciendo jugar a otras variables los papeles de NUM y S. Así, para obtener lo que se desea, las 3 personas sólo tendrán que hacer ejecutar cada una de ellas una acción parametrizada:

Leila: SUMAENTEROS (N, TOTAL)
Miguel: SUMAENTEROS (I, ACUMULADO)
Ana: SUMAENTEROS (V, SUMA)

Diremos que cada una de las 3 acciones anteriores es una llamada al algoritmo SUMAENTEROS. La llamada de Leila tiene como parámetros N y TOAL, lo cual provoca la ejecución del algoritmo en un entorno en el que la variable N corresponde a la variable NUM y la variable TOTAL corresponde a ña variable S.

¿Qué diferencia se puede hacer entre los parámetros, tales como NUM y S y aquellos de la acción utilizada por Leila?. Diremos que NUM y S son los parámetros formales del algoritmo y que N y TOTAL son los parámetros actuales de una llamada a este tipo de algoritmo.

Un parámetro formal es una variable elegida como parámetro en la definición de un algoritmo. Un parámetro actual es una variable utilizada en una llamada en lugar de un parámetro formal. En la ejecución de una llamada, la correspondencia entre el parámetro actual y el parámetro formal es definida por la posición: el primer parámetro actual corresponde al primer parámetro formal, el segundo parámetro actual corresponde al segundo parámetro formal, etc...

De esto resulta que cada llamada debe comprender tanto parámetros actuales como parámetros formales hay en la definición del algoritmo. Tenemos que añadir que cada parámetro actual debe ser del mismo tipo que el parámetro formal correspondiente. El modo de establecer la correspondencia en el momento de una llamada puede hacer intervenir varias técnicas distintas, pues la ejecución de una llamada obliga al algoritmo a trabajar directamente con los parámetros formales.

Distinguiremos 3 categorías de parámetros: parámetros-datos, parámetros-resultados y parámetros-datos/resultados. Un parámetros de un algoritmo es un

parámetro-dato cuando su valor inicial es utilizado por el algoritmo y queda invariable después de la ejecución de este algoritmo. Un parámetro-resultado es un parámetro en el que el valor inicial no tiene significado para el algoritmo; la ejecución del algoritmo asigna a un parámetro tal un nuevo valor. En el algoritmo SUMAENTEROS, NUM es un parámetro-dato y S un parámetro-resultado. Un parámetro-dato/resultado es una parámetro cuyo valor inicial es utilizado por el algoritmo, pero es remplazado al final de la ejecución por un valor-resultado.

Observemos que el valor inicial de un parámetro-resultado no influye en el desarrollo del algoritmo. En la llamada de Leila (SUMAENTEROS (N, TOTAL)), si N tiene el valor 10 antes de ejecutar la llamada, el parámetro-resultado TOTAL tomara el valor 55 (= 1+2+...+10) después de la ejecución de la llamada, independientemente de su valor inicial.

Supongamos un usuario que quiere utilizar el algoritmo SUMAENTEROS para calcular la suma de los 15 primeros enteros y que dispone, para ello, de una variable numérica K, de valor 15, y de otra N de valor 4. Para obtener el resultado deseado en N, la llamada correcta es SUMAENTEROS (K, N) que remplazará el valor inicial de N por la suma esperada. Pero, si el usuario efectúa la llamada SUMAENTEROS (K, N) el algoritmo tratará a N como parámetro-dato y a K como parámetro-resultado: no modificará a N y calculará en K la suma de los 4 primeros enteros.

Hemos convenido, hasta ahora, que la ejecución de una llamada hace trabajar al algoritmo directamente sobre los parámetros actuales. Esto puede tener consecuencias sorprendentes si un mismo parámetro actual corresponde, al mismo tiempo, a un parámetro-dato y a un parámetro-resultado. Consideremos, por ejemplo, la llamada SUMAENTEROS (K, N) y supongamos que antes de la llamada N tiene el valor 10. En la ejecución de la llamada, N juega a la vez el papel de NUM y de S, y conforme a la descripción antes analizada, se ejecutan 4 acciones:

N ← 0,
ENTERO ← 1,
N ← N + ENTERO que da a N el valor 1,
ENTERO ← ENTERO + 1 que da a ENTERO el valor 2.

La ejecución de la llamada se termina, ya que la condición de parada de la repetición se cumple (el valor de ENTERO es igual al de N incrementado en 1). No se obtiene, por consiguiente, en N la suma de los 10 primeros enteros. Para evitar este tipo de incidentes, vamos a impedir, desde ahora, que un mismo parámetro actual corresponda a varios parámetros formales, salvo en el caso de que todos los parámetros formales sean parámetros-dato. Por el mismo motivo, no utilizaremos nunca una constante como parámetro formal.

A la noción de parámetro se opone la de variable interna. Entre las variables que forman parte del entorno de un algoritmo, sólo tienen importancia para el usuario

aquellas que son parámetros y son conocidas por él. Llamaremos variables internas a cualquier otra variable. En el algoritmo SUMAENTEROS, la variable ENTERO es una variable interna.

3- ESQUEMAS CONDICIONALES.

3.1- Noción de predicado.

Sea L el conjunto de los adjetivos cierto, falso. Dado un conjunto E, una función proposicional o predicado es una aplicación de E en el conjunto L. Un predicado, denominado propiedad, es un enunciado cierto para algunos elementos de E y falso para el resto. Daremos un ejemplo:

Sea N el conjunto de los enteros naturales. Se define en N la siguiente función proposicional: “x es un múltiplo de 3”. Esta función adopta el valor cierto para algunos valores de x (3, 6, 9, ...) y el valor falso para los otros (5, 7, 11, ...).

En programación, encontraremos predicados cuyo conjunto de partida es el conjunto de los valores tomados por una o más variables. Así, si X e Y son variables numéricas, el enunciado “mayor que 3” (X>3), es un predicado cierto para algunos valores de X, falso para el resto. Igual que “X mayor que Y” (X>Y) es un predicado.

3.2- Cálculo de un predicado.

En programación, existen numerosas operaciones dependientes de la evaluación de una condición, es decir, del valor de un predicado. En el tema 1, hemos propuesto un algoritmo para realizar la agrupación de las fotocopias. Hemos utilizado esquemas repetitivos en los que la condición que detenía la repetición era de hecho un predicado (por ejemplo, “hasta que una de las mesas esté vacía” es una aplicación del conjunto de los estados de las mesas en el conjunto L). Igualmente la comparación de ENTERO con el valor de (NUM + 1) utilizada en el algoritmo SUMAENTEROS del tema 2 es un predicado que aplica el conjunto de pares de valores tomados por ENTERO y NUM en L.

En la composición de algoritmos, expresaremos los predicados que después serán evaluados por el procesador en el momento de la ejecución del algoritmo. Para expresar un predicado, escribiremos, a menudo, una comparación entre dos valores del mismo tipo. Una comparación tal se llamará predicado elemental. Para las comparaciones de valores de tipo numérico o carácter utilizaremos =, <, ≤, >, ≥, ≠, como operadores de comparación y, para las comparaciones de valores lógicos, utilizaremos los operadores = o ≠.

Las comparaciones de tipo numérico se expresan sencillamente. Por ejemplo, si X e Y son dos variables numéricas cuyos valores son respectivamente 4 y 20, la tabla siguiente da los valores respectivos de algunos predicados:

Predicado	Valor
X = Y	Falso
X > 3	Cierto
Y < X	Falso

La comparación de los caracteres es posible en la medida en que, en el conjunto de los caracteres utilizados, existe una relación de orden definida por el orden alfabético: ‘a’ < ‘b’ < ...< ‘z’ < ‘A’ < ‘B’ < ... < ‘Z’. Para las cadenas de caracteres, la comparación se efectúa de izquierda a derecha, de la misma manera que en un diccionario. Así, se puede decir que la constante ‘THOMAS’ es menor que la constante ‘TOMATE’ (aparece antes en el diccionario). De la misma manera, si NOMBRE es una variable de tipo carácter, el predicado NOMBRE < ‘PIERRE’ se evaluará comparando el valor de la variable NOMBRE con la constante ‘PIERRE’. Por ejemplo, en el caso de que NOMBRE tenga el valor ‘JEANNE’, el predicado es cierto; si este valor es un ‘URSULA’, el predicado es falso.

Cuando la utilización de comparaciones no es suficiente para expresar una situación, se utilizan los conectores no, y, o, para formar, a partir de predicados elementales, predicados compuestos.

La evaluación de predicados compuestos se apoya en leyes de lógicas (álgebra booleana); los lectores no familiarizados con las reglas de cálculo booleano encontrarán en un anexo algunas nociones de lógica.

En el ejemplo tratado en el tema 1, relacionado con la agrupación de fotocopias, el predicado expresado en la condición “hasta que una de las mesas que sin hojas” es, de hecho, un predicado compuesto. Se puede expresar de la forma:

> “hasta que la mesa 1 quede sin hojas o la mesa 2 quede sin hojas o ... o la mesa 10 quede sin hojas”.

Hay que tener en cuenta la importancia de los paréntesis que a veces es necesario utilizar en la expresión de un predicado compuesto. Así, el predicado:

(X = 1 o Y = 2) y Z > 3 es diferente del predicado X = 1 o (Y = 2 y Z > 3).

Demostrémoslo evaluando estos dos predicados para los valores de X, Y, Z respectivamente a 1, 3, 2:

X =1	Y = 2	Z > 3
cierto	falso	falso
X = 1 o Y = 2		
cierto o falso		
cierto		
cierto y falso		

falso

X =1	Y = 2	Z > 3
cierto	falso	falso
	falso y falso	
	falso	
cierto o falso		
cierto		

Por tanto, estos dos predicados no son equivalentes.

También hay que tener en cuenta que una condición que matemáticamente se escribe A < B < C (A, B y C son tres variables), en programación se denotará A < B y B < C. La mayor parte de los procesadores no aceptan expresiones tales como A < B < C.

3.3- El esquema condicional.

A partir de un ejemplo, vamos a ilustrar lo que se entiende, en programación, por esquema condicional. Consideremos el siguiente enunciado que describe la regla de cálculo de un salario:

a: calcular el salario bruto

Si el empleado es ingeniero, entonces añadir la prima de técnico

p b

c: añadir la prima antigüedad

La acción b, “añadir la prima de técnico”, sólo es ejecutada cuando la condición expresada por medio del predicado p, “empleado es ingeniero”, se cumple, es decir, cuando el predicado p es cierto. El enunciado puede expresarse:

a; si p entonces b; c

En el caso en que el empleado es ingeniero, las acciones a, b, c serán ejecutadas. En caso contrario, sólo lo serán a y c.

Modifiquemos, en el enunciado anterior, la frase condicional remplazándola por:

Si el empleado es ingeniero, entonces añadir la prima de técnico

p b

sino pagar las horas extras

p’

La secuencia de acciones es ahora la siguiente:

a; si p entonces b sino b'; c

Si el empleado es ingeniero, las acciones a, b y c serán ejecutadas. En el caso contrario, se ejecutará a, b' y c.

Para hacer ejecutar a un procesador operaciones tales como las que acabamos de dar en el ejemplo. Será necesario disponer de esquemas condicionales que permitan expresar que una secuencia de acciones sólo se ha de ejecutar si se cumple una cierta condición. La condición será representada por un predicado, y la forma de los esquemas que utilizaremos puede definirse a partir del ejemplo. Antes de precisar estos esquemas, veamos que es necesario delimitarlos correctamente. Así en el enunciado:

a si p entonces b c d

para el cual p es un predicado y a, b, c y d, acciones, no es posible determinar cuáles son las acciones que se han de ejecutar cuando p es cierto o cuando lo es: las acciones c y d ¿forman o no parte del esquema condicional?. Con el fin de evitar toda ambigüedad, utilizaremos los delimitadores.

si p
entonces
a
finsi

si p
entonces
a
sino
b
finsi

En el primer caso, el delimitador *finsi* indica el final de la secuencia de acciones relativas al apartado *entonces*. En el segundo caso, ésta es delimitada por el *sino*. En los esquemas precedentes, a y b pueden ser secuencias de acciones y algunas de estas acciones pueden, a su vez, expresarse por medio de esquemas condicionales. Tenemos un anidamiento de estos esquemas. Damos un ejemplo en el que la parte *sino* de un primer esquema contiene una primera acción seguida de un segundo esquema condicional.

```
si     p
    entonces                    predicados          acciones
    ejecutadas
        a                       p       q
    sino                        cierto  cierto          a
        b                       cierto  falso           a
    si      q                   falso   cierto          b, c
        entonces                falso   falso          b, d
            c
        sino
            d
    finsi
finsi
```

Si no presenta ningún problema para le procesador que lo trata, un anidamiento de esquemas condicionales puede representar, sin embargo, dificultades de comprensión para una persona que lo lea, en particular un principiante en programación. Por lo tanto, no creemos inútil insistir en lagunas evidencias. Para una secuencia de acciones que comprende un esquema condicional, diremos que una acción, es exterior al esquema si aparece antes del si (es anterior) o después del *finsi* (es posterior). Si dicha acción está incluida en la parte entonces o en la parte sino, es interior al esquema condicional. Para un esquema condicional definido a partir de un predicado p, la ejecución de toda acción exterior es incondicional con relación a p: el hecho de ella se produzca o no, no depende del valor de p. La ejecución de toda acción interior es condicional con relación a p. Apliquemos las observaciones anteriores al ejemplo:

```
a { X ← K + L
S  si        X ≠ Y
   entonces
        Y ← Z * Y
        Z ← 0
   sino
        b { Y ← 0
        s  si      Z = X
           entonces
                Z ← 0
                X ← 2 * X
           finsi
        c { X ← X + Z
   finsi
d { Z ← Z + K
```

En este ejemplo, todas las variables utilizadas son de tipo numérico. La parte sino del esquema condicional S contiene otro esquema condicional s. Las acciones b y c son interiores a S, pero exteriores a s. Serán ejecutadas si el valor de X después

de la ejecución de a es igual al de Y, y esto para cualquiera que sea el valor del predicado Z = X.

La siguiente tabla describe dos ejecuciones distintas del conjunto de la secuencia a partir de dos estados iniciales diferentes:

	Estado inicial 1	Estado inicial 2
Valores iniciales de las variables	X : 2 Y : 5 Z : 4 K : 7 L : 3	X : 2 Y : 5 Z : 6 K : 2 L : 3
Acciones ejecutadas	X ← K + L Y ← Z * Y Z ← 0 Z ← Z + K	X ← K + L Y ← 0 X ← X + Z Z ← Z + K
Valores finales	X : 10 Y : 20 : 7	X : 11 Y : 0 Z : 8

En esta tabla, una expresión como X:2 debe leerse como “X tiene el valor 2” y considerarse como una afirmación. La utilización del signo “:” preferentemente al signo = evita que confundamos esta afirmación con el predicado X = 2.

3.4- Lógica del esquema condicional.

3.4.1- Lógica de una acción.

Hemos visto que una acción modifica el entorno al que se aplica. Es posible caracterizar esta modificación describiendo el entorno antes y después de la ejecución de la acción considerada. Para esto, se formulan dos afirmaciones. La primera llamada antecedente, describe un estado posible del entorno antes de la ejecución de la acción; la segunda, el consecuente, describe el nuevo estado del entorno si la acción ha sido ejecutada cuando la primera afirmación era cierta. Hay que observar que existen varios posibles estados del entorno antes de la ejecución de una acción, un estado dado que depende de las acciones ejecutadas anteriormente. Por tanto, se concibe que igualmente hay varios consecuentes.

Se anotarán el antecedente y el consecuente entre llaves. Se escribirá, por ejemplo:

{X : 4}	x ← X + 2	{X : 6}
el antecedente	la acción	el consecuente

De forma general, se utilizará la notación siguiente:

{antecedente} acción {consecuente}

3.4.2- Lógica del esquema condicional.

Expresar la lógica de un esquema condicional es afirmar que el predicado es cierto si se encuentra en el apartado *entonces* del esquema y falso si se encuentra en el apartado *sino*.

Sea S el esquema:

si p *entonces* a *sino* s *finsi*

Si para las acciones a y s tenemos los antecedentes y consecuentes:

{p}	a	{q}
{no p}	s	{r}

entonces se puede escribir:

{p}	*S*	*{q}*
{no p}	S	{r}

En función de los antecedentes {p} o {no p}, la ejecución del esquema S conduce a {q} o {r}. Veamos un ejemplo:

Problema: Una variable numérica tiene un valor que comporta a lo sumo tres decimales. Se quiere poder determinar si este valor es entero; es decir, calcular el predicado: “la variable es entera”.

Principio básico: Si X es la variable a examinar, el algoritmo responderá por medio de una variable lógica XENTERO, que tomará el valor cierto si el valor de X es entero y falso en caso contrario. El valor de X contiene, a lo sumo, tres decimales; para separar estas tres cifras se multiplicará el valor de X por a000 (se obtiene un valor entero), a continuación se divide este entero por 1000 (división euclidiana) y se calcula el resto de esta división; si el resto es nulo, es que el valor de X es entero.

```
algoritmo EXAMEN (X, XENTERO)

inicio
    P ← X * 1000
    Q ← P div 1000
    R ← P – (1000 * Q)
    si R = 0
        entonces XENTERO ← cierto
        sino XENTERO ← falso
    finsi
fin
```

Escribamos ahora la lógica de cada acción:

algoritmo EXAMEN (X, XENTERO)

inicio

$\{ X : n + d * 10^{-3},\ n \text{ y } d \text{ enteros},\ 0 \le d < 1000 \}$ $P \leftarrow X * 1000$ $\{ P : n * 10^3 + d,\ 0 \le d < 1000 \}$

$\{ P : n * 10^3 + d,\ 0 \le d < 1000 \}$ $Q \leftarrow P \text{ div } 1000$ $\{Q : n\}$

$\{ P : n * 10^3 + d,\ Q : n \}$ $R \leftarrow P - 1000 * Q$ $\{ R : d,\ 0 \le R < 1000 \}$

{R : d} *si* R = 0

entonces {R : 0} XENTERO ← cierto {XENTERO : cierto y d : 0}

sino {R ≠0} XENTERO ← falso {XENTERO : falso y d ≠0}

finsi

fin

Por la lectura de los dos consecuentes posibles para el esquema condicional que termina el algoritmo, se puede constatar que el algoritmo está verificado.

3.5- Enunciado condicional generalizado.

La utilización de esquemas condicionales anidados es pesada y difícil de releer cuando nos encontramos ante un gran número de anidamientos. Demos un ejemplo.

Se dispone de un cierto número de algoritmos que queremos ejecutar en función del valor de una variable DIA de tipo carácter, de acuerdo con la tabla siguiente:

valor de DIA	algoritmos a ejecutar
‘LUNES’	FACTURACIÓN
‘MARTES’	RECEPCIÓN
‘MIÉRCOLES’	PEDIDOS
‘JUEVES’	CONTABILIDAD
‘VIERNES’	PROVEEDORES
Otro valor distinto de los anteriores	ERROR

La utilización dele esquema condicional conduce a la siguiente secuencia algorítmica:

```
inicio
si DIA = 'LUNES'
    entonces FACTURACIÓN
    sino si DIA = 'MARTES'
            entonces RECEPCIÓN
            sino si DIA = 'MIÉRCOLES'
                    entonces PEDIDOS
                    sino si DIA = 'JUEVES'
                            entonces CONTABILIDAD
                            sino si DIA = 'VIERNES'
                                    entonces PROVEEDORE
                                    sino ERROR
                                finsi
                        finsi
                finsi
        finsi
finsi
fin
```

El anidamiento de *si* encadenados es, por una parte, fastidioso de escribir y, por otra, difícil de releer. Utilizaremos en este caso el esquema condicional generalizado que se presenta a continuación.

```
a
opción V de
V1 : b1
V2 : b2
.
.
.
vi : bi
.
.
vn : bn
otros ba
finopción
c
```

En este esquema, V es una variable y b1, b2...bn, ba son acciones o secuencias de acciones. En función del valor (v1, v2,... o vn) de V, una y sólo una de las acciones b1, b2...bn será ejecutada. Si, por ejemplo, V tiene el valor vi, las acciones a, bi y c serán ejecutadas. Si el valor de V no pertenece al conjunto {v1, v2,... o vn}, se ejecutará ba. El esquema generalizado aplicado al ejemplo anterior se da a continuación.

opción DIA *de*
    ‘LUNES’ : FACTURACIÓN
    ‘MARTES’ : RECEPCIÓN
    ‘MIÉRCOLES’ : PEDIDOS
    JUEVES’ : CONTABILIDAD
    ‘VIERNES’ : PROVEEDORES
    otros : ERROR
finopción

En la práctica, se utiliza el enunciado generalizado del esquema condicional cuando nos encontramos ante una elección múltiple de secuencias de acciones a ejecutar en función del valor de una variable.

ANEXO: Aproximación a la lógica.

Proposiciones.

Dado el conjunto L = {cierto , falso}, una frase cualquiera es una proposición si se le puede atribuir uno y sólo uno de los elementos de L.

Consideremos las frases siguientes:

1. El número 3 es más grande que el número 1.
2. Nunca llueve en Francia.
3. La Tierra es redonda.
4. Se debe conducir por la derecha.
5. El cubo de un número X es positivo.

Las tres primeras frases son proposiciones: las frases 1 y 3 son ciertas, la frase 2 es falsa. Por el contrario, las frases 4 y 5 no son proposiciones; ya que a veces son ciertas y a veces falsas.

Conectores lógicos.

Se pueden componer proposiciones por medio de conectores lógicos. Existen tres conectores lógicos elementales, la conjunción (y), la disyunción (o, no exclusivo) y la negación (no). Consideremos las frases siguientes para comprender la función de un conector lógico:

“Si Luis ha acabado su trabajo y si tiene suficiente dinero, irá al cine”.

Esta frase está constituida por tres proposiciones P1, P2 y P3. Las proposiciones P1 y P2 están enlazadas por la conjunción ‘y’; ésta permite expresar el hecho de que P3 es cierto (Luis va al cine) si y solamente si P1 y P2 son ambas ciertas. Si P1 es falsa, o si P2 es falsa, o si las dos proposiciones son falsas, P3 es entonces falsa.

La frase siguiente ilustra la función del conector ‘o’:

> “El barco partirá mañana si hay al menos veinte pasajeros o si hace buen tiempo”.

Las dos condiciones no se han de cumplir obligatoriamente: el barco partirá si se cumple una u otra de las condiciones (o las dos).

Tablas de verdad.

Para conocer los valores en L = {cierto , falso} tomados por una proposición resultante de la composición de varias proposiciones, se puede recurrir a una tabla de verdad; se determinan los valores tomados por una proposición para todos los valores posibles de las proposiciones que la componen.

Damos aquí las tablas de verdad de los conectores lógicos no, y, o.

p	no p
cierto	falso
falso	cierto

p	q	p y q
falso	falso	falso
falso	cierto	falso
cierto	falso	falso
cierto	cierto	cierto

p	q	p o q
falso	falso	falso
falso	cierto	cierto
cierto	falso	cierto
cierto	cierto	cierto

No hay que confundir el o (no exclusivo, p o q), del cual hemos hablado anteriormente y del que hemos dado la tabla de verdad, con el o exclusivo indicado por o bien (o exclusivo). Si p y q son dos proposiciones, (p o bien q) es cierto si y sólo si una de las dos proposiciones p y q es cierta. La tabla de verdad del o exclusivo es la siguiente:

p	q	p o bien q
falso	falso	falso
cierto	falso	cierto
falso	cierto	cierto
cierto	cierto	falso

Las leyes de la lógica.

Los conectores lógicos elementales obedecen a leyes (asociativa, conmutativa, distributiva...), de las cuales las principales han sido resumidas en la siguiente tabla:

1. no (no p) $\Leftrightarrow$ p
2. p o (no p) siempre es cierto
3. p y q $\Leftrightarrow$ q y p
4. p o q $\Leftrightarrow$ q o p

5. (p y q) y r ⇔ p y (q y r)
6. (p o q) o r ⇔ p o (q o r)
7. p y (q o r) ⇔ (p y q) o (p y r)
8. p o (q y r) ⇔ (p o q) y (p o r)
9. no (p y q) ⇔ (no p) o (no q)
10. no (p o q) ⇔ (no p) y (no q)

Vamos a hablar más particularmente de las leyes 9 y 10, conocidas con el nombre de leyes de De Morgan, que a menudo son utilizadas en programación cuando necesitamos la negación de un predicado compuesto.

Volvamos a las dos frases que ilustran la utilización de y y de o. La negación de estas frases puede ser expresada de la manera siguiente:

> "Si Luis no ha acabado su trabajo o si no tiene suficiente dinero, no irá al cine".
>
> "El barco no partirá mañana si no hay al menos veinte pasajeros o si no hace buen tiempo".

En la primera frase, se ha tomado la negación de proposición (p y q): se ha compuesto la negación de p con la negación de q por medio del conector o.

– no (p y q) ⇔ (no p) o (no q)

En la segunda frase, la negación de la proposición p o q nos ha llevado a la expresión (no p) y (no q)

– no (p o q) ⇔ (no p) y (no q)

Estas leyes de De Morgan pueden demostrarse fácilmente con ayuda de una tabla de verdad, considerando todos los valores que pueden tomar p y q y aplicando las propiedades de los conectores y, o, no.

La tabla siguiente presenta la tabla de verdad de la primera ley de De Morgan (9):

p	q	p y q	no (p y q)	no p	no q	(no p) o (no q)
falso	falso	falso	cierto	cierto	cierto	cierto
falso	cierto	falso	cierto	cierto	falso	cierto
cierto	falso	falso	cierto	falso	cierto	cierto
cierto	cierto	cierto	falso	falso	falso	falso

La cuarta columna es idéntica a la última. Tenemos:

no (p y q) ⇔ (no p) o (no q)

para todos los valores tomados por p y q, en L.

Se pueden hacer numerosas extensiones a partir de estas nociones elementales de lógica.

4- ITERACIONES.

Llamamos iteración al hecho de repetir la ejecución de una secuencia de acciones o de una acción. Para describir una iteración, si conocemos el número n de repeticiones, podemos escribir simplemente n veces la acción o la secuencia de acciones a repetir. Sin embargo, si n es grande, esta operación puede resultar fastidiosa y la secuencia algorítmica que resultará será pesada de leer.

Por otra parte, es frecuente que el número de repeticiones no pueda determinarse fácilmente. En la agrupación de fotocopias descrita en el tema 1, el número de ejemplares agrupados (es decir, el número de repeticiones de la operación de agrupar) sólo puede conocerse al final de la agrupación, cuando se hayan agotado las copias para una de las hojas del fotocopiado. Es, por tanto, necesario en programación disponer de estructuras algorítmicas que permitan describir una iteración de forma cómoda. Vamos, en principio, a estudiar algunos de los esquemas clásicos de iteración.

4.1- Estudio de algunos esquemas iterativos.

4.1.1- El esquema *repetir*.

Es un esquema que ya hemos visto y utilizado en los temas anteriores. Así, la acción C de la agrupación de fotocopias ha sido descompuesta como:

repetir
    C1: coger las hojas necesarias para un ejemplar
    C2: presillar el ejemplar
    C3: ponerlo sobre el escritorio
hasta que (C4) una de las mesas quede sin hojas

En un esquema tal, se distingue:

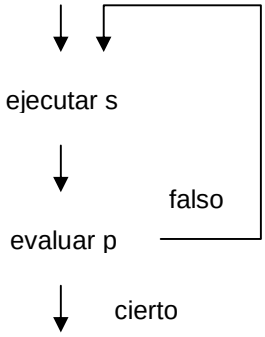
- la secuencia s de las acciones repetidas (aquí C1, C2, C3),
- la condición de parada de la repetición expresada bajo la forma de un predicado p (aquí C4).

El predicado se evalúa después de cada repetición e la secuencia; ésta es, por tanto, ejecutada al menos una vez, y se puede describir la forma general del esquema *repetir* como lo el siguiente:

repetir s *hasta que* p

s: acción o secuencia de acciones

p: predicado de control de la iteración



Observemos que, en esta estructura repetitiva, el predicado de control es un predicado de salida: expresa la condición que hace "salir" ala procesador de la iteración. Observemos también que, con un esquema así, el número de repeticiones no es, en general, conocido con antelación. En el caso de nuestro ejemplo, para conocerlo antes de ejecutar la agrupación sería necesario:

- contar, para cada paquete, el número de ejemplares,
- determinar, a continuación, el más pequeño de los 10 números obtenidos en la cuenta previa.

El algoritmo que hemos compuesto en el tema 1 no efectúa todos estos cálculos, que son inútiles para la realización de las agrupaciones: el número de repeticiones sólo podrá ser conocido a a: será el número de ejemplares agrupados.

4.1.2- El esquema *mientras*.

El esquema *repetir* no permite representar fácilmente todos los casos de iteración. Demostrémoslo estudiando un nuevo ejemplo.

Supongamos, contrariamente a lo que habitualmente admitimos en este libro, que sólo se dispone de un procesador rudimentario, para le cual las únicas operaciones primitivas son la división y la resta. Se quiere realizar, para este procesador, un algoritmo que permita la división de dos enteros naturales. A partir de un dividendo y un divisor, este algoritmo debe determinar un cociente y un resto. El entorno de este algoritmo se describe como sigue:

Entorno del algoritmo DIVISIÓN

Variable	Función	Estado inicial	Estado final
DVDO	parámetro-dato, variable numérica cuyo valor debe ser entero y representar el dividendo.	a, entero ≥ 0	a
DVSR	parámetro-dato, variable numérica	b, entero > 0	b

	cuyo valor debe ser entero y representar el divisor.		
C	parámetro-resultado, variable numérica en la que se calcula el cociente.	indeterminado	a div b
R	parámetro-resultado, variable numérica en la que se calcula el resto.	indeterminado	resto de la división de a por b

Para encontrar un principio de resolución del problema, razonemos a partir de un ejemplo. Supongamos que estamos en el estado inicial {DVDO : 32 DVSR : 9}. El cociente a determinar es 3 y el resto es 5. La división euclidiana ($a = bq + r$, $0 \leq r < b$) se expresa, en este ejemplo, por $32 = 3 * 9 + 5$. La igualdad anterior puede escribirse también: $32 - 3 * 9 = 5$, expresando así el hecho de que se puede sustraer como máximo 3 veces 9 de 39, obteniendo 5 como resultado de la última resta.

Para efectuar la división, vamos a restar el valor del divisor del valor del dividendo tantas veces como sea posible, y contar en C el número de restas. El valor-resultado de la última resta será el resto que se busca. Para evitar modificar DVDO y ya que el último resultado de las restas debe ser el valor final de R, recopiaremos, en principio, el valor de DVDO en R e iremos restando a R. Utilizando un esquema *repetir*, vamos a componer la secuencia:

```
R ← DVDO
C ← 0
repetir
    R ← R – DVSR
    C ← C + 1
hasta que R < DVSR
```

Desgraciadamente, esta secuencia es incorrecta. Produce un resultado falso cuando el dividendo es inferior al divisor. Con un dividendo de 15 y un divisor de 20 calculará un cociente de 1 y un resto de –5 (en lugar de 0 y 20). Para que sea correcta, convendría que la secuencia repetida en la iteración no sea ejecutada ciando la condición de parada (R < DVSR) se cumple en el momento en que se aborda la iteración. Como el esquema *repetir* implica al menos una ejecución de la secuencia repetida, vamos a proponer una segunda estructura iterativa y expresar el algoritmo como sigue:

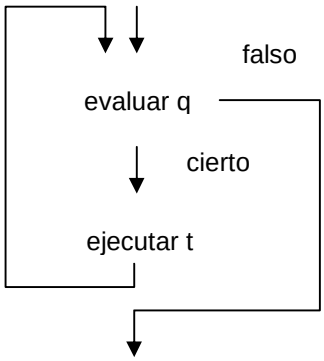
```
algoritmo DIVISIÓN (DVDO, DVSR, C R)

inicio
R ← DVDO
C ← 0
mientras R ≥ DVSR hacer
```

R ← R – DVSR
C ← + 1
finmientras
fin

En este algoritmo, el esquema *mientras* expresa la repetición de una secuencia de dos acciones, repetición controlada por el predicado R ≥ DVSR, que expresa la condición de mantenimiento de la iteración. Este predicado es evaluado antes de cada ejecución de la secuencia: si es falso, la repetición casa. Por consiguiente, si es falso en el momento en que la iteración es abordada, la secuencia a repetir no se ejecuta.

Las características del esquema *mientras* son resumidas como sigue:

mientras q *hacer* t *finmientras*



En este esquema, la palabra *finmientras* sirve como delimitador para indicar el fin de la secuencia a repetir. Con el esquema *repetir*, la secuencia es delimitada por las palabras *repetir* y *hasta que*.

Como en el esquema anterior, el número de repeticiones obtenidas con un esquema *mientras* no es en general conocido con antelación. En el caso del ejemplo analizado, este número sólo puede conocerse después de la ejecución de la iteración: es el valor final de C.

4.1.3- El esquema *para*.

El algoritmo SUMAENTEROS, presentado en el tema 2, encierra una iteración cuyo número de repeticiones es conocido con antelación: es el valor del parámetro NUM. También se puede describir en una frase el tratamiento efectuado por este algoritmo:

> "Para cada uno de los valores de ENTERO de 1 a NUM, acumular ENTERO en S"

En programación, numerosas iteraciones tienen la misma forma que la anterior y es cómodo disponer de un esquema para describirlas. Es el esquema *para* que podemos utilizar en una reformulación del algoritmo SUMAENTEROS:

```
algoritmo SUMAENTEROS2 (S, NUM)

inicio
S ← 0
para ENTERO de 1 a NUM hacer
     S ← S + ENTERO
finpara
fin
```

En el algoritmo sumaenteros2, la variable ENTERO es la variable de control de un esquema *para*. Su variación de 1 en 1 es definida en el esquema por el valor inicial (1) y el valor-límite (valor de NUM), para los cuales la acción S ← S + ENTERO será ejecutada. Con un esquema *para*, el procesador se encarga, él mismo, de la variación de la variable de control y de la parada de la iteración. El modo de variación, mientras no esté precisado, supone implícitamente un incremento en 1 entre los valores sucesivos. Es posible, como lo indica la forma general presentada a continuación, especificar un paso de variación diferente a 1. En esta figura presentamos un esquema *para* con crecimiento de la variable de control. La equivalencia dada en este esquema muestra las operaciones de control que serán realizadas por el procesador en la ejecución del esquema *para*. Se observará que, si el valor inicial vi es superior al valor límite vf, la secuencia a no es ejecutada. Recordemos que la ausencia de una indicación de variación de la variable de control implica una variación por pasos sucesivos de 1.

```
para V de vi a vf incremento inc          ⇔   V ← vi
hacer a finpara                               mientras V ≤ vf hacer
                                                        a
                                                        V← V + inc
                                              finmientras
```

V: variable de control.
vi: primer valor de la variable.
vf: valor límite de la variable.
inc: valor positivo de incremento.
a: acción o secuencia de acciones a repetir.

Igualmente, se pueden utilizar los esquemas *para*, en los cuales la variable de control decrece. Su forma general se describe a continuación:

```
para V de vi a vf decremento dec          ⇔   V ← vi
hacer a finpara                               mientras V ≥ vf hacer
                                                        a
                                                        V← V - dec
                                              finmientras
```

V: variable de control.
vi: primer valor de la variable.
vf: valor límite de la variable.
inc: valor positivo de disminución.
a: acción o secuencia de acciones a repetir.

Veamos aquí, ya que la variable de control decrece, que la condición que mantiene en la iteración se expresa, a la inversa de la variación creciente, por el predicado V ≥ vf. En este caso, un valor inicial al valor límite impide toda repetición.

4.1.4- El esquema general.

En Informática, se han de tratar frecuentemente sucesiones finitas de informaciones, cuyo número de elementos no se puede conocer con antelación: sucesión de modificaciones registradas en las cuentas bancarias, sucesión de las variaciones del stock de los productos, etc... Estas sucesiones de informaciones están colocadas en los aparatos de lectura de manera que cada información a tratar pueda entrar en el entrono del algoritmo de tratamiento por medio de una acción de lectura. Una de las características de los aparatos de lectura es que, para señalar el fin de una sucesión, responden por una señal de rechazo a todo intento de lectura de una información posterior a la última. Por consiguiente, sólo sabremos que se ha tratado la última información de una sucesión después de haber intentado leer una información suplementaria.

La descripción del tratamiento de una sucesión de informaciones tal comprende necesariamente una iteración en la que cada repetición trata una información. Como el número de elementos de la sucesión no se conoce con antelación, no es posible utilizar el esquema *para*. Con un esquema *repetir*, una secuencia tal que no permite tratar las sucesiones vacías, ya que impone que toda sucesión tratada contenga, al menos, una información.

```
leer una información    {la primera sucesión}
repetir
    tratar la información leída
    intentar leer una información
hasta que rechazo de lectura
```

Con un esquema *mientras* se puede expresar correctamente una secuencia de tratamiento:

```
intentar leer una información
mientras no rechazo de lectura hacer
        tratar la información leída
        intentar leer una información
finmientras
```

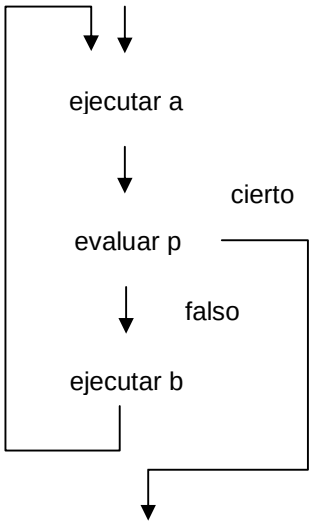
Esta secuencia presenta, sin embargo, el inconveniente de que aparece dos veces el enunciado “intentar leer una información”. De hecho, los problemas que se nos presentan para describir simplemente la iteración provienen del hecho de que el control de esta iteración no debe ser efectuado ni al principio ni al final de la secuencia repetida, sino entre la lectura y el tratamiento. Necesitamos proponer la secuencia:

```
iterar
    intentar leer una información
    salirsi rechazo de lectura
    tratar la información leída
finiterar
```

en la que utilizamos un nuevo esquema que llamaremos el esquema general y cuyo principio está explícito el el siguiente esquema:

```
iterar
    a
    salirsi p
    b
finiterar
```

a, b: acciones o sucesiones de acciones.
p: predicado de control.



Los esquemas *mientras* y *repetir* son casos particulares del esquema general.

Cuando a es una acción vacía, el esquema anterior es equivalente a:

mientras *no* p *hacer* b *finmientras*

Cuando b es una acción vacía, es equivalente a:

repetir a *hasta que* p

Si a y b no son vacías, a es ejecutada una vez más que b.

En el resto del libro sólo utilizaremos dos esquemas iterativos:

- el esquema para, que estará reservado a las iteraciones para las cuales el número de repeticiones es conocido con antelación;
- el esquema general, que será empleado en el resto de los casos.

La elección que acabamos de hacer sólo tiene valor en el contexto de una introducción a la programación.

4.2- Noción de vector.

Vamos ahora a definir una nueva escritura de objeto, frecuentemente utilizada en Informática, y a la que se le aplican tratamientos iterativos,

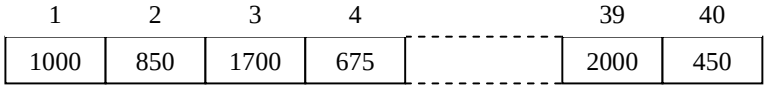
Consideremos el siguiente problema: una empresa recoge cada mes el importe de las ventas realizadas por cada una de sus 40 sucursales. Se quiere componer una secuencia algorítmica que calcule el total de ventas.

Una primera solución consiste en definir 40 variables VENTA1, VENTA2, ... Si TOAL es la variable de acumulación se escribirá la acción:

TOTAL ← VENTA1 + VENTA2 + ... + VENTA40

En la realidad, habrá que indicar explícitamente las 40 variables, ya que ningún procesador comprenderá el significado de los puntos suspensivos que hemos utilizado. En la práctica, raramente se adopta una solución así: es fastidioso y costoso enumerar y tratar demasiadas variables.

Otro método consiste en agrupar los valores en un objeto de estructura análoga a la de una variable compuesta, pero donde todas las subdivisiones son del mismo tipo y cada una contiene uno de los 40 valores:



Demos un nombre a este objeto: llamémosle VENTA. VENTA puede representarse esquemáticamente en forma de una tabla. Para designar uno de los 40 valores se utilizará su posición en la tabla: VENTA(4) representa el cuarto valor (675 en nuestro esquema).

Acabamos de definir un vector (VENTA) en el que los 40 elementos son VENTA1, VENTA2, ... , VENTA40. Para designar un elementos se utilizará el nombre del vector seguido de una expresión aritmética entre paréntesis. El valor de la expresión de la posición del elemento:

VENTA (10), VENTA (I), VENTA (k + 3)

En la expresión VENTA (I), I debe ser una variable numérica de valor entero i, tal que $1 \leq i \leq 40$: el elemento designado es el i-ésimo del vector. Si k es el valor de la variable K, V(k + 3) designa el (k +3)ésimo elemento: K debe ser entero y estar comprendido entre -2 y 37.

Volviendo al problema planteado, podemos ya describir una solución diciendo que es suficiente con acumular cada elemento del vector VENTA en la variable TOTAL: si VENTA(I) representa un elemento, hay que acumular VENTA(I), para los valores 1, 2, ..., 40 de I. De donde:

```
{I variable numérica}

TOTAL ← 0
para I de 1 a 40 hacer
    TOTAL ← TOTAL + VENTA(I)
finpara
```

La noción de vector, que hemos introducido a partir de un ejemplo, puede formalizarse de la siguiente manera:

Dado un conjunto E de n valores del mismo tipo (n ≥ 1), un **vector** V es una aplicación del intervalo {1 , n} en E. El número n es la **dimensión** del vector. La **indexación** es la operación elemental indicada por V(i) que da el iésimo valor (1 ≤ i ≤ n). En Informática, se designa por abuso de lenguaje bajo el nombre de vector al conjunto {V(1), V(2), ... , V(n)} y V(1), V(2), ... , V(n) son los **elementos** del vector.

Un elemento de vector puede contener un valor compuesto; entonces, se le tratará exactamente como una variable compuesta de igual estructura. Consideremos, por ejemplo, un vector RESUL, en el que se han registrado los importes de las 40 ventas el problema anterior, pero indicando el lugar de cada sucursal:

RESUL (5)		RESUL (6)	
Lugar	Importe	Lugar	Importe
'Habana'	10500	'Tunas'	8650

El vector RESUL contiene 40 elementos; cada elemento es relativo a una sucursal y comprende dos subdivisiones:

- la subdivisión LUGAR, de tipo carácter, cuyo valor indica la ciudad donde se encuentra la sucursal,
- la subdivisión IMPORTE, de tipo numérico, cuyo valor indica el importe de las ventas de la sucursal.

Para designar una subdivisión de un elemento particular, se utilizará una notación análoga a ha adoptada por las variables compuestas:

RESUL (5). LUGAR,

designa la subdivisión LUGAR del elemento de la posición 5. si en el entorno en el que interviene RESUL no existe ningún otro objeto que contenga una subdivisión de nombre LUGR, se puede simplificar la expresión designando la quinta ciudad por LUGAR (5), ya que no hay ninguna ambigüedad en cuanto al valor indicado.

En Informática, sucede frecuentemente que se deba efectuar un tratamiento dado a cada elemento de un vector. La descripción algorítmica de un trabajo de este tipo comportará una iteración en la que la secuencia repetida se deberá efectuar para cada elemento del vector. Esta iteración se describirá en general muy simplemente por medio de un esquema *para*. Por ejemplo, supongamos que se desea, a partir de un vector RESUL, imprimir la lista de las ciudades cuyas sucursales han realizado ventas de más de $10 000 (se admitirá que la ejecución de una acción

escribir corresponde a la impresión de una línea). Es necesario, por tanto, examinar *cada* elemento del vector: *si* el importe indicado para este elemento es superior a 10 000, se imprime e lnombre de la ciudad correspondiente. La solución se deduce de la frase anterior: las palabras subrayadas indican que la secuencia debe contener un esquema *para* (cada), cuya secuencia repetida será un esquema condicional (si). Si I es una variable numérica, se puede entonces escribir:

```
inicio
Escribir 'LISTA DE CIUDADES CON VENTAS E NSUCURSALES > 10
000' {título}
para I de 1 a 40 hacer
    si RESUL (I).IMPORTE > 10 000
        entonces
        escribir RESUL(I).LUGAR
    finsi
finpara
fin
```

4.3- Búsqueda de una información en un vector.

Abordamos aquí un problema frecuentemente planteado en Informática: ¿Cómo determinar si una información dada está o no presente en un vector?. Este problema no es tan elemental como parece, y para los principiantes presenta un cierto número de dificultades que intentaremos poner de relieve cuando estudiemos la coherencia de una iteración más adelante. Conformémonos con dar aquí el estudio del problema a partir de un ejemplo.

Consideremos un vector INSCRITO de 50 elementos, donde cada elemento es de tipo carácter y contiene el nombre de una persona matriculada en un curso. Se supone que, entre los matriculados, no hay homónimos: todos los nombres que figuran en el vector son distintos. Se dispone también de una variable NOMBRE, de tipo carácter, que contiene el nombre de una persona. Se desea saber si esta persona está matriculada y, en caso afirmativo, cuál es la posición del elemento de INSCRITO relativo a esta persona.

Para el algoritmo que vamos a elaborar. El entorno comprende dos parámetros-dato: el vector INSCRITO y la variable NOMBRE. Tenemos que completar este entorno con los objetos necesarios para la resolución del problema. De hecho, la cuestión planteada es doble:

1. ¿Está inscrita la persona?.
2. Si está inscrita, ¿en qué elemento aparece?.

Completamos entonces el entorno con la definición de las dos variables ENCONTRADO y POSICIÓN:

Variable	Función	Estado inicial	Estado final

ENCONTRADO	Parámetro-resultado, variable lógica que indica si la persona está o no matriculada.	indeterminado	cierto si el valor de NOMBRE está presente en el vector, si no, falso.
POSICIÓN	Parámetro-resultado, variable lógica que indica la posición de la persona en INSCRITO, si está matriculada.	indeterminado	– indeterminado si ENCONTRADO: falso. – si ENCONTRADO cierto: POSICIÓN: r, tal que 1≤r≤50 y INSCRITO(r)=NOMBRE.

Definidas estas variables, ¿cómo resolver el problemas?. Se puede examinar el vector, elemento por elemento, partiendo del primero y parándose cuando se encuentre el nombre que se busca o, en su defecto, cuando se alcanza el último elemento del vector. La iteración que aparece tratará, en cada repetición, un elemento del vector, pero el número de repeticiones, que será igual al número de elementos examinados, no se puede conocer con antelación: por consiguiente, aquí no podemos utilizar un esquema *para*. Añadiendo al entorno una variable numérica I que sirva para designar los elementos del vector, podemos describir la iteración por:

```
I ← 1 {considerar el primer elemento de INSCRITO}
iterar
    salirsi NOMBRE = INSCRITO (I) o I = 50 {el nombre que se busca no está
                                            en el elemento considerado y
                                            este elemento no es el último}
    I ← I + 1 {considerar el elemento siguiente}
finiterar
```

Ala salida de la iteración, queda por determinar si la búsqueda se ha parado por haber encontrado el nombre buscado o si ha sido infructuosa. Esta operación se expresa simplemente con ayuda de un esquema condicional y se puede dar el algoritmo completo:

```
algoritmo BUSCAR (INSCRITO, NOMBRE, ENCONTRADO, POSICIÓN)

inicio
    I ← 1
    iterar
        salirsi NOMBRE = INSCRITO (I) o I = 50
        I ← I + 1
    finiterar
    si NOMBRE = INSCRITO (i)
        entonces
            ENCONTRADO ← cierto
            POSICIÓN ← I
        sino
            ENCONTRADO ← falso
    finsi
```

fin

4.4- Cómo definir una iteración.

4.4.1- Utilización del esquema *para*.

Ya hemos estudiado este esquema. Se puede resumir estas formas de la siguiente manera:

para V *de* vi *a* vf { incremento inc / decremento dec } *hacer* a *finpara*

En esta representación, a es una acción o una secuencia de acciones que llamaremos el núcleo de la iteración. Con este esquema, es el procesador quien se encarga de la variación de la variable de control de la iteración, Y lo hace según las indicaciones dadas en el esquema. Las acciones implícitas de control que se ejecutan por el procesador están encuadradas.

para con *incremento*	*para* con *decremento*
V ← vi	V ← vi
Iterar	*Iterar*
salirsi V > vf	*salirsi* V < vf
a	a
V ← V + inc	V ← V - inc
finiterar	*finiterar*

De una manera general, vi, vf, inc y dec son constantes o variables numéricas. Sin embargo, numerosos procesadores impiden que inc y dec estén representados por variables, para evitar errores tales como el que consistiría en designar como incremento una variable cuyo valor fuera negativo. Es fácil constatar entonces que el número de repeticiones es determinado en el momento en que el procesador aborda la ejecución de la iteración. Por ejemplo, en un esquema *para* con incremento, sea n la parte entera de (vf / vi) / inc. El número de repeticiones es igual a n + 1 si vi ≤ vf. Es nulo si vi > vf. Por tanto, es simple asegurar el hecho de que el número de repeticiones será finito. Hemos indicado ya que el esquema *para* está concebido para describir iteraciones cuyo número de repeticiones se conoce con antelación. Sin embargo, si el núcleo de un esquema *para* contiene acciones

que modifican a V o vf o inc o dec, el número de repeticiones ya no podrá ser determinado a priori. Si se describe una iteración con un esquema *para* y si se observa que es necesario escribir en el núcleo acciones que modifican a V, vf, inc o dec, esto significa que no se ha elegido la estructura correcta y que hay que utilizar un esquema general.

4.4.2- Utilización de un esquema general.

Toda iteración debe ser finita: debe terminarse después de un número finito de repeticiones. Si una iteración está descrita por un esquema *para*, utilizado de acuerdo con las indicaciones del párrafo anterior, es fácil demostrar que satisface la propiedad anterior. No funciona igual con un esquema general, para le cual el programador deberá:

- verificar la coherencia del predicado de control,
- asegurarse de que una de las evaluaciones de este predicado pondrá fin a las repeticiones.

4.4.2.1- Coherencia del predicado de control.

Diremos que un predicado es **coherente** si su evaluación es siempre posible a lo largo de la ejecución de la iteración.

Para explicitar esta definición, volvamos al ejemplo del algoritmo BUSCAR y consideremos otra solución que puede ser construida a partir de un principio ligeramente diferente: se examina el vector elemento por elemento ... si se rebasa el 50avo elemento, significa que el nombre que se busca no se encuentra en INSCRITO:

```
inicio
    I ← 1
    iterar
A       salirsi NOMBRE = INSCRITO (I) o I = 50
        I ← I + 1
    finiterar
    si I > 50
        entonces
            ENCONTRADO ← falso
        sino
            ENCONTRADO ← cierto
            POSICIÓN ← I
    finsi
fin
```

En la iteración A, el predicado de control es incoherente: si el nombre que se busca no se encuentra en INSCRITO para I : 50:

NOMBRE ≠INSCRITO (I) *y* I > 50

el predicado de control será falso y la iteración continuará con I : 51. En esta última repetición el procesador no podrá evaluar el predicado elemental NOMBRE = INSCRITO (I), ya que el vector INSCRITO no tiene el 51avo elemento.

Se pudiera pensar que, para eliminar la incoherencia, es suficiente con volver a escribir el predicado de control:

NOMBRE = INSCRITO (I) *o* I = 50

y obligar así al procesador a comenzare por la evaluación de I > 50. Dando para I : 51 el valor cierto, el predicado elemental NOMBRE = INSCRITO (I) no tendrá necesidad de ser evaluado. Observemos que el conector o es conmutativo: cualquiera que sea la escritura del predicado de control, no se puede prejuzgar el sentido en que será tratado por el procesador: éste podrá comenzar la evaluación tanto por la derecha como por la izquierda, o también evaluar sistemáticamente los dos predicados elementales.

Es, general, más fácil probar la incoherencia de un predicado de control (es suficiente encontrar un caso de evaluación incorrecta) que demostrar su coherencia. En la mayoría de los casos, será suficiente con asegurar que el predicado será evaluado correctamente en los casos límite (aquí, primer y último elemento del vector). El examen profundizado de la coherencia se hará con el estudio de la terminación de la iteración.

4.4.2.2- Parada de la iteración.

Para demostrar que una iteración descrita por un esquema general es finita, basta con probar que el predicado de control tomará el valor cierto al final de un número determinado de repeticiones. Por ejemplo, con la iteración del algoritmo BUSCAR:

```
I ← 1
iterar
    salirsi NOMBRE = INSCRITO (I) o I = 50
    I ← I + 1
finiterar
```

en principio, se puede constatar que la variable I variará de 1 en 1 de forma creciente a partir de 1. Si, por consiguiente, el nombre buscado se encuentra en uno de los elementos de INSCRITO de posición r ($1 \leq r \leq 50$), la iteración se acabará al final de r repeticiones, cuando el predicado NOMBRE = INSCRITO (I) tome el valor cierto. Si el nombre buscado no se encuentra en INSCRITO, la iteración cesará después de 50 repeticiones, cuando el predicado I = 50 tome el valor cierto.

Cuando no sea fácil probar la parada de la iteración, se puede controlar la construcción de la iteración utilizando afirmaciones, conforme a lo que se puede llamar la lógica del esquema general. Esta lógica está descrita en el siguiente esquema:

```
I ← 1
iterar
    salirsi p
    {no p} b {C}
finiterar
{p}
```

Expresa, a primera vista, que cada vez que la iteración continúa después del examen del predicado de control, no p es un antecedente de b. Cuando la iteración cesa, se pasa a la acción siguiente al esquema general, p es evidentemente un antecedente de esta acción. Finalmente, si C es un consecuente de b, C es igualmente un antecedente de a a partir de la segunda repetición.

4.5- Eficacia de una iteración.

La eficacia de un algoritmo se define, en general, en función del contexto en el que dicho algoritmo es utilizado. En ciertos casos, un algoritmo será más eficaz que otro si efectúa el mismo trabajo más rápidamente. En una situación en la que la capacidad del entorno es limitada, el algoritmo que utilice el mínimo de objetos podrá ser considerado como el más eficaz. No vamos a abordar los problemas asociados al estudio de la eficacia de un algoritmo. Sin embargo, a propósito de lo esquemas iterativos y a partir de algunos ejemplos, vamos a poner en relieve la posibilidad de influir, en algunos casos, sobre la duración de una iteración. De hecho, la rapidez de ejecución de una iteración puede tener una gran influencia en el desarrollo de un algoritmo.

Toda iteración implica una repetición. Si elegimos entre dos acciones a repetir, una más rápida que la otra, la repetición multiplicará la diferencia. Para convencernos, vamos a considerar el siguiente ejemplo, en el cual supondremos que la acción a es ejecutada diariamente para cada una de las 20 000 cuentas de los clientes de una empresa.

```
a                b                c
iterar           iterar           iterar
    b                c                d
    salirsi p        salirsi q        salirsi r
finiterar        finiterar        finiterar
```

La acción a es una iteración en la que se repite la acción b. Consideremos la hipótesis de que, si se observa un gran número de ejecuciones de a, se constata

que b se repite un promedio de 100 veces para una ejecución de a. Supondremos también que, para cada ejecución de b, c es repetida un promedio de 50 veces y d 20, para una ejecución de c. Cada día se ejecuta la acción d:

20 000 * 100 * 50 * 20 = 2 000 000 000 veces

Supongamos, finalmente, que una ejecución de d dura 10 microsegundo (i microsegundo = 1 millonésima de segundo). Si vemos que es posible reemplazar d por d', que sólo dura 8 microsegundos, la ganancia de tiempo diario será de:

$(2 . 10^{9}) * (2 . 10^{-6}) = 4\ 000$ segundo

más de un ahora de trabajo del procesador.

```
iterar
    X ← X + Y
    salirsi X > K + T
    Y ← Y * Z
finiterar
```

Para mejorar la actuación de una iteración, se puede intentar reemplazar las acciones repetidas por acciones equivalentes, pero más rápidas. No hay que olvidar que la duración de una repetición comprende igualmente el tiempo necesario de la evaluación del predicado de control: si se puede reducir este tiempo, la iteración es más rápida. Consideremos, por ejemplo, esta secuencia en la que X, Y, Z, K, y T son variables numéricas. Las variables K y T no varáin en la iteración. Sin embargo, para cada repetición, le procesador debe rehacer la operación K + T, cuyo resultado no varía, para poder evaluar el predicado de control. Utilizando otra variable numérica S para calcular antes de la iteración la suma K + T, se puede reemplazar la secuencia de l esquema por:

```
S ← K + T
iterar
    X ← X + Y
    salirsi X > S
    Y ← Y * Z
finiterar
```

Si n es el número medio de repeticiones de la iteración y si n es superior a 1, la nueva secuencia es más rápida, ya que efectúa (n – 1) sumas menos que la del esquema:

```
iterar
    X ← X + Y
    salirsi X > K + T
    Y ← Y * Z
```

finiterar

una suma más fuera de la iteración, n menos en la iteración).

Para hacer una iteración más rápida, se puede también intentar reducir el número de comparaciones efectuadas a título de evaluación del predicado de control. Por tanto, para las iteraciones controladas por un predicado compuesto de la forma *salirsi* v *o* w (en la que v y w son predicados elementales), a veces es posible eliminar una de las dos evaluaciones de v y w. Volvamos al algoritmo de búsqueda secuencial de una información en un vector, que recordamos seguidamente.

```
algoritmo BUSCAR (INSCRITO, NOMBRE, ENCONTRADO, POSICIÓN)

inicio
    I ← 1
    iterar
        salirsi NOMBRE = INSCRITO (I) o I = 50
        I ← I + 1
    finiterar
    si NOMBRE = INSCRITO (i)
        entonces
            ENCONTRADO ← cierto
            POSICIÓN ← I
        sino
            ENCONTRADO ← falso
    finsi
fin
```

En la iteración de este algoritmo, el predicado elemental I = 50, que forma parte del predicado de control, sólo sirve si el nombre buscado no está en el vector: si estuviéramos seguros de encontrarlo, se podría omitir este control. De aquí la idea de “prolongar” el vector INSCRITO con un 51avo elemento, en el que se colocará, antes de la iteración, el nombre buscado de “centinela” (esto sólo es posible si el entorno del problema permite esta extensión del vector). En la iteración de búsqueda, el predicado I = 50 ya no es necesario: incluso si el nombre buscado no es uno de los 50 primeros valores, el valor colocado de centinela hará cierto al predicado NOMBRE = INSCRITO (I) para I = 51. Se obtiene entonces el algoritmo:

```
algoritmo BUSCARCENTINELA (INSCRITO, NOMBRE, ENCONTRADO,
POSICIÓN)

inicio    {INSCRITO es un vector de 51 elementos donde sólo los 50
          primeros contienen inicialmente unja información significativa; el
          51avo elemento es utilizado como centinela}
    I ← 1
    INSCRITO (51) ← NOMBRE        {el nombre se coloca de centinela}
```

```
    iterar
        salirsi NOMBRE = INSCRITO (I)
        I ← I + 1
    finiterar
    {si se para sobre el centinela, el nombre buscado no pertenece al
    vector}
    si I = 51
        entonces
            ENCONTRADO ← falso
        sino
            {I ≤ 50}
            ENCONTRADO ← cierto
            POSICIÓN ← I
    finsi
fin
```

En el algoritmo BUSCAR, se efectúan, en cada repetición de la iteración, dos comparaciones y un incremento. Con el algoritmo BUSCARCENTINELA, cada repetición no necesita más que una comparación y un incremento, y, si se admite que una comparación y una suma duran el mismo tiempo, el tiempo de ejecución de la iteración disminuye en un tercio.

La transformación del ejemplo anterior necesita que se pueda prolongar el vector utilizado. Esto no siempre es posible: la longitud de un vector puede estar limitada de forma estricta por las condiciones en las que trabaja el procesador. Sin embargo, puede suceder que sea posible efectuar el mismo tipo de operación en el predicado de control, sin tener que modificar los objetos utilizados.