CAPÍTULO 1

CONSTRUCCIÓN DE ALGORITMOS EN PSEUDOCÓDIGO

Resolución de problemas

⇨ El objetivo principal de la programación es la resolución de problemas de tipo computacional (resolubles por una computadora). Sin embargo, en líneas generales, los mismos métodos aplicables a la resolución de cualquier problema pueden ser aplicados a los problemas computacionales.

Por otro lado, no es fácil determinar cuándo un problema es computacional y cuándo no. Por ejemplo, supongamos que enfrentamos el siguiente problema: se ha pinchado un neumático de un auto y se lo debe cambiar. Prácticamente, toda persona a la que se le preguntara de qué tipo de problema se trata, diría que no es computacional. ¿Pero qué ocurriría si decimos que se cuenta con un robot para hacer el trabajo? ¿Acaso el robot no necesitaría una computadora que lo controle? ¿No sería necesario programarlo?

Como se ve, las diferencias entre los problemas computacionales y los que no lo son, son más sutiles de lo que parecen. Es por eso que encararemos la resolución de problemas en general, para recién luego encarar la resolución de problemas computacionales básicos.

Tratemos de plantear una solución para el problema del neumático pinchado.

Ante todo, ¿se está con el auto andando o está detenido? ¿se cuenta con los elementos necesarios para cambiarla? Supondremos el caso ideal: el auto está detenido, en un lugar seguro y contamos con todos los elementos para resolver el problema.

Planteemos un método para resolver el problema. Para algunas personas (las llamaremos ***interlocutores***) no será necesaria la descripción de un método y bastará con decirle:

1. Cambiar el neumático pinchado

Sin embargo, esta descripción del método puede no ser suficiente para quien nunca vio cambiar uno. Tal vez sea necesario descomponer el paso 1 en:

1.1. Levantar el auto
1.2. Quitar la rueda pinchada
1.3. Traer la rueda de auxilio
1.4. Colocar la rueda de auxilio

1.5. Bajar el auto
1.6. Guardar la rueda pinchada

Para la mayoría de los interlocutores esta descripción bastará. Pero ¿qué ocurriría si nuestro interlocutor es totalmente inexperto en estas tareas? Tal vez el paso 1.1 se deberá descomponer en:

1.1.1. Traer el gato
1.1.2. Colocar el gato en la ranura
1.1.3. Subir el gato hasta que la rueda se separe del suelo

Y el paso 1.4 se podría descomponer en:

1.4.1. Poner la rueda en posición
1.4.2. Ajustar las 4 tuercas

Así, podríamos seguir refinando el método de solución, por ejemplo, para un humano de una cultura que no utiliza automóviles, o para un robot.

Ahora que ya se ha descripto el método se debe ***ejecutarlo*** para solucionar el problema.

⇨ Hemos definido tres pasos para la resolución del problema: lo hemos enunciado sin ambigüedades, hemos encontrado un método que sea comprensible por el interlocutor y el interlocutor ha ejecutado el método.

A la programación le interesa la segunda de estas tres etapas. Parte de un problema ya enunciado sin ambigüedades y halla un método de resolución acorde con el interlocutor, para que luego éste lo ejecute.

Algunas definiciones

Llamamos ***interlocutor*** a toda entidad capaz de comprender un método enunciado (o "codificado") y ejecutarlo. En el caso anterior, el interlocutor sería la persona que va cambiar la rueda.

El conjunto de los recursos necesarios para ejecutar un método se denomina ***ambiente***. En el caso anterior supusimos que el ambiente tenía todo lo necesario para la ejecución de la tarea: rueda de auxilio, gato, hombre que realizará la tarea, etc. . Todo lo que no es el ambiente del problema se denomina ***medio*** o ***medio externo***.

Una ***acción*** es un evento que modifica el ambiente. Los pasos numerados del ejemplo anterior constituyen acciones, ya que cada una de ellas modifica de algún modo el ambiente. Por ejemplo, la acción 1.5 toma al ambiente con el auto sobre el gato y lo modifica, quedando éste sobre el piso.

Sin embargo, no todas las acciones son ejecutables por cualquier interlocutor. En el ejemplo

anterior vimos que, según el tipo de interlocutor, deberíamos refinar más o menos las acciones. Una acción es ***primitiva*** para un interlocutor dado si éste puede ejecutarla sin información adicional. Por ejemplo, para casi cualquier habitante actual de la Argentina, la acción 1 va a ser primitiva; no ocurriría lo mismo para antiguo azteca.

Un ***algoritmo*** es una secuencia ordenada de acciones primitivas que pueden ser ejecutadas por un interlocutor y que llevan a la solución de un problema. Como se ve, hay tantas clases de algoritmos como de interlocutores.

Hablamos de niveles de interlocutores según su grado de comprensión de las acciones de una tarea. Los interlocutores que entiendan los algoritmos de una menor cantidad de acciones (o, lo que es lo mismo, que puedan ejecutar acciones más complejas sin una descomposición de las mismas) los llamaremos interlocutores de ***alto nivel***. Si un interlocutor necesita un algoritmo muy refinado para poder ejecutarlo diremos que es de ***bajo nivel***.

Los términos alto nivel y bajo nivel no otorgan una connotación peyorativa a este último. Piénsese en términos humanos y se descubrirá que un abogado va a ser un interlocutor de bajo nivel cuando se describa una tarea ingenieril; de la misma manera, un ingeniero será un interlocutor de bajo nivel cuando se trate de resolver un problema jurídico.

Tipos de acciones

Las acciones pueden ser de tres tipos básicos: de ***secuencia***, de ***selección*** y de ***iteración*** o de repetición.

Una serie de acciones está en secuencia cuando se deben ejecutar una tras otra hasta terminar. Es el caso de todos los métodos propuestos más arriba para la resolución del problema del neumático pinchado.

Sin embargo, la acción 1.4.2 se pudo haber expresado como:

```
1.4.2.  Repetir
            ajustar 1 tuerca
        hasta que estén todas ajustadas
```

En este caso estaremos utilizando una acción de iteración.

También se pudo haber comenzado el problema con la acción:

0. Si el auto está andando
   entonces detenerlo en la banquina
            encender las balizas
   fin si

Y esta sería una acción de selección.

⇨ Se puede demostrar que todo algoritmo se puede describir con acciones de secuencia, selección e iteración. Ver el teorema de Bohm-Jacopini, en el capítulo 2.

Condiciones lógicas

Una ***condición lógica***, o simplemente condición, es una expresión que puede ser verdadera o falsa en un momento dado.

Son condiciones, en el ítem anterior:

"estén todas ajustadas" o
"el auto está andando"

Nótese que estas expresiones pueden ser verdaderas o falsas, según el momento en que se evalúen.

Estas condiciones pueden ser compuestas cuando se utilizan conectivos lógicos, como *y*, *o* o *no*. Por ejemplo:

"*no* estén todas ajustadas" tiene el valor contrario a "estén todas ajustadas".

"el auto está andando *y* hay un neumático pinchado" sólo va a ser verdadera si ambas condiciones simples ("el auto está andando" y "hay un neumático pinchado") son verdaderas.

"hay un neumático pinchado *o* hay un neumático desinflado" sólo va a ser falsa si ambas condiciones simples ("hay un neumático pinchado" y "hay un neumático desinflado") son falsas.

Más adelante se profundiza más en estos conceptos.

Pseudocódigo

Cuando se trabaja con algoritmos para ser programados para una computadora, se deben seguir ciertas reglas sintácticas y semánticas para que la traducción de algoritmo a programa se pueda hacer de manera sencilla. Nuestro interlocutor será similar a un traductor de un lenguaje de programación.

Utilizaremos un ***pseudocódigo*** (falso código o falso lenguaje) que iremos definiendo en este ítem y los siguientes.

Cada algoritmo de nuestro pseudocódigo comenzará con un título, por ejemplo:

Algoritmo CAMBIAR RUEDA

y terminará con un delimitador *fin*, por ejemplo:

fin CAMBIAR RUEDA.

Nuestro algoritmo lo escribiremos poniendo una acción por renglón, cuando éstas estén en secuencia.

Las acciones de selección las representaremos por:

```
si condición
entonces acciones en secuencia
sino acciones en secuencia
fin si
```

Esto significa: si la condición es verdadera, hacer las acciones en secuencia del *entonces*; si no lo es, hacer las acciones del *sino*. Por ser acciones en secuencia, tanto las del *entonces* como las del *sino* se escribirán una por renglón. La parte del *sino* puede omitirse.

Las acciones de repetición se escribirán así:

```
mientras condición hacer
    acciones en secuencia
fin mientras
```

Esto significa que si la condición es verdadera se ejecutan las acciones en secuencia y se vuelve a evaluar dicha condición. Si sigue siendo verdadera, se vuelven a ejecutar, y así sucesivamente. Cuando la condición evaluada sea falsa, se seguirá en secuencia con el algoritmo.

Los conectivos lógicos *y*, *o* y *no* los representaremos con los símbolos: ^, *v* y ~.

Constantes, variables, expresiones y asignación

En programación, todo elemento de datos cuyo valor no puede variar se denomina ***constante***. Las constantes pueden tener o no un nombre. Por ejemplo:

44 es una constante anónima

si escribimos "constante PI = 3.141592654", *PI* es una constante con nombre (de valor 3.141592654).

Un elemento de datos cuyo valor puede variar se denomina ***variable***. Toda variable tiene un nombre que la identifica y un tipo que describe su uso (el concepto de tipo lo veremos más adelante).

Una ***expresión*** es una combinación de constantes y variables conectadas mediante operadores, que tiene por resultado un valor. Por ejemplo, son expresiones:

8 + T
3 + (Y*5 - 2)

Como hemos dicho, el valor de una variable puede cambiar. Es por eso que no se puede darle un valor fijo, como se hace con las constantes. Para darle un valor a una variable se usa una acción denominada ***asignación***. La sintaxis que usaremos para la asignación será:

variable ← expresión

Esto provocará que se evalúe la expresión y su valor se almacene en la variable. Por ejemplo, analicemos las siguientes acciones:

1. V ← 4
2. W ← X + 5
3. Z ← Z + 1

La acción 1 provocará que se almacene el valor 4 en la variable V. En la acción 2 , al valor que tuviera la variable X se le suma 5 y luego se almacena el resultado en la variable W. En la acción 3, al valor que tuviera la variable Z se le suma 1 y el resultado se almacena en Z; o, lo que es lo mismo, se incrementa en 1 el valor de Z. Los valores de las variables no cambian por el solo hecho de ser usadas; por ejemplo, la variable X conserva su valor anterior luego de la acción 2.

Se puede pensar en una variable como una caja que puede contener un valor por vez. Antes de la primera asignación la caja está vacía y no se puede usar su valor (ya que no existe). Cada vez que se le asigna un valor se saca de la caja el valor anterior.

Ejercicio 1.1

Enunciado:

Escribir un algoritmo que sume los primeros diez números naturales.

Solución:

Pensemos en cómo lo resolveríamos en forma manual. Una primera forma sería anotar los 10 números en forma encolumnada y luego proceder a sumarlos. Sin embargo, como veremos más adelante, nuestros algoritmos sólo podrán sumar los valores tomándolos de a dos.

Por lo tanto, debemos operar de la siguiente manera: tomaremos los dos primeros números y los sumamos, anotando este primer total en un papel; luego, a este total le sumamos el tercer número y lo anotamos tachando el total anterior; así seguimos el proceso cambiando en cada paso el total hasta que hayamos sumado los diez primeros.

Si pensamos en los términos en que estamos trabajando debemos hablar de una variable donde acumularemos el total y otra en la que se van colocando cada uno de los diez números a sumar.

Una primera versión podría ser:

```
Algoritmo SUMAR 10 NATURALES
    SUMA ← 1                    -- tomamos el primer número y lo anotamos en SUMA
    NÚMERO ← 2                  -- tomamos el número siguiente
    SUMA ← SUMA + NÚMERO        -- se lo sumamos a la SUMA
    NÚMERO ← 3                  -- y así sucesivamente...
    SUMA ← SUMA + NÚMERO
    NÚMERO ← 4
    SUMA ← SUMA + NÚMERO
    NÚMERO ← 5
    SUMA ← SUMA + NÚMERO
    NÚMERO ← 6
    SUMA ← SUMA + NÚMERO
    NÚMERO ← 7
    SUMA ← SUMA + NÚMERO
    NÚMERO ← 8
    SUMA ← SUMA + NÚMERO
    NÚMERO ← 9
    SUMA ← SUMA + NÚMERO
    NÚMERO ← 10
    SUMA ← SUMA + NÚMERO        -- en SUMA tenemos lo que queríamos
fin SUMAR 10 NATURALES.
```

Pero utilizando una acción de iteración, el algoritmo se simplifica bastante, quedando así:

```
Algoritmo SUMAR 10 NATURALES
    NÚMERO ← 1                  -- tomamos el primer número
    SUMA ← NÚMERO               -- lo anotamos en SUMA
    mientras NÚMERO < 10 hacer
        NÚMERO ← NÚMERO + 1     -- generamos el número siguiente
        SUMA ← SUMA + NÚMERO    -- se lo sumamos al acumulado en SUMA
    fin mientras                -- en SUMA tenemos lo que queríamos
fin SUMAR 10 NATURALES.
```

Hemos utilizado el doble guión, --, como se hace en el lenguaje Ada, para indicar que lo que sigue es un comentario, que se escribe para aclarar el algoritmo, pero que no es parte del mismo.

También podemos modificar el primer paso, para que quede un algoritmo más sencillo. ¿Por qué no partir de un total en cero (0) e irle sumando cada valor a este total desde la primera vez?

El algoritmo resultante es el que sigue:

```
Algoritmo SUMAR 10 NATURALES
```

```
    NÚMERO ← 0
    SUMA ← 0
    mientras NÚMERO < 10 hacer
        NÚMERO ← NÚMERO + 1
        SUMA ← SUMA + NÚMERO
    fin mientras
fin SUMAR 10 NATURALES.
```

En este sencillo ejercicio aparecen dos técnicas de mucho uso. La primera de ellas, la del acumulador. Las variables *NÚMERO* y *SUMA* son dos acumuladores. Como vimos en el ítem anterior, a ambas se las está incrementando en cada ciclo: a la variable *NÚMERO* se la incrementa siempre en 1 y a la variable *SUMA* se la incrementa en un valor diferente cada vez. Precisamente en este acumulador *SUMA* queda el resultado que se pretendía que calculara el algoritmo. Por ser acumuladores, ambos deben ser inicializados en 0.

Otra técnica que aparece es la de introducir un contador (acumulador de incremento unitario) para evaluar la cantidad de veces que se está ejecutando un ciclo. Éste es el caso de la variable *NÚMERO*. Cuando ésta llega a 10, esto implica que el ciclo se ha ejecutado 10 veces. Por eso el encabezamiento del ciclo dice *mientras NÚMERO < 10*.

Se podría generalizar el algoritmo para que, en vez de sumar los primeros 10 números naturales, sumara los *N* primeros, pudiéndose modificar el valor de *N*. En una primera aproximación éste quedaría así:

```
Algoritmo SUMAR N NATURALES
    constante N = 10
    NÚMERO ← 0
    SUMA ← 0
    mientras NÚMERO < N hacer
        NÚMERO ← NÚMERO + 1
        SUMA ← SUMA + NÚMERO
    fin mientras
fin SUMAR N NATURALES.
```

Para sumar los primeros 15 naturales bastará con cambiar el valor de la constante N poniéndole 15.

⇨ Nótese que estamos utilizando una sangría para indicar las estructuras de control. Ésta es una práctica sencilla para hacer más legibles los algoritmos.

Entrada y salida

Se denomina ***salida*** o ***escritura*** a la acción que tiene por objeto comunicar un valor del ambiente al medio externo. Las computadoras suelen comunicar los resultados de sus ejecuciones a los usuarios (mediante listados en papel, mensajes en pantalla, etc.), a otras computadoras en forma inmediata o diferida (mediante la escritura de datos en cintas magnéticas o discos, envío de datos por cable, etc.) o a otros dispositivos (por ejemplo, un programa que mueve un brazo mecánico de un robot).

⇨ No tendría sentido escribir algoritmos que efectuaran toda una serie más o menos compleja de tareas si éstas no son comunicadas de una forma u otra al medio externo. Por ejemplo, el algoritmo que escribimos en el ejercicio 1.1 no tiene ninguna utilidad precisamente por esto que acabamos de decir: nadie puede conocer el resultado del cálculo que realizó.

Para indicar una salida o escritura escribiremos:

escribir <lista de expresiones>

Las expresiones se separarán entre sí por comas y los literales se colocarán entre comillas. Por ejemplo, el ejercicio anterior podría quedar así:

```
Algoritmo SUMAR N NATURALES
    constante CANTIDAD = 10
    NÚMERO ← 0
    SUMA ← 0
    mientras NÚMERO < CANTIDAD hacer
        NÚMERO ← NÚMERO + 1
        SUMA ← SUMA + NÚMERO
    fin mientras
    escribir "La suma de los primeros ", CANTIDAD," números naturales es ", SUMA
fin SUMAR N NATURALES.
```

Se denomina ***entrada*** o ***lectura*** a la acción que tiene por objeto introducir un valor al ambiente desde el medio externo. Así, a una computadora se le dan datos a través de un teclado, de un "mouse" o ratón, de un lápiz óptico, de un cable que proviene de otra computadora o dispositivo, de un disco magnético, etc. Toda lectura se hace asignándole luego ese valor a una variable.

Para indicar una entrada o lectura escribiremos:

leer <lista de variables>

y ello tendrá por objeto el ir tomando valores del medio externo e ir asignándoselos a cada una de las variables de la lista.

⇨ Casi todos los algoritmos deben leer datos. En el del ejercicio 1.1, por ejemplo, una lectura de datos le proporcionaría generalidad. Hemos visto que si le agregábamos una constante, se podría modificar la cantidad de valores a sumar con sólo cambiar esa constante. Sin embargo, ello significaría una modificación del algoritmo cada vez. Una forma mejor sería la siguiente:

```
Algoritmo SUMAR N NATURALES
    leer CANTIDAD
    NÚMERO ← 0
    SUMA ← 0
    mientras NÚMERO < CANTIDAD hacer
        NÚMERO ← NÚMERO + 1
        SUMA ← SUMA + NÚMERO
    fin mientras
    escribir "La suma de los primeros ", CANTIDAD," números naturales es ", SUMA
fin SUMAR N NATURALES.
```

En este caso el programa no deberá ser modificado cada vez que se quiera sumar una cantidad diferente de números. Simplemente, durante la ejecución del mismo se informará esa cantidad.

A lo largo de todo el capítulo se supondrá que las entradas de datos se realizan desde el teclado de una computadora y las salidas a través de una pantalla o una impresora.

Ejercicio 1.2

Enunciado:

Dada una serie de números que se lee, informar si se encuentran en orden estrictamente creciente, interrumpiendo el proceso cuando se encuentre el primer valor desordenado. El lote de datos termina con el valor 999 y se debe suponer que ningún valor en el lote supera a éste.

Solución:

```
Algoritmo ORDEN
    CRECIENTE ← Verdadero
    leer NUM
    NUMANTERIOR ← NUM          -- guarda siempre el valor anterior
```

```
    mientras CRECIENTE = Verdadero hacer
        NUMANTERIOR ← NUM        -- guarda el valor anterior antes de leer otro
        leer NUM
        si NUMANTERIOR < NUM
        entonces CRECIENTE ← Verdadero
        sino CRECIENTE ← Falso
        fin si
    fin mientras
    si CRECIENTE = Verdadero
    entonces escribir "La secuencia está en orden creciente"
    sino escribir "La secuencia no está en orden creciente"
fin ORDEN.
```

En realidad, mirando el algoritmo nos damos cuenta de que el *si NUMANTERIOR < NUM* no necesita de la parte *entonces*, ya que podría escribirse, sin alterar el funcionamiento:

```
si NUMANTERIOR < NUM
entonces        -- no hacemos nada
sino CRECIENTE ← Falso
fin si
```

Como esto no queda demasiado elegante, podemos dar vuelta la condición del *si*, quedando:

```
si NUMANTERIOR >= NUM
entonces CRECIENTE ← Falso
fin si
```

El ejercicio anterior hace uso de ***acciones anidadas***. Por ejemplo, el ciclo *mientras* tiene dentro una acción de selección *si*. Cuando se escriben algoritmos en pseudocódigo es muy común hacer uso de acciones anidadas unas dentro de otras, y no existe restricción al respecto: se pueden anidar acciones de selección entre sí, acciones de repetición entre sí, acciones de selección dentro de acciones de repetición y viceversa.

También nos muestra la conveniencia del uso de variables y expresiones lógicas para expresar algoritmos de manera más sencilla.

Dado que las construcciones *mientras* y *si* evalúan condiciones lógicas, y dado que expresiones como *NUMANTERIOR < NUM* tienen un resultado lógico, el algoritmo en su primera versión pudo haber sido escrito así:

```
Algoritmo ORDEN
    CRECIENTE ← Verdadero
    leer NUM
    NUMANTERIOR ← NUM
```

```
    mientras CRECIENTE hacer        -- CRECIENTE = Verdadero
        NUMANTERIOR ← NUM
        leer NUM
        CRECIENTE ← NUMANTERIOR < NUM       -- reemplaza al si del primer algoritmo
    fin mientras
    si CRECIENTE                    -- CRECIENTE = Verdadero
    entonces escribir "La secuencia está en orden creciente"
    sino escribir "La secuencia no está en orden creciente"
fin ORDEN.
```

Subalgoritmos

Necesariamente, cuando introducimos un repertorio de instrucciones como lo estamos haciendo, suponemos un cierto nivel de interlocutor. Pero, ¿qué ocurre cuando queremos escribir algoritmos más complejos que nos resulta más sencillo escribirlos para un interlocutor de mayor nivel? Un caso de esto podría ser el de la construcción de un algoritmo en nuestro pseudocódigo para que un robot cambie el neumático pinchado de un auto.

Como vimos más arriba, no es tan difícil hacerlo si se van escribiendo acciones no primitivas y se les van haciendo refinamientos sucesivos. Luego, reconstruyendo la cadena de acciones primitivas resultantes se puede llegar al algoritmo buscado.

Sin embargo, cada vez que se quisiera introducir una modificación en el algoritmo, esto demandaría volver a construir las acciones no primitivas.

Obviamente, lo más cómodo sería conservar toda la estructura de acciones no primitivas. En nuestro pseudocódigo lo haremos mediante subalgoritmos. En el algoritmo escribiremos las acciones no primitivas directamente y luego las refinaremos en subalgoritmos. Un subalgoritmo lo desarrollaremos igual que un algoritmo. Por ejemplo, podríamos escribir el algoritmo antedicho de la siguiente forma:

```
Algoritmo CAMBIAR RUEDA PINCHADA
    LEVANTAR AUTO
    QUITAR RUEDA PINCHADA
    TRAER RUEDA AUXILIO
    COLOCAR RUEDA AUXILIO
    BAJAR AUTO
    GUARDAR RUEDA PINCHADA
fin CAMBIAR RUEDA PINCHADA.
```

```
Acción LEVANTAR AUTO
    TRAER EL GATO
    COLOCAR EL GATO EN LA RANURA
    SUBIR EL GATO HASTA QUE LA RUEDA SE SEPARE DEL SUELO
    PONER LA RUEDA EN POSICIÓN
    AJUSTAR LAS 4 TUERCAS
fin LEVANTAR AUTO.

Acción QUITAR RUEDA PINCHADA
    ...
    etc.
```

... y desarrollando todas las acciones no primitivas en subalgoritmos hasta llegar al nivel del pseudocódigo que estamos utilizando habremos resuelto el problema.

Ejercicio 1.3

Enunciado:

Escribir un algoritmo que resuelva una ecuación de segundo grado del tipo $Ax^2 + Bx + C = 0$, contemplando todos los casos posibles de raíces.

Solución:

```
Algoritmo ECUACIÓN SEGUNDO GRADO
    LEER COEFICIENTES
    EVALUAR DISCRIMINANTE
    HALLAR RAÍCES
fin ECUACIÓN SEGUNDO GRADO

Acción LEER COEFICIENTES
    leer A, B, C
fin LEER COEFICIENTES

Acción EVALUAR DISCRIMINANTE
    DISCRIMINANTE ← B*B - 4*A*C
    si DISCRIMINANTE < 0
    entonces COMPLEJAS ← Verdadero
    sino    COMPLEJAS ← Falso
            si DISCRIMINANTE = 0
            entonces IGUALES ← Verdadero
            sino IGUALES ← Falso
            fin si
    fin si
```

```
fin EVALUAR DISCRIMINANTE

Acción HALLAR RAÍCES
    si COMPLEJAS = Verdadero
    entonces   escribir "Hay dos raíces complejas conjugadas"
               RADICANDO ← (-1) * DISCRIMINANTE
               EVALUAR RAÍZ
               REAL ← -B / (2*A)
               IMAG ← RAÍZ / (2*A)
               escribir "Parte real: ", REAL
               escribir "Parte imaginaria: ±", IMAG
    sino       si IGUALES
               entonces   escribir "Hay dos raíces reales iguales"
                          VALOR ← -B / (2*A)
                          escribir "Valor de las raíces: ", VALOR
               sino       escribir "Hay dos raíces reales diferentes"
                          RADICANDO ← DISCRIMINANTE
                          EVALUAR RAÍZ
                          RAÍZ1 ← (-B + RAÍZ) / (2*A)
                          RAÍZ2 ← (-B - RAÍZ) / (2*A)
                          escribir "Primera raíz: ", RAÍZ1
                          escribir "Segunda raíz: ", RAÍZ2
        fin si
    fin si
fin HALLAR RAÍCES

Acción EVALUAR RAÍZ
    RAÍZ ← √(RADICANDO)
fin EVALUAR RAÍZ
```

Tipos de datos y funciones

Más arriba dijimos que una variable siempre debe tener dos atributos: un nombre que la identifica y un tipo que describe su uso. Por ello, una variable será de un determinado tipo: numérica, una cadena de caracteres, etc. Asimismo, las constantes y las expresiones también tienen un tipo.

Un tipo de datos describe los valores que un dato de ese tipo puede tomar y cómo se trabaja con ellos. Por eso, un tipo es un conjunto de operandos más un conjunto de operaciones.

En pseudocódigo trabajaremos con tres tipos de datos: numérico, cadena de caracteres y lógico.

El tipo de datos numérico tiene como conjunto de operandos a todos los números. Como conjunto de operaciones binarias (se aplican a dos expresiones numéricas colocando sus símbolos entre ambas) tiene a la suma, resta, producto, división y exponenciación, representados por los símbolos: **+**, **-**, *, /, y **^**. Las operaciones monarias (se aplican a una sola expresión numérica) son el cambio de signo, el valor absoluto y la parte entera, cuyas aplicaciones a un valor numérico *x* las representaremos por *-x*, *|x|* y *[x]*. Existen también operaciones que tienen por resultado un valor lógico: **=**, **#**, **>**, **<**, ≥ y ≤ (igual, distinto, mayor, menor, mayor o igual y menor o igual). Las constantes numéricas se representarán por una serie de dígitos precedidas o no de un signo menos y con o sin punto decimal. Por ejemplo, son constantes numéricas: 0, 97, -33, 22.5, -1.3474.

El tipo de datos cadena o hilera de caracteres tiene como conjunto de operandos a todos las posibles cadenas y como operación a la concatenación, representada por el símbolo //. Existen también operaciones que tienen por resultado un valor lógico: **=**, **#**, **>**, **<**, ≥ y ≤. Las constantes cadena de caracteres se representarán por una serie de caracteres entre comillas. Por ejemplo, son constantes cadena: "hola", "¿qué tal?", "La suma de los primeros ", "somos 15".

El tipo de datos lógico tiene como conjunto de operandos a los valores lógicos verdadero y falso (representados como *Verdadero* y *Falso*) y como conjunto de operaciones a los introducidos en el ítem "Condiciones lógicas", además de la igualdad y la desigualdad.

En los algoritmos escritos en este pseudocódigo no se podrán utilizar variables o constantes de distintos tipos y del mismo nombre.

Una función realiza una serie de tareas orientadas al cálculo de un valor. En nuestro pseudocódigo trabajaremos con las siguientes (se muestra un ejemplo de uso de cada función):

Función	Tipo argumento	Tipo resultado	Significado	
ArcTan(X)	numérico	numérico	arcotangente	arctg X
Cos(X)	numérico	numérico	coseno	cos X
Exp(X)	numérico	numérico	exponencial	e^X
Ln(X)	numérico	numérico	log. natural	ln X
Log(X)	numérico	numérico	log. decimal	log X
Long(C)	cadena	numérico	longitud	
Sen(X)	numérico	numérico	seno	sen X
X div Y	numéricos enteros	numérico entero	división entera	
X mod Y	numéricos enteros	numérico entero	resto de división entera	

Hay ciertas funciones que las utilizaremos como operadores binarios, como ya venimos haciendo con la suma, resta, concatenación, etc. Ellas serán la división entera y el resto de la división entera.

Ejercicio 1.4

Enunciado:

Escribir un algoritmo que lea dos reales *X* e *Y*, y efectué la potencia X^Y utilizando logaritmos.

Solución:

Si llamamos *Z* al resultado que queremos hallar:

$Z = X^Y \Rightarrow \ln Z = Y * \ln X \quad \Rightarrow Z = e^{(Y * \ln X)}$

```
Algoritmo POTENCIA
    leer X, Y
    Z ← Exp (Y*Ln(X))
    escribir Z
fin POTENCIA
```

Ejercicio 1.5

Enunciado:

Escribir un algoritmo que lea un texto letra por letra y determine cuántas veces ocurre que la primera letra de una palabra se vuelve a repetir en la misma. Las palabras se encuentran separadas por uno ó más de los siguientes caracteres: blanco, coma, punto y coma y punto. El texto termina con el carácter "!".

Solución:

```
Algoritmo LETRAS REPETIDAS
    VECES ← 0
    leer C
    mientras C # "!" hacer
        PROCESAR PALABRA
        SALTEAR SEPARADORES
    fin mientras
    escribir "La cantidad de veces es ",VECES
fin LETRAS REPETIDAS.

Acción PROCESAR PALABRA
    PRIMERA ← C
    REPITE ← Falso
    leer C
    mientras (C # " ") ^ (C # ",") ^ (C # ".") ^ (C # "!") hacer
        si (REPITE = Falso) ^ (C = PRIMERA)
        entonces   REPITE ← Verdadero
```

```
                    VECES ← VECES + 1
            fin si
            leer C
        fin mientras
fin PROCESAR PALABRA.

Acción SALTEAR SEPARADORES
    mientras (C = " ") v (C = ",") v (C = ".") hacer
        leer C
    fin mientras
fin SALTEAR SEPARADORES.
```

A este ejercicio lo vamos a ir haciendo más complejo a través de los siguientes capítulos. Por ejemplo, ¿qué ocurriría si se pidiera ver cuántas veces ocurre que una letra cualquiera de una palabra está repetida? ¿Podemos resolverlo con los elementos de este capítulo?

Otras estructuras de control

En nuestro pseudocódigo existen otras acciones de iteración y de selección que no hemos usado por no ser estrictamente necesarias. Sin embargo, en muchas ocasiones permitirán escribir algoritmos con una mayor naturalidad.

La primera de ellas es el ciclo con condición a la salida:

```
repetir
    acciones en secuencia
hasta que condición
```

y ejecuta las acciones en secuencia repetidamente hasta que se cumpla la condición de salida; luego sigue con la próxima acción del algoritmo.

La segunda es el ciclo con contador:

```
para variable desde expresión1 hasta expresión2
    acciones en secuencia
fin para
```

y equivale al siguiente ciclo mientras:

```
variable ← expresión1
mientras variable ≤ expresión2
    acciones en secuencia
    variable ← variable + 1
fin mientras
```

La variable y las expresiones deberán ser numéricas y adoptar valores enteros solamente. Ninguna de ellas podrá ser modificada dentro del ciclo.

La tercera es la selección de varias salidas:

```
según expresión hacer
    valor1 : acciones en secuencia
    valor2 : acciones en secuencia
    valor3 : acciones en secuencia
    etc.
    otro valor : acciones en secuencia
fin según
```

y ejecuta las acciones que correspondan según el valor que tome la expresión, siguiendo luego con la próxima acción del algoritmo. Es una estructura de mucha utilidad para evitar las decisiones anidadas.

Ejercicio 1.6

Enunciado:

Dado un texto terminado en punto, determinar cuántas veces aparece cada vocal minúscula en el mismo.

Solución:

```
Algoritmo VOCALES
    constante FIN = "."
    CA ← 0
    CE ← 0
    CI ← 0
    CO ← 0
    CU ← 0
    leer C
```

```
    mientras C # FIN hacer
        según C hacer
            "a" : CA ← CA+1
            "e" : CE ← CE+1
            "i" : CI ← CI+1
            "o" : CO ← CO+1
            "u" : CU ← CU+1
        fin según
        leer C
    fin mientras
    escribir "  Vocal    Cantidad de apariciones"
    escribir "  a              ", CA
    escribir "  e              ", CE
    escribir "  i              ", CI
    escribir "  o              ", CO
    escribir "  u              ", CU
fin VOCALES.
```

Cuando se vea el manejo de arreglos, este ejercicio podrá ser resuelto de una manera mucho más cómoda. Este ejemplo fue, más que nada, para ver el uso de la selección por caso.

Algunas observaciones finales del capítulo 1

Cuando se trabaja con acciones de iteración se debe tener en cuenta lo siguiente:

- Se debe progresar hacia el objetivo en cada repetición, acercándose a la condición de fin. Como corolario, la condición de fin debe estar afectada de algún modo por el bloque de la iteración. Por ejemplo, la siguiente sentencia posee una falla:

```
mientras I > 0 hacer
    K ← 2 * K
fin mientras
```

- Se debe evitar todo cálculo repetitivo de una expresión, ahorrando así tiempo de ejecución. Por ejemplo, la iteración:

```
mientras I < 3*N hacer
    T ← X + Y*Z + Z*I
    escribir T
    I ← I+1
fin mientras
```

se puede mejorar así:

```
N3 ← 3*N;
U ← X + Y*Z;
mientras I < N3 hacer
    T ← U + Z*I
    escribir T
    I ← I+1
fin mientras
```

Esta recomendación se debe tomar con cuidado. No se debe buscar eficiencia en tiempo de máquina cuando esto signifique menor claridad del texto del programa.

Se deben evitar los anidamientos muy profundos en la medida de lo posible. La siguiente secuencia:

```
mientras COND1 hacer
    si COND2
    entonces   repetir
               ACCIÓN1
               mientras COND3 hacer
                   si COND4
                   entonces ACCIÓN2
                   fin si
               fin mientras
               hasta que COND5
    fin si
fin mientras
```

se vuelve difícil de seguir, y no es fácil determinar bajo qué condiciones se va a ejecutar la acción *ACCIÓN2*. Un buen uso de subalgoritmos clarificaría mucho este segmento de algoritmo.

También se deben tener en cuenta las reglas de De Morgan para negar condiciones lógicas:

~ (P ^ Q) es lo mismo que ~P v ~Q
~ (P v Q) es lo mismo que ~P ^ ~Q