APÉNDICE – DIAGRAMACIÓN

El presente material es un resumen del libro "**HolaMundo.pascal**" (HolaMundoPascal.blogspot.com), escrito por **Pablo Augusto Sznajdleder**. Se proporciona como lectura complementaria para el libro "**Análisis y Diseño de Algoritmos**", escrito por **Gustavo López**, **Ismael Jeder** y **Augusto Vega**.

Introducción.

Un diagrama de flujo es una representación gráfica de un algoritmo. El objetivo de utilizar diagramas es proveer (a nosotros mismos o a terceros) una visión más amplia, simplificada e independiente de cualquier lenguaje de programación sobre el funcionamiento del algoritmo en cuestión.

Bien utilizados, los diagramas permiten exponer la estrategia de resolución de un algoritmo –omitiendo detalles complejos de implementación– o bien, si fuese necesario, permiten también entrar en detalle casi a nivel de código fuente. El nivel de detalle que debemos proveer con un diagrama debe ser suficiente, de matera tal que permita comunicar unívocamente la lógica de resolución de nuestro algoritmo.

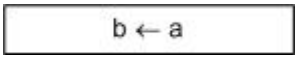
Existen diferentes tipos de diagramas con los que se pueden representar algoritmos. En este capítulo utilizaremos la diagramación **Chapín** que, según consideramos, es una excelente herramienta para graficar algoritmos estructurados utilizando la metodología de desarrollo **top-down**.

Recordemos que según el Teorema de la Programación Estructurada, todo problema computacional puede resolverse mediante la aplicación de tres tipos de estructuras de control: **Acciones Simples**, **Acciones de Decisión** y **Acciones Iterativas**.

A lo largo de este capítulo, analizaremos diferentes ejemplos con los que aprenderemos a representar mediante diagramas los diferentes casos que se pueden dar en el diseño y desarrollo de algoritmos.

Representación de Acciones Simples.

Llamamos “Acción Simple” a la acción de **escribir** (en pantalla, impresora o archivo), **leer** (de teclado o archivo) y **asignar** valor a una variable. Cada una de estas acciones las representamos de la siguiente forma:

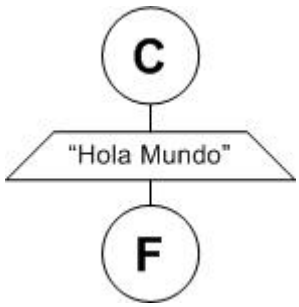
Acción Simple	Símbolo	Descripción
Salida (o “escritura”)		Muestra por pantalla la cadena “Hola”.
Entrada (o “lectura”)		Lee un valor que se ingresa por teclado, asignándolo en la variable **a**.
Asignación (o “acción”)		Asigna a la variable **b** el valor de la variable **a**.

También indicaremos el comienzo y el fin del algoritmo con los siguientes símbolos:

Comienzo		Indica el comienzo del algoritmo.
Fin		Indica el final del algoritmo.

Ejemplo 1.

Desarrollar un algoritmo que imprima por pantalla la cadena “Hola Mundo”.

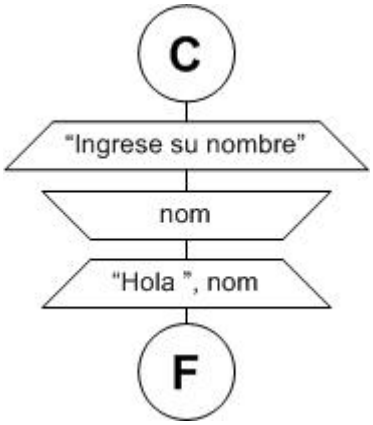


La solución a este problema es trivial: El algoritmo comienza, imprime la leyenda “Hola Mundo” y finaliza.

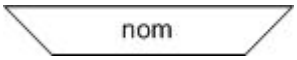
Ejemplo 2.

Desarrollar un algoritmo que permita al usuario ingresar su nombre y luego emita un mensaje de salutación.

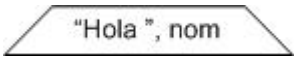
Para resolver este problema debemos emitir un mensaje que le indique al usuario que debe ingresar su nombre. Luego, debemos leer (por teclado) el nombre del usuario, asignarlo a una variable y, por último, mostrar el contenido de la variable anteponiendo el saludo.



En el diagrama, luego de emitir el mensaje que le indica al usuario que ingrese su nombre, leemos por teclado y asignamos el nombre ingresado por el usuario en la variable **`nom`**. Esto queda reflejado en el siguiente símbolo:



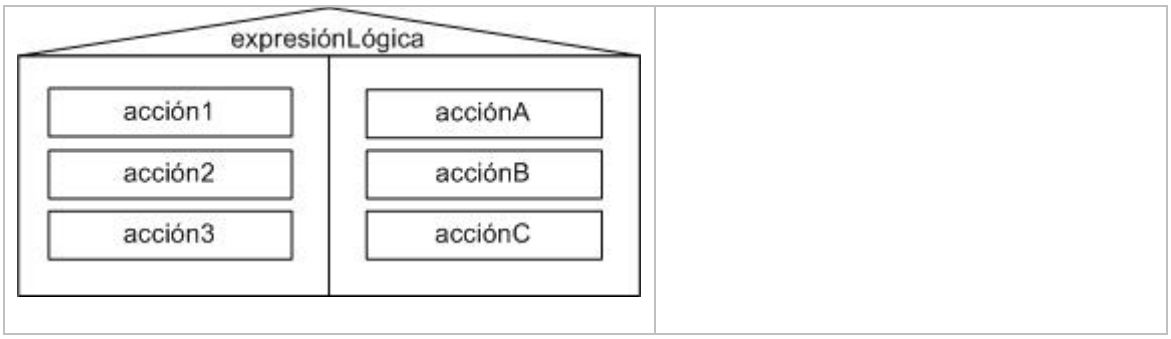
Luego, con el nombre asignado en la variable **`nom`**, mostramos el saludo, combinando en un símbolo de salida la cadena literal “Hola” seguida del valor de la variable **`nom`**:



Representación de la Acción Condicional.

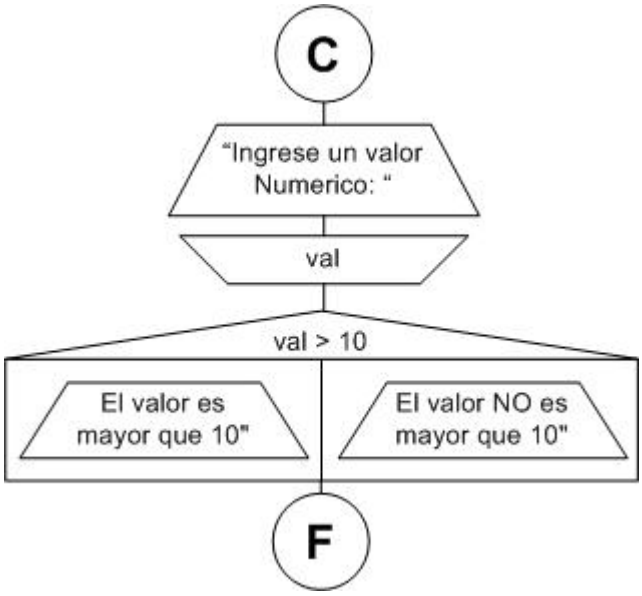
Llamamos “Acción Condicional” o “Estructura de Decisión” a aquella estructura que permite decidir entre ejecutar uno u otro conjunto de acciones. La representamos así:

	Dependiendo del valor de **`expresiónLógica`**, decide entre ejecutar las acciones ubicadas en la parte izquierda (**`true`**) o en la parte derecha (**`false`**) de la estructura.



Ejemplo 3.

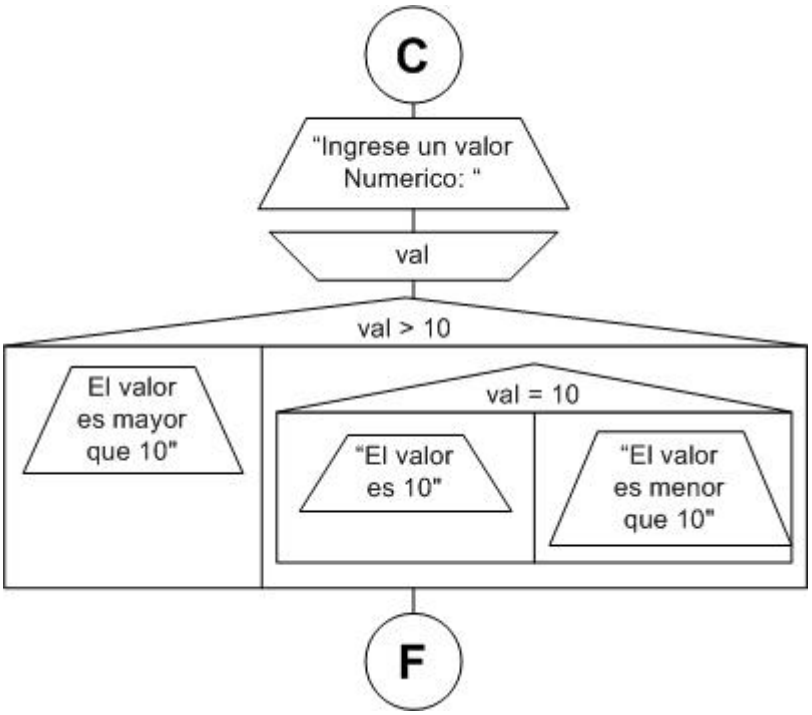
Leer por teclado un valor. Indicar si el valor leído es mayor que 10.



Utilizamos una Estructura Condicional (o Estructura de Decisión) para preguntar si el valor ingresado por el usuario es mayor que 10. La expresión lógica **`val > 10`** tiene un valor de verdad. Éste puede ser verdadero (**`true`**) o falso (**`false`**). Si la expresión se verifica, es decir que su valor de verdad es **`true`**, entonces se ejecutará la acción ubicada a la izquierda de la estructura, indicando al usuario que el valor que ingresó es mayor que 10. Si la condición no se verifica, se ejecutará la acción ubicada a la derecha de la estructura, mostrando un mensaje que indica que el valor ingresado no es mayor que 10. Obviamente, no podemos asegurar que sea menor, también podría ser igual.

Ejemplo 4.

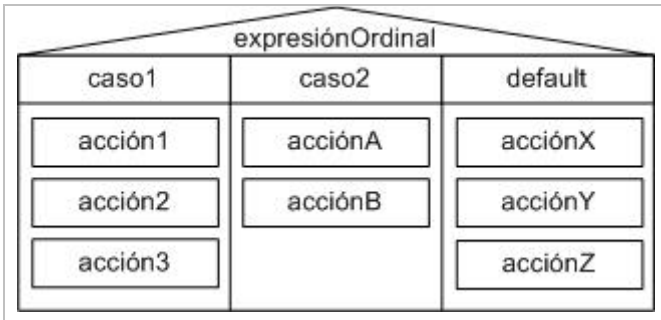
Ídem ejemplo anterior, pero ahora indicar si el valor ingresado es mayor, menor o igual que 10.



Para resolver este problema utilizamos Estructuras de Decisión anidadas. Primero preguntamos si el valor ingresado es mayor que 10. Si se verifica, el problema terminó. Si no se verifica, tenemos que preguntar si el valor ingresado es 10. Si se verifica, el problema queda resuelto y si no, por descarte, podemos concluir que el valor ingresado es menor que 10.

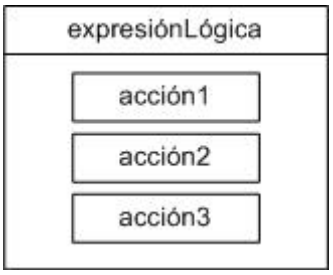
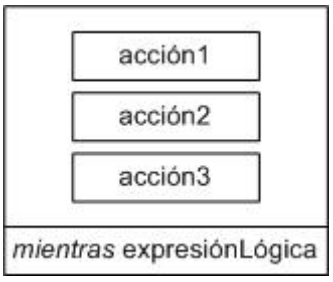
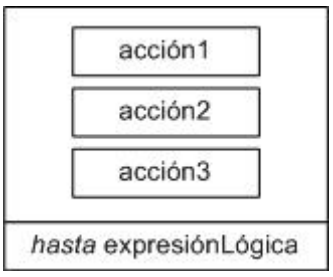
Decisión Múltiple.

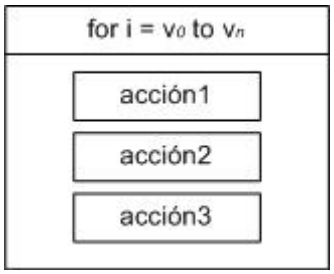
Los lenguajes de programación proveen estructuras de Decisión Múltiple que, en ocasiones, simplifican el uso de la estructura de Decisión Simple. Se representa de la siguiente manera:

	Dependiendo del valor de **`expresiónOrdinal`**, decide entre ejecutar las acciones ubicadas debajo del "caso" correspondiente, o bien, debajo de "*default*" si su valor no coincide con ninguno de los "casos" especificados. **`expresiónOrdinal`** puede ser un caracter o un valor entero.

Representación de la Acción Iterativa.

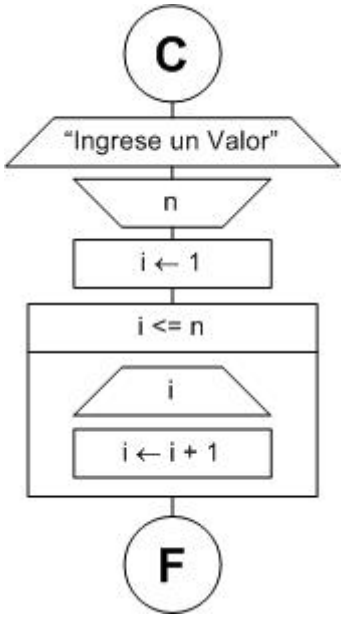
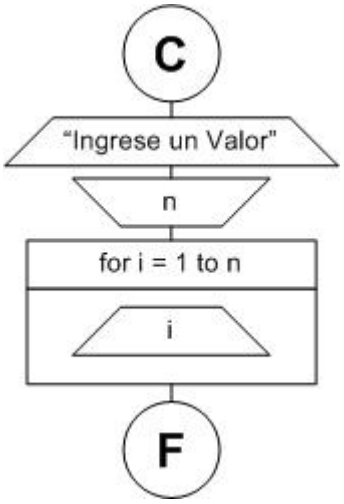
La "Acción Iterativa" o la "Estructura de Repetición" es aquella que permite repetir la ejecución de un conjunto de acciones *n* veces, con *n*>=0.

Acción Iterativa	Símbolo	Descripción
`while`		Mientras se cumpla la **`expresiónLógica`**, se repetirán las acciones 1, 2 y 3. Esta estructura puede iterar 0 o más veces.
`do-while`		La entrada a la estructura no está condicionada. Sólo la salida está condicionada a que se verifique la expresión lógica; por lo tanto, este ciclo repetirá las acciones 1, 2 y 3 al menos una vez. Luego, mientras se verifique **`expresiónLógica`**, las seguirá repitiendo.
`repeat-until`		Esta estructura es idéntica a la anterior. Sólo cambia la forma de evaluar la **`expresiónLógica`**. En este caso, el ciclo iterará **hasta** que se cumpla dicha expresión. Este ciclo también itera al menos 1 vez.

`for`		Itera exactamente *n* veces con $n = v_n - v_0 + 1$. El ciclo asigna a una variable de control (en este caso **`i`**) valores consecutivos, comenzando desde v_0 hasta llegar a v_n inclusive.

Ejemplo 5.

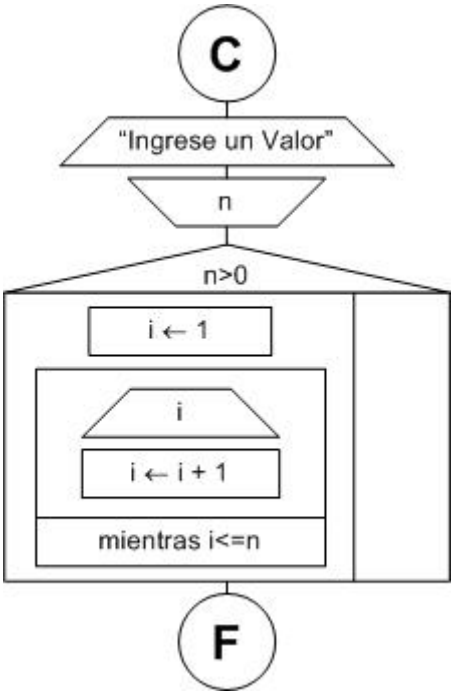
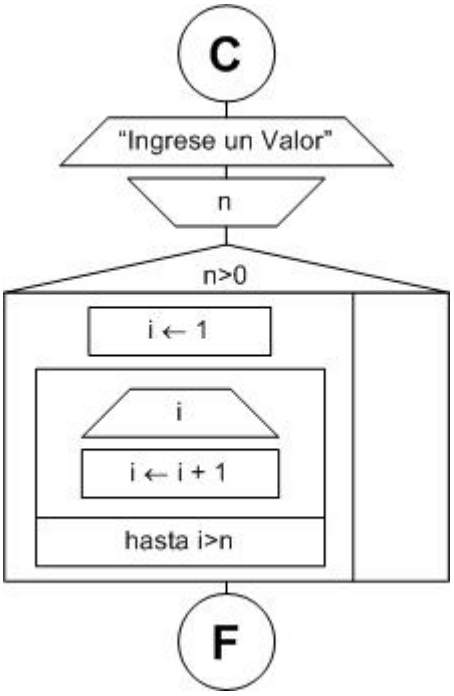
Desarrollar un programa que muestre los primeros *n* números naturales. El valor de *n* lo ingresa el usuario por teclado.

Solución 1 (con **`while`**)	Solución 2 (con **`for`**)
	

En el caso de la solución 1, debemos incrementar un contador (**`i`**) y, mientras se mantenga menor o igual que *n*, mostramos su valor y lo incrementamos. En la solución 2, el ciclo **`for`** se encarga de incrementar automáticamente el valor de la variable **`i`**.

Notemos que en ambos casos, si el usuario ingresa cero o un valor negativo, el algoritmo no emitirá ninguna salida.

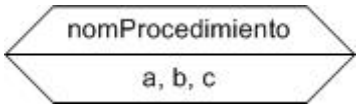
Para lograr el mismo efecto utilizando los ciclos **`do-while`** o **`repeat-until`**, tenemos que validar que el valor ingresado por el usuario sea mayor que cero, ya que el ingreso a estos ciclos no está condicionado.

con **`do-while`**	con **`repeat-until`**
	

Representación de Procedimientos y Funciones.

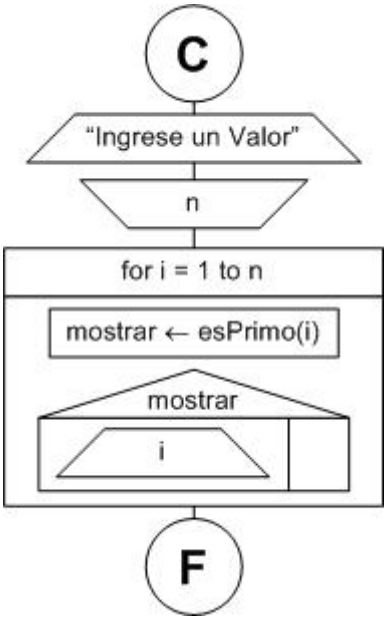
La base de la programación estructurada y del diseño **top-down** se fundamenta en la correcta aplicación de procedimientos y/o funciones que permitan descomponer un problema complejo en varios problemas más simples de resolver.

La invocación a procedimientos y funciones la representamos así:

	Símbolo	**Descripción**
Funciones		Invoca a la función **`nomFunción`** pasándole los argumentos **`x`** e **`y`**. Además, asigna en la variable **`v`** el valor que retorna la función.
Procedimientos		Invoca al procedimiento **`nomProcedimiento`** pasándole los argumentos **`a`**, **`b`** y **`c`**.

Ejemplo 6.

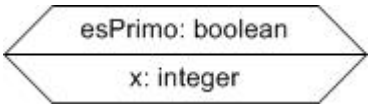
Desarrollar un algoritmo que imprima por pantalla los primeros *n* números primos, siendo *n* un valor que se ingresa por teclado.



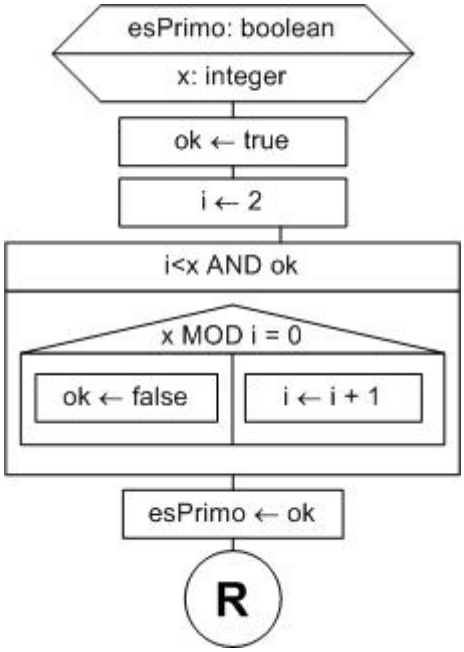
Para resolver este problema utilizamos la función **`esPrimo`**, que recibe un valor entero como parámetro y retorna **`true`** o **`false`** –lo que indica si se trata de un número primo o no–.

Según el nivel de detalle que se quiera proveer, el problema puede darse por terminado aquí, o bien, podemos continuar diagramando el algoritmo de la función **`esPrimo`**.

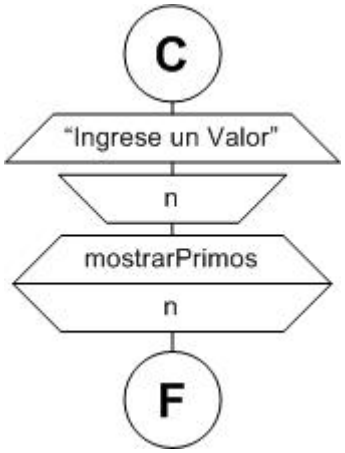
Para diagramar el algoritmo de una función utilizamos una cabecera en la que se detalla el nombre de la función (**`esPrimo`**), el tipo de dato del valor de retorno (**`boolean`**) y la lista de parámetros y sus tipos de datos (**`x: integer`**).



Ahora sí, veamos el desarrollo de la función **`esPrimo`**:

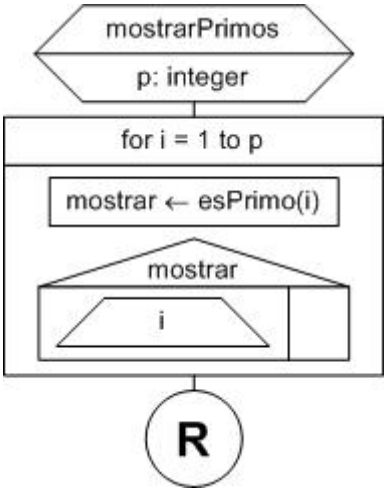


Ahora resolveremos el mismo problema pero utilizando, además, un procedimiento:



Este ejemplo es todavía más amplio. Leemos un valor e invocamos al procedimiento **`mostrarPrimos`**, quien se encargará de mostrar los primeros *n* números primos.

El desarrollo del procedimiento es el siguiente:



Notemos que tanto la función como el procedimiento finalizan con una **`R`** encerrada en un círculo, que representa el retorno (**`return`**) al programa que lo invocó.

Tabla de Variables y Parámetros.

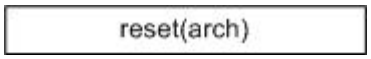
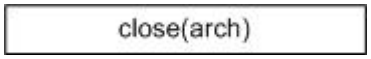
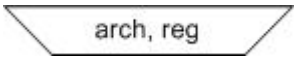
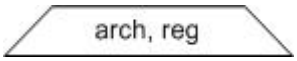
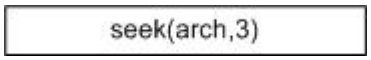
Dado que el diagrama no aporta información sobre las variables locales de cada procedimiento y función, muchas veces es conveniente adjuntar al diagrama una tabla que brinde un resumen de las variables y parámetros con los que trabaja cada procedimiento y/o función.

La siguiente tabla corresponde a la última versión del **ejemplo 6** resuelto con el procedimiento **`mostrarPrimos`** y la función **`esPrimo`**.

Procedimiento/Función	Variable	Parámetro	Tipo de Parámetro	Tipo de Dato
Programa Principal	n	-----	-----	**`integer`**
`mostrarPrimos`	-----	p	por valor	**`integer`**
	i	-----	-----	**`integer`**
`esPrimo`	-----	x	por valor	**`integer`**
	i	-----	-----	**`integer`**
	ok	-----	-----	**`boolean`**

Representación del Uso de Archivos.

La siguiente tabla muestra cómo representar las acciones relacionadas con el uso de archivos.

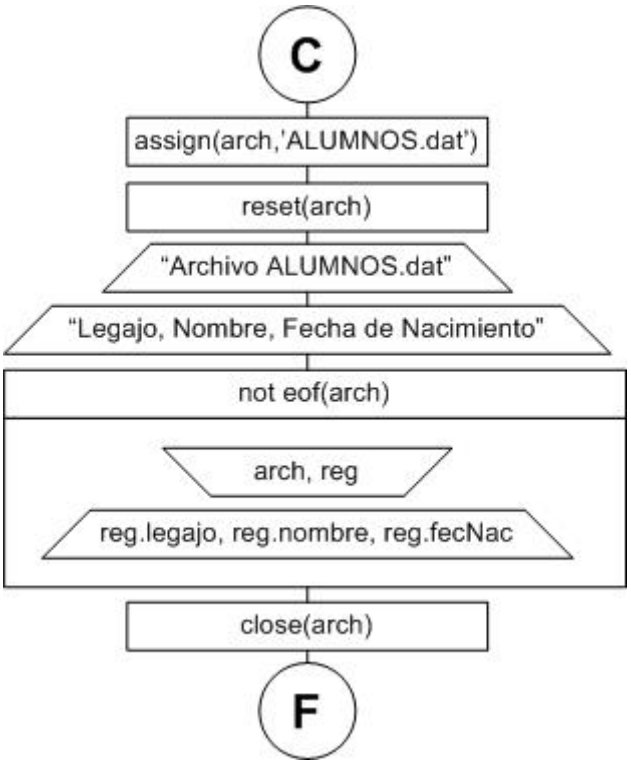
Acción	Símbolo	Descripción
Abrir un archivo		Relaciona el archivo físico "**`ALUMNOS.dat`**" con la variable de archivo **`arch`**.
Resetear		Mueve el puntero del archivo **`arch`** al byte número cero.
Cerrar el archivo		Cierra el archivo asociado con la variable **`arch`**.
Leer desde un archivo		Lee un registro del archivo asociado con **`arch`** y almacena los datos leídos en la variable **`reg`**.
Escribir en un archivo		Escribe en el archivo asociado con **`arch`** los datos contenidos en **`reg`**.
Posicionar		Posiciona el puntero en el registro número 3.

Ejemplo 7.

De acuerdo con el archivo **ALUMNOS.dat**, cuyo diseño se describe a continuación, desarrolle un algoritmo que muestre su contenido en pantalla.

```
type
RAlumno = record
     legajo: integer;
     nombre: string[15];
     fecNac: longint;
end;

FAlumnos = file of RAlumno;
```



Como podemos observar, las funciones para manejo de archivos (**assign**, **reset** y **close**) las recuadramos como lo hacemos con cualquier acción simple.

La única diferencia con lo estudiado hasta el momento es que en la lectura pasamos dos argumentos: **arch** (variable de archivo tipo **FAlumnos**) y **reg** (variable tipo **RAlumno**). Con esto representamos que la lectura será desde el archivo apuntado por **arch** y que los datos leídos quedarán almacenados en el registro **reg**.

La tabla de variables y parámetros para este diagrama es la siguiente:

Procedimiento/Función	Variable	Parámetro	Tipo de Parámetro	Tipo de Dato
Programa Principal	**arch**	-----	-----	**FAlumnos**
	reg	-----	-----	**RAlumno**

Ejemplo 8.

Dado el archivo **EMPLEADOS.dat**, cuyo diseño se detalla a continuación, desarrollar un algoritmo que permita modificar un "X%" el sueldo de un empleado, cuyo número de legajo se ingresa por teclado. Considere que el archivo está ordenado por el campo **legajo**.

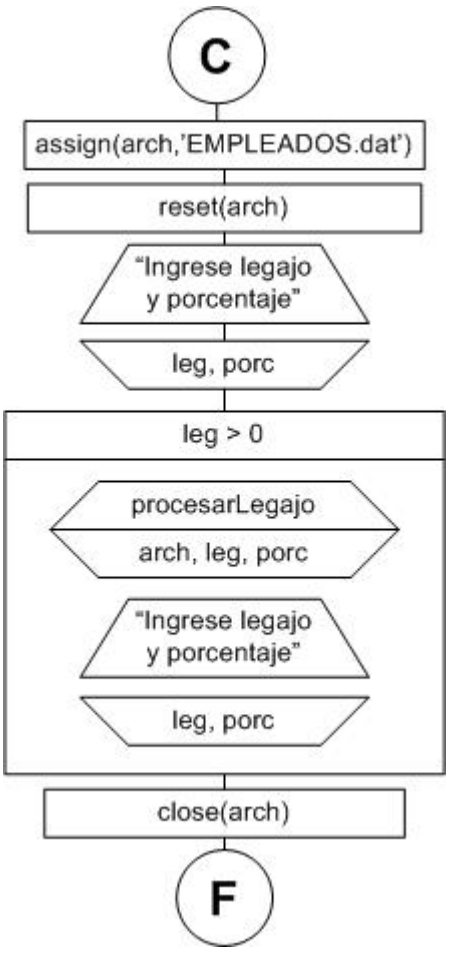
```
type
REmpleado = record
     legajo: integer;
     nombre: string[15];
     sueldo: real;
end;

FEmpleados = file of REmpleado;
```

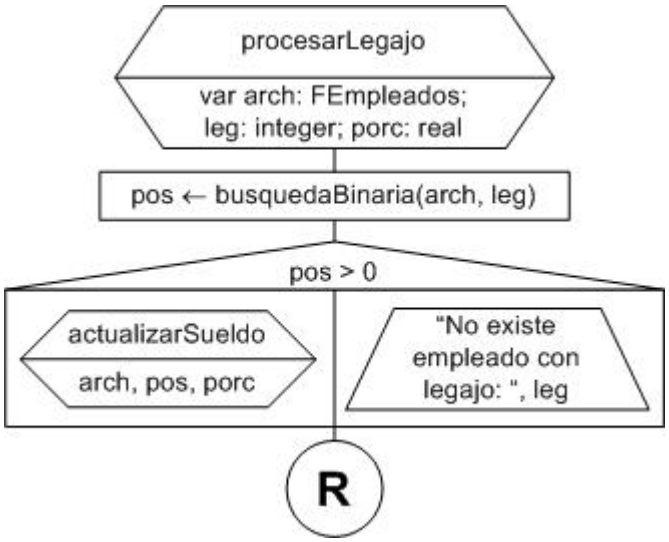
Análisis.

Si bien este problema es un poco más complejo que el anterior, podemos resolverlo fácilmente utilizando de manera adecuada procedimientos y funciones.

La idea es que el usuario ingrese el número de legajo y el porcentaje (positivo o negativo) que quiera aplicar al sueldo del empleado, cuyo legajo ingresó. Así será hasta que se ingrese un número de legajo negativo para finalizar el programa.

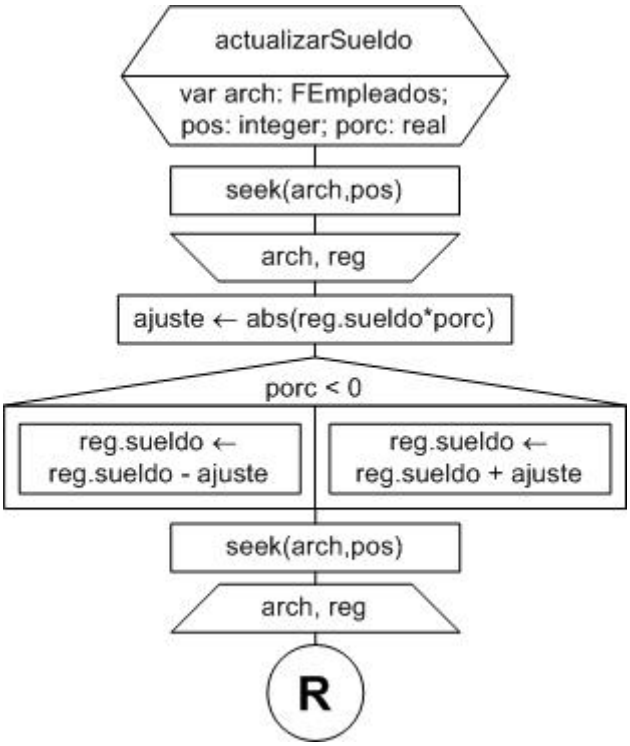


Como vemos, en el programa principal abrimos el archivo y manejamos la interacción con el usuario. El “problema” de actualizar el sueldo lo delegamos en el procedimiento **`procesarLegajo`**, cuyo desarrollo vemos a continuación.



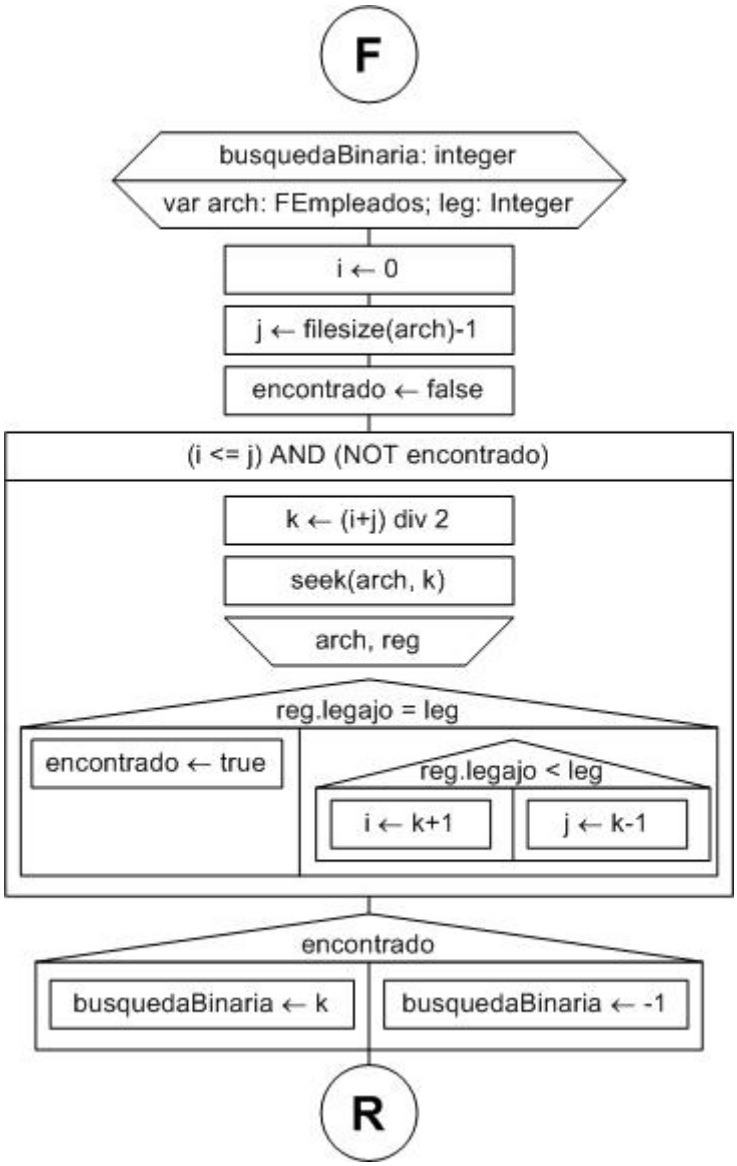
En **`procesarLegajo`** primero verificamos que el usuario haya ingresado un legajo correcto. Para esto (aprovechando que el archivo está ordenado por el campo **`legajo`**), realizamos una búsqueda binaria sobre el archivo. Esta búsqueda la lleva a cabo la función **`busquedaBinaria`**, que retorna el número de registro en el que se encuentra el empleado en cuestión, o un valor negativo si no existe ningún empleado con el legajo especificado. Si este último fuese el caso, entonces mostramos un mensaje de error indicando que el legajo ingresado es incorrecto.

Si el legajo existe, entonces debemos actualizar el valor del sueldo del empleado. Esto lo hacemos en el procedimiento **`actualizarSueldo`**.



Este procedimiento recibe la posición del registro cuyo campo **`legajo`** debemos actualizar. Para esto, posicionamos el puntero del archivo y leemos el registro que se ha de modificar. Calculamos el porcentaje de ajuste, es decir, si el usuario ingresó como porcentaje el valor 0.15, significa que quiere aumentar el sueldo un 15%; pero si ingresó un porcentaje -0.15, significa que quiere bajar el sueldo del empleado. Ahora, supongamos que el sueldo actual es $1000, entonces el ajuste (para arriba o para abajo) será 1000*0.15 = $150. Luego, dependiendo del signo del porcentaje, sumamos o restamos este valor. Volvemos a posicionar el puntero del archivo y grabamos el registro con el valor del sueldo modificado.

Por último, si se quiere además proveer el detalle de cómo se realiza la búsqueda binaria, se puede agregar el diagrama de la función.



Ahora veamos la tabla de variables y parámetros para este programa:

Procedimiento/Función	**Variable**	**Parámetro**	**Tipo de Parámetro**	**Tipo de Dato**
Programa Principal	**`arch`**	-----	-----	**`FEmpleados`**
	`leg`	-----	-----	**`integer`**
	`porc`	-----	-----	**`real`**
`procesarLegajo`	-----	**`arch`**	referencia	**`FEmpleados`**
	-----	**`leg`**	valor	**`integer`**

	-----	**porc**	valor	**real**
	pos	-----	-----	**integer**
actualizarSueldo	-----	**arch**	referencia	**FEmpleados**
	-----	**pos**	valor	**integer**
	-----	**porc**	valor	**real**
	reg	-----	-----	**REmpleado**
busquedaBinaria	-----	**arch**	referencia	**FEmpleados**
	-----	**leg**	valor	**integer**
	i	-----	-----	**integer**
	j	-----	-----	**integer**
	k	-----	-----	**integer**
	encontrado	-----	-----	**boolean**