# Taller de Programación

Guía para las clases

*Mgs. Lic. Marcos Prunello*

*14-09-2019*

# Índice general

# Capítulo 1

# Bienvenidos al Taller

La presente es una breve guía que resume los conceptos más importantes a desarrollar en el Taller de Programación de la Licenciatura en Estadística en la Facultad de Ciencias Económicas y Estadística, Universidad Nacional de Rosario. La misma irá siendo revisada a lo largo del cuatrimestre y no está exenta de presentar errores o expresar ideas que puedan ser mejoradas. ¡Esperamos que juntos podamos enriquecerla al dar nuestros primeros pasos en la programación!

## Horario y lugar de cursado

Lunes de 8:00 a 12:00 en el Laboratorio de Computación de la Escuela de Estadística.

## Consultas

A través del Foro de Consultas en el Campus Virtual o al finalizar las clases.

## Material de estudio

El material de estudio está compuesto por esta guía, prácticas y algunas presentaciones en diapositivas, que, junto con cualquier otro material que necesitemos, estarán disponibles en nuestro espacio en el Campus Virtual de la UNR.

Todo el material disponible (guía, diapositivas, prácticas, publicación online) fue creado utilizando el lenguaje de programación estadística `R` y las herramientas del entorno de desarrollo integrado `RStudio`.

## Evaluación

Tendremos tres Actividades Evaluativas (AE) cuyos detalles serán explicados oportunamente en el cuatrimestre. Las tres notas se promediarán para obtener una Nota Final con las siguientes ponderaciones: 20 % AE1, 40 % AE2 y 40 % AE3. Si la Nota Final es mayor o igual a ocho (8), se promueve la materia; si es es mayor o igual a seis (6) y menor a ocho (8), se alcanza la condición de regular; y si es menor a seis (6), el alumno es libre.

Habrá una única instancia opcional de recuperación al final del cuatrimestre de carácter integrador para aquellos que no alcanzaron la promoción, cuya nota no reemplaza a ninguna actividad evalulativa. Si la misma es mayor o igual a ocho (8) se alcanza la promoción y en caso contrario se conserva la condición ya obtenida.

## Campus virtual

Además de alojar todo el material del curso, utilizaremos este espacio para la entrega de trabajos y para realizar consultas, que esperamos puedan ser debatidas y respondidas entre los mismos estudiantes.

# Capítulo 2

# Introducción

## 2.1. Una breve reseña histórica sobre la computación

La palabra **computación** proviene del latín *computatio*, que deriva del verbo *computare*, cuyo significado es "enumerar cantidades". Computación, en este sentido, designa la acción y efecto de computar, realizar una cuenta, un cálculo matemático. De allí que antiguamente computación fuese un término usado para referirse a los cálculos realizados por una persona con un instrumento expresamente utilizado para tal fin (como el ábaco, por ejemplo) o sin él. En este sentido, la computación ha estado presente desde tiempos ancestrales.

A través de la historia, la computación progresó de manera relativamente lenta. En 1623, el alemán Wilhelm Schickard inventó la primera calculadora mecánica, capaz de realizar cómputos aritméticos sencillos. Este, y otros modelos posteriores, eran puramente mecánicos, sin motores ni otras fuentes de energía. El operador ingresaba números ubicando ruedas de metal en posiciones particulares y al girarlas otras partes de la máquina se movían y mostraban el resultado.

Durante la Revolución Industrial, el rápido crecimiento de la tecnología hizo posible considerar nuevas formas para realizar cálculos matemáticos, tomando provecho de que las máquinas de vapor podían proveer energía para hacer funcionar nuevos mecanismos. En este tiempo, se destacó el matemático británico Charles Babbage, quien diseñó dos tipos de máquinas calculadoras, una para crear tablas de funciones matemáticas (llamada *máquina diferencial*), y otra que concibió como de uso general, capaz de realizar distintas funciones de acuerdo a cómo fuese "programada" (*máquina analítica*). La operación de esta máquina era controlada por un patrón de perforaciones hechas sobre una tarjeta que la misma podía leer. Al cambiar el patrón de las perforaciones, se podía cambiar el comportamiento de la máquina para que resuelva diferentes tipos de cálculos.

Figura 2.1: Réplica de la máquina calculadora de Schickard.

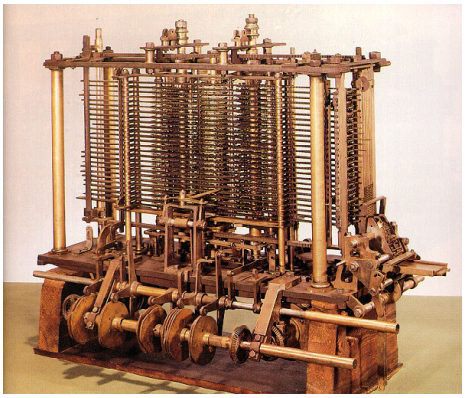

Figura 2.2: La máquina analítica de Babbage.

Si bien no llegó a concretar sus diseños en vida y fueron finalizados por otras personas, sentó bases importantes en la historia de la computación.

En 1890, Herman Hollerith utilizó tarjetas perforadas para automatizar la tabulación de datos para el censo de Estados Unidos, y con el fin de comercializar esta tecnología fundó una compañía que terminaría siendo la famosa International Business Machine (IBM). Sin embargo, la visión de Babbage de una computadora programable no se hizo realidad hasta los años 1940, cuando el advenimiento de la electrónica hizo posible superar a los dispositivos mecánicos existentes. El primer prototipo de una computadora electrónica fue armado por John Atanasoff y Clifford Barry en Iowa State College en 1939, que contaba con 300 tubos de vacíos, componentes electrónicos que pueden modificar una señal eléctrica mediante el control del movimiento de los electrones produciendo una respuesta.

La primera computadora electrónica a gran escala fue la ENIAC, *Electronic Numerical Integrator and Computer*, completada por Presper Eckert y John

Figura 2.3: Réplica de la máquina calculadora de Schickard.

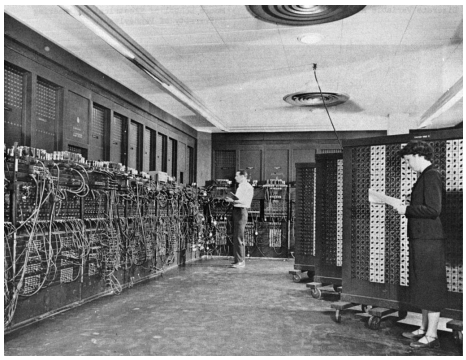

Figura 2.4: La ENIAC en Filadelfia, Pennsylvania.

Mauchly en Pennsylvania. Tenía más de 18000 tubos de vacío, ocupaba una sala de 9x15 metros y era controlada conectando ciertos cables en un panel, pero poder programarla resultó ser más difícil de lo que los inventores esperaban. Sin embargo, un gran avance se produjo en 1946, cuando John von Neumann en Princeton propuso que los programas, es decir, las instrucciones para que la máquina opere, y los datos necesarios podrían ser representados y guardados en una memoria interna.

Desde entonces, la computación ha evolucionado muy rápidamente, con la introducción de nuevos sistemas y conceptos. Algunos historiadores dividen al desarrollo de las computadoras modernas en cuatro generaciones:

- **Primera generación**: se trata de las computadoras electrónicas que usaban tubos de vacío para su circuito interno como la de Atanasoff.
- **Segunda generación**: nació a partir de 1947 con el desarrollo del *transistor*, un dispositivo electrónico semiconductor que entrega una señal de salida en respuesta a una señal de entrada, mucho más pequeño que los

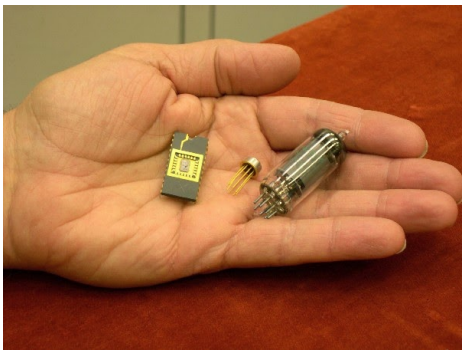

Figura 2.5: De derecha a izquierda: un tubo de vacío, un transistor y un chip.

tubos de vacío y que consumen menos energía eléctrica. Aún así, una computadora podía tener cientos de miles de transistores, ocupando mucho espacio.

- **Tercera generación**: se inició en 1959 con el desarrollo de un circuito integrado ("chip") que se trata de una pequeña placa de silicio sobre el cual se imprime un gran número de transistores conectados. La primera computadora de este estilo fue de IBM en 1960.
- **Cuarta generación**: comenzó en 1975 cuando los avances tecnológicos permitieron construir la unidad entera de procesamiento de una computadora sobre un único chip de silicio. Los procesadores que consisten de un único chip se llaman *microprocesadores* y son utilizados en la mayoría de las computadoras de hoy.

## 2.2. Software y hardware

La computadora en sí misma es sólo una parte de la historia. La máquina física que uno puede comprar y llevar al escritorio de casa es un ejemplo de **hardware**. Es tangible. Pero para que una computadora pueda cumplir con el propósito general de servir para una variada gama de tareas, debe ser **programada**. El acto de programar una computadora consiste en proveer un conjunto de instrucciones - un **programa** - que especifica todos los pasos necesarios para resolver un problema que se le asigna. Estos programas generalmente se conocen como **software**, y es la conjunción de ambos, hardware y software, la que le da vida a la computación.

A diferencia del hardware, el software es una entidad abstracta, intangible. Se trata de una secuencia de pasos simples y operaciones, especificadas en un lenguaje que el hardware puede interpretar. En nuestro **Taller de Programación** nos concentraremos en el diseño de la solución de un problema y cómo transmitírsela a la computadora para que la misma pueda ejecutarla.


Figura 2.6: Representación de la diferencia entre hardware y software.

# Capítulo 3

# Problemas, algoritmos y lenguajes de programación

Como seres humanos, tenemos incorporada intuitivamente la resolución de problemas cotidianos gracias a nuestra experiencia, y para intentar afrontar un inconveniente, solemos hacer un proceso rápido de selección e intentamos buscar la opción más favorable.

Un **algoritmo** es una estrategia consistente de un conjunto ordenado de pasos que nos lleva a la solución de un problema o alcance de un objetivo. Un algoritmo cumple con las siguientes características:

- **Está expresado de manera clara y precisa**: el lector debe poder entender sin ambigüedades cuáles son los pasos involucrados.
- **Es efectivo**: en el sentido de que sea factible llevar a cabo dichos pasos.
- **Es finito**: debe arrojar una respuesta o brindar la solución al problema en una cantidad finita de tiempo, es decir, termina luego de un número acotado de pasos.

Los algoritmos, así como los problemas que intentan resolver, varían ampliamente en complejidad. Algunos problemas son tan simples que inmediatamente se nos ocurre un algoritmo apropiado para su resolución. Para problemas complejos, en cambio, desarrollar un algoritmo requiere más tiempo y razonamiento, e incluso se pueden originar distintos algoritmos para solucionar un mismo problema.

La resolución computacional de un problema consiste de dos etapas básicas:

- **Diseño algorítmico**: desarrollar un algoritmo, o elección de uno existente, que resuelva el problema.

- **Codificación**: expresar un algoritmo en un lenguaje que la computadora pueda interpretar.

Al aprender sobre programación, comenzamos enfrentándonos a problemas simples para los cuales la primera etapa parece sencilla, mientras que la codificación se torna dificultosa ya que hay que aprender las reglas del lenguaje de programación. Sin embargo, mientras que con práctica rápidamente podemos ganar facilidad para la escritura de códigos, el diseño algorítmico se torna cada vez más desafiante al encarar problemas más complejos. Es por eso que haremos incapié en el planteo y desarrollo de algoritmos como una etapa fundamental en la programación.

## 3.1. El diseño algorítmico

Frente a cada problema, el primer paso es idear una algoritmo para su solución y expresarlo por escrito, por ejemplo, en español, pero adaptando el lenguaje humano a formas lógicas que se acerquen a las tareas que puede realizar una computadora. En programación, el lenguaje artificial e informal que usan los desarrolladores en la confección de algoritmos recibe el nombre de **pseudocódigo**. Es la herramienta que utilizamos para describir los algoritmos mezclando el lenguaje común con instrucciones de programación. No es en sí mismo un lenguaje de programación, es decir, la computadora no es capaz de entenderlo, sino que el objetivo del mismo es que el programador se centre en la solución lógica y luego lo utilice como guía al escribir el programa.

El pseudocódigo, como cualquier otro lenguaje, está compuesto por:

- Un **léxico**: conjunto de palabras o frases válidas para escribir las instrucciones
- Una **sintaxis**: reglas que establecen cómo se pueden combinar las distintas partes
- Una **semántica**: significado que se les dá a las palabras o frases

Además el pseudocódigo, y toda codificación en lenguajes de programación, sigue una **estructura secuencial**: define una acción o instrucción que sigue a otra en secuencia. Esta estructura puede representarse de la siguiente forma:

```
ALGORITMO: "Ejemplo"
COMENZAR
    Acción 1
    Acción 2
    ...
    Acción N
FIN
```

Se comienza con un título que describa el problema que el algoritmo resuelve, seguido por la palabra `COMENZAR`. Luego se detallan las acciones o instrucciones a seguir y se concluye con la palabra `FIN`. Por ejemplo, si nuestro problema es poner en marcha un auto, el algoritmo para resolverlo puede ser expresado mediante el siguiente pseudocódigo:

```
ALGORITMO: "Arrancar el auto"
COMENZAR
    INSERTAR la llave de contacto
    UBICAR el cambio en punto muerto
    GIRAR la llave hasta la posición de arranque
    SI el motor arranca
        ENTONCES
            DEJAR la llave en posición "encendido"
        SI NO
            LLAMAR al mecánico
    FINSI
FIN
```

Es importante destacar la presencia de sangrado (*indentación*), instrucciones, verbos y estructuras de control en el ejemplo anterior.

## 3.2. Codificación: creación y edición de programas

El algoritmo anterior está presentado en pseudocódigo utilizando el lenguaje español, una opción razonable para compartir esta estrategia entre personas que se comuniquen con este idioma. Claramente, si queremos presentarle nuestro algoritmo a alguien que sólo habla francés, el español ya no sería una buena elección, y mucho menos si queremos presentarle el algoritmo a una computadora.

Para que una computadora pueda entender nuestro algoritmo, debemos traducirlo en un **lenguaje de programación**: un idioma artificial diseñado para expresar cómputos que puedan ser llevados a cabo por equipos electrónicos, es decir es un medio de comunicación entre el humano y la máquina. Ejemplos de lenguajes de programación son Fortran, BASIC, C++ o Java. En este curso, aprenderemos a utilizar un lenguaje incorporado en el software estadístico SAS: **IML** (*Interactive Matrix Language*).

Cada una de las acciones que componen al algoritmo son codificadas con una o varias **instrucciones** o **sentencias**, expresadas en el lenguaje de programación elegido, y el conjunto de todas ellas constituye un **programa**. El programa se guarda en un **archivo** con un nombre generalmente dividido en dos partes por un punto, por ejemplo: `miPrimerPrograma.sas`. La primera parte es la **raíz**


Figura 3.1: Distintos lenguajes de programación y sus logos.

del nombre con la cual podemos describir el contenido del archivo. La segunda parte es indicativa del uso del archivo, por ejemplo, `.sas` indica que contiene un programa escrito en el lenguaje de SAS. El proceso general de ingresar o modificar el contenido de un archivo se denomina **edición**.

Existen distintos tipos de lenguajes de programación que cumplen la función de intermediarios entre el desarrollador y el hardware. Simplificando esta gran variedad, podemos decir que hay dos grupos generales. Por un lado, se encuentran los lenguajes más próximos a la arquitectura del hardware, denominados **lenguajes de bajo nivel**, que son más rígidos y complicados de entender para nosotros. Por otro lado, están aquellos más cercanos a los programadores y usuarios, denominados **lenguajes de alto nivel**, diseñados para que sea fácil para los humanos expresar los algoritmos sin necesidad de entender en detalle cómo hace exactamente el hardware para ejecutarlos. El lenguaje que utilizaremos en este taller, IML, es de alto nivel.

Para que un programa escrito en un lenguaje de alto nivel pueda ser ejecutado, se necesita de **compiladores** o **intérpretes** que básicamente lo traducen al lenguaje de bajo nivel que es apropiado para el hardware que se dispone. Un *compilador* toma un programa escrito en lenguaje de alto nivel y produce como resultado un programa escrito en otro lenguaje, que luego puede ser ejecutado, mientras que un *intérprete* traduce y ejecuta simultáneamente.

## 3.3. Errores de programación y depuración

Tal como el lenguaje humano, los lenguajes de programación tienen su propio vocabulario y su propia sintaxis, que es el conjunto de reglas gramaticales que establecen cómo se pueden combinar las distintas partes del lenguaje. Estas reglas sintácticas determinan que ciertas sentencias están correctamente construidas mientras otras no.

Cuando compilamos un programa, el compilador primero chequea si el mismo es sintácticamente correcto. Si hemos violado alguna regla, mostrará un mensaje de error y debemos editar nuestro programa para corregirlo.

Figura 3.2: Encontrando un "bug."[en] un programa.

Aunque los errores de sintaxis pueden ser frustrantes, los errores lógicos suelen ser los más problemáticos. Por ejemplo, el programa puede ser compilado sin errores pero arroja resultados incorrectos o ningún resultado. En estos casos hay que revisar el programa para encontrar algún error en la lógica del mismo. Este tipo de errores suelen llamarse **bugs** y la corrección de los mismos **debugging** (depuración).

En nuestro recorrido por el taller, seguramente nos enfrentaremos a varias situaciones en las cuales nos parece que el algoritmo es correcto, para luego descubrir que falla porque nos hemos olvidado de considerar alguna situación particular. Si bien esto puede resultar desalentador, forma parte de la práctica cotidiana de todos los programadores.

## 3.4. Resumiendo

A la hora de resolver un problema computacional, seguiremos los siguientes pasos:

- Analizar el problema que vamos a resolver.
- Imaginar una solución (algoritmo).
- Traducir la solución a pseudocódigo.
- Implementar en un lenguaje de programación todo lo analizado.
- Compilar o correr el programa.
- Realizar pruebas de ejecución.
- Corregir los errores que haya.

# Capítulo 4

# Primeros pasos en SAS/IML

## 4.1. Qué es SAS

- **SAS** es un paquete de programas orientados al análisis estadístico desarrollado por **SAS Institute** a finales de los años sesenta.
- Este instituto se inició a finales de los años sesenta como un proyecto en la Universidad de Carolina del Norte para crear un *sistema de análisis estadístico* (**S**tatistical **A**nalysis **S**ystem) originalmente utilizado por los departamentos de Agricultura de algunas universidades.
- En 1976 se convirtió en una compañía y privada y desde entonces tomó relevancia y gran prestigio en la comunidad estadística internacional.

## 4.2. Cómo instalar SAS

- SAS no es un software libre y como tal para poder instalarlo y hacer uso del mismo se debe adquirir una licencia.
- Sin embargo, disponen de una versión gratuita pensada para estudiantes universitarios llamada SAS University Edition.
- El link anterior lleva a la página oficial que detalla todos los pasos para poder instalar y usar esta versión gratuita.
- Como alternativa, el siguiente video también provee instrucciones detalladas para Windows.

```
## PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed,
```

## 4.3. Qué es SAS/IML

- IML son las siglas de *Interactive Matrix Language*.
- Es decir, IML es un lenguaje de programación que hace foco en la utilización y manejo de vectores y matrices, permiendo hacer con ellos cómputos de alto nivel.
- Permite interactuar con otros procedimientos de SAS, acceder a archivos, y crear gráficos entre otras cosas.
- Incluye un variado conjunto de funciones y operadores para asistir en la programación.
- Leer la guía sobre IML disponible en el Campus Virtual.

## 4.4. Cómo usar IML

- El procedimiento de SAS que implementa este lenguaje se llama de la misma forma, por lo cual todo el código de nuestros programas estará siempre encerrado en bloques del estilo:

```
proc iml;
    ...
    líneas de código
    ...
quit;
```

# Capítulo 5

# Elementos para la creación de algoritmos

## 5.1. Procesador, ambiente y acciones

Hemos definido a un algoritmo como una lista de instrucciones que serán traducidas con un lenguaje de programación para ser ejecutadas por computadora. En este sentido, el concepto físico de la máquina o computadora hacer referencia a la necesidad de contar con un **procesador** para resolver el problema. Se define como **procesador** a todo agente capaz de entender las órdenes del programa y ejecutarlas.

Para cumplir con esto, el procesador empleará ciertos recursos que forman parte del sistema en el cual se ejecuta el programa. Por ejemplo, utilizará dispositivos de almacenamiento para guardar datos o dispositivos de salida para comunicar el resultado. Todos los elementos disponibles para ser utilizados por el procesador constituyen su **entorno** o **ambiente**.

Cada una de las instrucciones que componen el algoritmo modifican el entorno de alguna manera y se denominan **acciones**.

### 5.1.1. Ejemplo 1

- **Problema**: preparar una tortilla de 6 huevos.
- **Entorno**: una mesa, una hornalla, una sartén, un plato, un tenedor, aceite, una fuente con huevos, un tarro de basura.
- **Procesador**: un adulto.
- **Acciones comprensibles por el procesador**: agarrar un huevo, romper un huevo en un plato, batir huevos, poner aceite, poner en la sartén, poner al fuego, retirar del fuego, tirar las cáscaras, encender el fuego.

¿Cuál es un algoritmo adecuado para solucionar este problema? Podría ser:

```
ALGORITMO: "Preparar una tortilla de 6 huevos"
COMENZAR
    ROMPER seis huevos en un plato
    TIRAR las cáscaras en el tacho
    BATIR los huevos
    CALENTAR aceite en la sartén
    PONER el contenido del plato en la sartén
    ESPERAR diez minutos
    RETIRAR la tortilla del fuego
    APAGAR el fuego
FIN
```

### 5.1.2. Ejemplo 2

- **Problema**: calcular el factorial del número 5.
- **Entorno**: se dispone de una calculadora común.
- **Procesador**: un adulto.
- **Acciones comprensibles por el procesador**: pulsar teclas de la calculadora.

¿Cuál es un algoritmo adecuado para solucionar este problema? Podría ser:

```
ALGORITMO: "Calcular 5!"
COMENZAR
    PULSAR [ON]
    PULSAR [1]
    PULSAR [X]
    PULSAR [2]
    PULSAR [X]
    PULSAR [3]
    PULSAR [X]
    PULSAR [4]
    PULSAR [X]
    PULSAR [5]
    PULSAR [=]
    MOSTRAR la pantalla
FIN
```

### 5.1.3. Acciones primitivas o compuestas

Las acciones del algoritmo pueden clasificarse en función de su complejidad:

- **Primitivas**: acción sencilla directamente realizable por el procesador.
- **Compuestas**: compuestas por una sucesión de acciones primitivas.

La descripción de un algoritmo en términos de acciones compuestas puede facilitar su comprensión, pero al desarrollar el programa será necesario descomponerlas en acciones primitivas que son las que realmente ejecuta el procesador. Por ejemplo, la acción compuesta en el ejemplo de la tortilla de "romper seis huevos en un plato" puede descomponerse en acciones más simples:

```
REPETIR 6 VECES
    TOMAR un huevo
    GOLPEAR el huevo para generar una fractura en la cáscara
    EJERCER presión sobre la cáscara
    DERRAMAR la clara y la yema sobre el plato
```

## 5.2. Objetos, constantes y variables

El procesador debe manipular distintas piezas de información que llamamos **objetos** y que componen al entorno. Según los valores que pueden representar, los objetos pueden ser de tipo:

- **Numérico**: Representan valores escalares de forma numérica y permiten realizar operaciones aritméticas comunes. Ejemplo: 230, 2.
- **Caracter**: Representan texto, no es posible hacer operaciones matemáticas con ellos y van entre comillas. Ejemplo: "hola", "chau123"
- **Lógicos**: Sólo pueden tener dos valores (*verdadero* o *falso*), ya que representan el resultado de la comparación entre otros objetos.

De manera general, al nombre de un objeto se le dice **identificador**, el cual es una secuencia de caracteres alfanuméricos que sirve para identificarlo a lo largo del algoritmo. Nombrar los objetos hace posible referirse a los mismos, lo cual es esencial para cualquier tipo de procesamiento simbólico.

Ciertos objetos almacenan temporalmente un valor durante la ejecución de un proceso y su contenido puede cambiar mientras corre el programa. Este tipo de objetos recibien el nombre de **variables**. Por ejemplo, en un programa creado para un comercio puede existir un objeto llamado *stock* (identificador) de tipo numérico que reprsenta la cantidad de artículos disponibles y cuyo valor se modifica cada vez que se registra una nueva venta. Podemos pensar a una variable como una caja que contiene una etiqueta con su identificador y un valor, por ejemplo, una simple letra (como "k", tipo caracter) o un valor numérico (como "2019", tipo numérico).

Si un objeto tiene siempre necesariamente un valor fijo, se dice que es una **constante**. El valor es siempre el mismo para cualquier ejecución del programa,

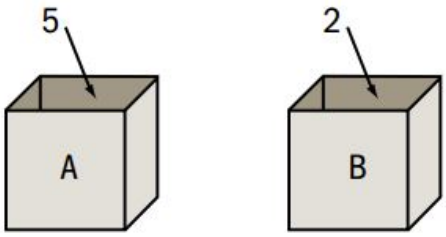


Figura 5.1: Las variables A y B contienen a los valores 5 y 2 respectivamente.

es decir, no puede cambiar de una ejecución a otra. Son ejemplos de constantes el número de meses del año, las constantes matemáticas tales como el número , los factores de conversión de unidades de medida, etc. Las constantes pueden usarse literalmente, es decir, usando explícitamente el valor, o también a través de un objeto que asocie un identificador al valor constante (por ejemplo, asociar el nombre "pi" a la constante 3.14159265).

### 5.2.1. Declaración de variables y constantes

Al expresar nuestros algoritmos en pseudocódigo tomaremos la costumbre de declarar al inicio del mismo las variables y constantes necesarias para resolver el problema, explicitando su identificador y determinando el tipo de valor que guarda. Muchos lenguajes de programación utilizan esta declaración para reservar en la memoria de la computadora un espacio para almacenar la información correspondiente de manera adecuada.

**En IML**. IML es un lenguaje dinámico que no requiere la declaración previa de las variables que serán utilizadas, sino que estas pueden definirse dinámicamente a lo largo del programa. Por eso, incluiremos declaración de variables y constantes sólo en los algoritmos.

### 5.2.2. Acción de asignación

Para hacer que una variable guarde un determinado valor se recurre a una **acción de asignación**. Mediante asignaciones podemos dar valores iniciales a las variables, modificar el valor que tenían hasta el momento o guardar en ellas resultados intermedios o finales del algoritmo.

En pseudocódigo expresaremos a la asignación como se muestra en estos ejemplos:

```
lugarNacimiento <- "Bombal"
pesoNacimiento <- 3.423
```

Si intervienen variables en la expresión a la derecha de una acción de asignación, se usará el valor que tenga la variable en ese momento. Por ejemplo, la siguiente secuencia de acciones en un algoritmo:

```
meses <- 2
dias <- meses
meses <- 7
saldo <- meses
```

resultará en que las variables `meses`, `dias` y `saldo` tengan almacenados los valores 7, 2 y 7 respectivamente. Un caso particular se da cuando a una variable se le asigna el valor de una operación de la que forma parte la misma variable. Por ejemplo:

```
dias <- dias + 30
```

**En IML**. La asignación se realiza a través del símbolo =. El código para el ejemplo anterior es:

```
proc iml;
    meses = 2;
    dias = meses;
    meses = 7;
    saldo = meses;
    print meses dias saldo;
quit;
```

## 5.3. Operadores

El desarrollo de un algoritmo involucra la necesidad de efectuar operaciones de distinto tipo: suma, resta, concatenación, procesos lógicos, etc. Los elementos que describen el tipo de operación a realizar entre dos objetos se denominan **operadores**.

### 5.3.1. Operadores aritméticos

Permiten realizar operaciones matemáticas con objetos de tipo numérico.

Los operadores aritméticos actúan con un orden de prioridad establecido, tal como estamos acostumbrados en matemática. Las expresiones entre paréntesis se evalúan primero. Si hay paréntesis anidados se evalúan desde adentro hacia afuera. Dentro de una misma expresión, los operadores se evalúan en este orden:

Cuadro 5.1: Signos para operadores aritméticos que se pueden utilizar en el pseudocódigo.

| Signo | Significado |
|---|---|
| + | Suma |
| - | Resta |
| * | Multiplicación |
| / | División |
| ** | Potenciación |
| MOD | Resto de la división entera |

1. Potenciación
2. Multiplicación, división, módulo
3. Suma, resta

Si la expresión presenta operadores con igual nivel de prioridad, se evalúan de izquierda a derecha. Ejemplos:

```
4 + 2 * 4 = 12
23 * 2 / 5 = 9.2
3 + 5 * (10 - (2 + 4)) = 23
2.1 * 1.5 + 12.3 = 15.45
2.1 * (1.5 + 12.3) = 28.98
1 MOD 4 = 1
8 * (7 - 6 + 5) MOD (1 + 8 / 2) - 1 = ...
```

**En IML**. Se utilizan los mismos símbolos, excepto para el operador módulo, cuya sintaxis es, por ejemplo, `mod(7, 4)` si se quiere obtener el resto de la división de 7 por 4.

### 5.3.2. Operadores relacionales

Sirven para comparar dos valores de cualquier tipo y dan como resultado un valor lógico: *verdadero* ("V") o *falso* ("F").

Ejemplos:

**En IML**. Se usan estos mismos símbolos.

### 5.3.3. Operadores lógicos

Mientras que los operadores relacionales comparan cualquier tipo de valores, los operadores lógicos sólo toman operandos de tipo lógico y producen también un resultado lógico. Los operadores lógicos que utilizaremos son **Y**, **O** y **NO**.

Cuadro 5.2: Signos para operadores lógicos que se pueden utilizar en el pseudocódigo.

| Signo | Significado |
|---|---|
| > | Mayor que |
| < | Menor que |
| >= | Mayor o igual que |
| <= | Menor o igual que |
| = | Igual a |
| ^= | Distinto a |

Cuadro 5.3: Ejemplos de la utilización de operadores relacionales.

| Operación relacional | Resultado |
|---|---|
| 5 >3 | verdadero |
| 5 <= 3 | falso |
| 3 * 4 = 10 + 2 | verdadero |
| 3 * 4 ^= 15 - 4 | verdadero |

- El operador lógico **Y** devuelve un valor **verdadero** sólo si son verdaderas las dos expresiones que vincula. Ejemplo: (3 > 2) Y (3 > 5) V Y F F.
- El operador lógico **O** devuelve un valor **verdadero** si al menos una de las dos expresiones que vincula es verdadera. Ejemplo: (3 > 2) O (3 > 5) V O F V.
- El operador lógico **NO**, niega un valor lógico, es decir, devuelve el opuesto. Ejemplo: NO (3 > 2) NO V F.

Las **tablas de la verdad** se utilizan para mostrar el resultado de los distintos tipos de operaciones lógicas:

Ejemplo: expresar con símbolos la expresión “x no es igual a 2 ni a 3”.

Expresiones correctas:

```
x ^= 2 Y x ^= 3
NO(x = 2 O x = 3)
```

Expresión incorrecta:

```
x ^= 2 O x ^= 3
```

Las Leyes de Morgan nos ayudan a trabajar en este tipo de casos:

- **NO (Valor1 O Valor2)** *es equivalente a* **NO Valor1 Y NO Valor2**
- **NO (Valor1 Y Valor2)** *es equivalente a* **NO Valor1 O NO Valor2**

Cuadro 5.4: Tabla de la verdad.

| Valor1 | Operador | Valor2 | Resultado |
|---|---|---|---|
| F | Y | F | F |
| F | Y | V | F |
| V | Y | F | F |
| V | Y | V | V |
| F | O | F | F |
| F | O | V | V |
| V | O | F | V |
| V | O | V | V |
| | NO | F | V |
| | NO | V | F |

Otra consideración es que expresiones que tienen sentido en matemática, pueden no tenerlo en programación y deben ser re estructuradas en las formas antes vistas. Por ejemplo, si queremos evaluar si es `V` o `F`la expresión `0 < x < 10`, debemos evaluar `0 < x Y x < 10`.

**En IML**. El operador Y se escribe como "&", el operador O es "|" y el operador NO es "~".

## 5.4. Entrada y salida de información

En la resolución de problemas puede ser necesario requerir que un usuario provea información, y registrar la misma como un valor que debe ser asignado a una variable. Para esto podemos utilizar la acción `LEER` dentro del algoritmo. Cuando deseamos mostrar un resultado en un mensaje empleamos la acción `ESCRIBIR`. Las palabras o frases literales que se desean mostrar en el mensaje deben estar encerradas entre comillas porque son cadenas de texto, mientras que si se desea mostrar el valor de una variable se debe escribir su identificador sin comillas.

**En IML**. La acción `LEER` la ejecutaremos mediante la asignación directa de un valor a una variable a través del símbolo "=". La acción `ESCRIBIR` será ejecutada a través de la sentencia `print` para obtener un resultado en la ventana de Output.

## 5.5. Empleo de comentarios en el algoritmo

A medida que los algoritmos se hacen más complejos, suele ser necesario añadir en el mismo ciertos comentarios que no son acciones que el procesador debe ejecutar, sino que cumplen con la función de explicar alguna característica del

algoritmo a otra persona que lo esté leyendo (o a uno mismo si lo leemos nuevamente en el futuro).

Cada vez que deseemos incluir un comentario, lo haremos en una línea que comience con el símbolo "\\".

## 5.6. Ejemplo

Los conceptos enunciados anteriormente pueden verse ejemplificados en el siguiente algoritmo para calcular el área de un círculo

```
ALGORITMO: "Calcular área de un círculo"
COMENZAR
    CONSTANTE numérica pi
    VARIABLE numérica radio, area
    pi <- 3.1416
    LEER radio
    area <- pi * radio ** 2
    ESCRIBIR "El área del círculo es " area
FIN
```

**En IML**:

```
/* PROGRAMA: Calcular área de un círculo */
proc iml;
    pi = 3.14159265358979;
    radio = 5;
    area = pi * radio ** 2;
    print "El área del círculo es " area;
quit;
```

# Capítulo 6

# Estructuras de Control

Como mencionamos anteriormente, un algoritmo está compuesto por una sucesión ordenada de comandos que se ejecutan uno detrás de otro. Sin embargo, con frecuencia es necesario recurrir a comandos especiales que alteran o controlan el flujo de las acciones. Por lo tanto, decimos que existen distintas **estructuras de control** que organizan a los algoritmos y que pueden clasificarse en **secuenciales**, **condicionales** e **iterativas**.

## 6.1. Estructuras de control secuenciales

Las estructuras secuenciales están compuestas por un número definido de acciones que se ubican en un orden específico y se suceden una tras otra. Los ejemplos que hemos discutido anteriormente están conformados por este tipo de estructura.

## 6.2. Estructuras de control condicionales

El curso de acción depende del resultado de la comparación una variable con un valor, que puede ser una constante u otra variable. Existen tres tipos de estructuras condicionales: **simples**, **dobles** y **múltiples**.

### 6.2.1. Estructuras condicionales simples

Representan una toma de decisión y se describen con la siguiente sintaxis:

```
SI <condición> ENTONCES
```

```
    Acción/es
FIN SI
```

La palabra `SI` indica el comando de comparación, `<condición>` indica la condición a evaluar y `Acción/es` son las instrucciones que se realizarán sólo si se cumple la condición. Si la condición no se verifica, no se ejecuta ninguna acción y el algoritmo sigue su estructura secuencial a continuación del `FIN SI`.

En el siguiente ejemplo utilizaremos esta estructura para determinar si una persona es mayor de edad o no.

```
ALGORITMO: "Determinar mayoría de edad"
COMENZAR
    VARIABLE numérica edad
    LEER edad
    SI edad >= 18 ENTONCES
        ESCRIBIR "Es mayor de edad"
    FIN SI
FIN
```

**En IML**. Usamos la estructura `if ... then ...;`:

```
/* PROGRAMA: Determinar mayoría de edad */
proc iml;
    edad = 19;
    if edad >= 18 then print "Es mayor de edad";
quit;
```

Si necesitamos ejecutar más de una acción al cumplirse la condición evaluada, debemos encerrarlas dentro de un bloque `do`, que siempre termina con un `end`:

```
/* PROGRAMA: Determinar mayoría de edad */
proc iml;
    edad = 19;
    if edad >= 18 then do;
        print "Es mayor de edad";
        print "Su edad es " edad " años";
    end;
quit;
```

### 6.2.2. Estructuras condicionales dobles

Este tipo de estructura añade una acción a ejecutarse en el caso de que la condición evaluada no se verifique. La sintaxis es:

```
SI <condición>
    ENTONCES
        Acción/es
    SI NO
        Acción/es
FIN SI
```

La palabra `ENTONCES` antecede a las acciones que se realizan si se cumple la condición y `SI NO` a las que se realizan si no se verifica la misma. Podemos ampliar el ejemplo anterior para que el algorimo indique también si la persona es menor de edad:

```
ALGORITMO: "Determinar mayoría de edad"
COMENZAR
    VARIABLE numérica edad
    LEER edad
    SI edad >= 18
        ENTONCES
            ESCRIBIR "Es mayor de edad"
        SI NO
            ESCRIBIR "Es menor de edad"
    FIN SI
FIN
```

**En IML**.

```
proc iml;
    edad = 17;
    if edad >= 18 then print "Es mayor de edad";
    else print "Es menor de edad";
quit;
```

Nuevamente, si necesitamos ejecutar múltiples acciones debemos encerrarlas dentro de un bloque `do`:

```
proc iml;
    edad = 19;
    if edad >= 18 then do;
        print "Es mayor de edad";
        print "Su edad es " edad " años";
    end;
    else do;
        print "Es menor de edad";
        print "Requerir ingreso acompañado";
    end;
quit;
```

### 6.2.3. Estructuras condicionales múltiples o anidadas

Permiten combinar varias estructuras condicionales para establecer controles más complejos sobre el flujo de las acciones, representando una toma de decisión múltiple. Podemos ejemplificar la sintaxis de la siguiente forma:

```
SI <condición>
    ENTONCES
        Acción/es
    SI NO
        SI <condición>
            ENTONCES
                Acción/es
            SI NO
                Acción/es
        FIN SI
FIN SI
```

Imaginemos que el ejemplo anterior de la mayoría de edad se da en el contexto de la entrada a una exhibición, donde los mayores de edad pueden ingresar pero los menores sólo pueden hacerlo si tienen la autorización de un adulto a cargo. Podemos plantear el siguiente algoritmo para determinar si una persona puede ingresar o no:

```
ALGORITMO: "Determinar ingreso"
COMENZAR
    VARIABLE numérica edad
    VARIABLE caracter autorizacion
    LEER edad
    SI edad >= 18
        ENTONCES
            ESCRIBIR "Permitir ingreso"
        SI NO
            LEER autorizacion
            SI autorizacion = "OK"
                ENTONCES
                    ESCRIBIR "Permitir ingreso"
                SI NO
                    ESCRIBIR "Denegar ingreso"
            FIN SI
    FIN SI
FIN

/* PROGRAMA: Determinar ingreso */
proc iml;
```

```
    edad = 16;
    autorizacion = "OK";
    if edad >= 18 then print "Permitir ingreso";
    else do;
        if autorizacion = "OK" then print "Permitir ingreso";
        else print "Denegar ingreso";
    end;
quit;
```

## 6.3. Estructuras de control iterativas

Las estructuras de control iterativas son útiles cuando la solución de un problema requiere que se ejecute repetidamente un determinado conjunto de acciones. El número de veces que se debe repetir dicha secuencia de acciones puede ser fijo o variable dependiendo de algún dato en el algoritmo.

### 6.3.1. Estructuras de control iterativas con un número fijo de iteraciones

Se aplican cuando se conoce de antemano el número exacto de veces que se debe repetir una secuencia de acciones. Por ejemplo, si deseamos mostrar en pantalla la tabla de multiplicar del número 8 completa podríamos hacer:

```
ALGORIMO: "Mostrar tabla del 8"
COMENZAR
    VARIABLE numérica resultado
    resultado <- 8 * 1
    ESCRIBIR "8 x 1 = ", resultado
    resultado <- 8 * 2
    ESCRIBIR "8 x 2 = ", resultado
    resultado <- 8 * 3
    ESCRIBIR "8 x 3 = ", resultado
    ...
    resultado <- 8 * 10
    ESCRIBIR "8 x 10 = ", resultado
FIN
```

Es evidente que hay dos acciones que se repiten a través de todo el algorimo con una leve variación en el número por el cual se está multiplicando al 8. Esto puede resumirse así:

```
ALGORIMO: "Mostrar tabla del 8"
```

```
COMENZAR
    VARIABLE numérica resultado
    PARA i DESDE 1 HASTA 10 HACER
        resultado <- 8 * i
        ESCRIBIR "8 x " i " = " resultado
    FIN PARA
FIN
```

De manera general, la sintaxis para este tipo de estructuras es:

```
PARA <variable> DESDE <valor1> HASTA <valor2> CON PASO <valor3> HACER
    Acción/es
FIN PARA
```

Dado un valor inicial `<valor1>` asignado a la `<variable>`, esta se irá aumentando o disminuyendo según el paso `<valor3>` hasta llegar a tomar el valor `<valor3>`. Si no se indica el paso se asume que la variable aumenta de uno en uno.

**En IML**.

```
/* PROGRAMA: Mostrar tabla del 8 */
proc iml;
    do i = 1 to 10;
        resultado = 8 * i;
        print "8 x " i " = " resultado;
    end;
quit;
```

### 6.3.2. Estructuras de control iterativas con un número indeterminado de iteraciones

En otras circunstancias se puede necesitar repetir un bloque de acciones sin conocer con exactitud cuántas veces, si no que esto depende de algún otro aspecto del ALGORITMO. Las iteraciones pueden continuar *mientras que* o *hasta que* se verifique alguna condición, dando lugar a dos tipos de estructuras.

#### 6.3.2.1. Mientras que

El conjunto de sentencias se repite mientras que se siga evaluando como `VERDADERO` una condición declarada al inicio del bloque. Cuando la condición ya no se cumple, el proceso deja de ejecutarse. La sintaxis es:

```
MIENTRAS QUE <condición> HACER
    Acción/es
FIN MIENTRAS
```

El siguiente ejemplo describe el algoritmo para escribir los múltiplos de 8 menores a 150 utilizando este tipo de estructura:

```
ALGORITMO: "Múltiplos de 8 menores a 150"
COMENZAR
    VARIABLE numérica multiplo
    multiplo <- 8
    MIENTRAS QUE multiplo < 150 HACER
        ESCRIBIR multiplo
        multiplo <- multiplo + 8
    FIN MIENTRAS
FIN
```

Observaciones:

- La evaluación de la condición se lleva a cabo antes de cada iteración, incluyendo la primera. Si la condición es `FALSO` inicialmente, entonces las acciones en el cuerpo de la estructura no se ejecutan nunca.
- La evaluación de la condición **sólo** se lleva a cabo al inicio de cada iteración. Si la condición se vuelve `FALSO` en algún punto durante la ejecución de un bloque, el programa no lo nota hasta que se termine de ejecutar el bloque y la condición sea evaluada antes de comenzar la próxima iteración.

**En IML**.

```
/* PROGRAMA: Múltiplos de 8 menores a 150 */
proc iml;
    multiplo = 8;
    do while (multiplo < 150);
        print multiplo;
        multiplo = multiplo + 8;
    end;
quit;
```

#### 6.3.2.2. Hasta que

A diferencia de la estructura *mientras que*, la estructura *hasta que* repite el bloque de acciones hasta que se cumpla una condición, es decir, se ejecuta mientras que dicha condición sea evaluada como `FALSA`. La sintaxis es:

```
REPETIR
    Acción/es
HASTA QUE <condición>
```

El algoritmo del ejemplo anterior puede ser re escrito con este tipo de estructura:

```
ALGORITMO: "Múltiplos de 8 menores a 150"
COMENZAR
    VARIABLE numérica multiplo
    multiplo <- 8
    REPETIR
        ESCRIBIR multiplo
        multiplo <- multiplo + 8
    HASTA QUE multiplo >= 150
FIN
```

Observación: en la estructura *mientras que* podría ser que el conjunto de sentencias nunca llegue a ejecutarse si desde partida la condición evaluada ya es falsa. Por el contrario, en la estructura *hasta que* el proceso se realiza al menos una vez, dado que la condición se evalúa al final.

**En IML**.

```
/* PROGRAMA: Múltiplos de 8 menores a 150 */
proc iml;
    multiplo = 8;
    do until (multiplo >= 150);
        print multiplo;
        multiplo = multiplo + 8;
    end;
quit;
```

# Capítulo 7

# Estructuras de Datos

Hasta ahora todos los algoritmos que hemos desarrollado hacen uso de objetos con datos individuales, que representaban un número, una cadena de texto o un valor lógico. Sin embargo, la verdadera utilidad de la computación radica en poder trabajar con conjuntos de datos. En este capítulo introduciremos el concepto de un **arreglo** (o *array*) que es una colección ordenada de valores del mismo tipo. Los arreglos son muy útiles para almacenar información en la memoria de la computadora, organizando valores que estén relacionados entre sí de alguna manera, por ejemplo, una lista de precios, los meses del año, el listado de calificaciones de los alumnos, etc.

Un **arreglo** se define entonces como una colección de valores individuales con dos características fundamentales:

- *Ordenamiento*: los valores individuales pueden ser enumerados en orden, debe ser posible identificar en qué posición del arreglo se encuentra cada valor.
- *Homogeneidad*: los valores individuales almacenados en un arreglo son todos del mismo tipo (numérico, caracter, lógico).

Antes de poder utilizar un arreglo, hay que reservar una zona de la memoria para su uso, así como definir el número de parámetros necesarios para acceder a cada elemento de la estructura, es decir, **dimensionarlos**, lo cual permite clasificar a los arreglos en *unidimensionales*, *bidimensionales* o *multidimensionales*.

## 7.1. Arreglos unidimensionales

Un **arreglo unidimensional** representa lo que habitualmente conocemos con un **vector** y por lo tanto también se lo llama de esa manera. Un **vector** tiene


Figura 7.1: Ejemplo vector numérico


Figura 7.2: Ejemplo vector caracter

*n* elementos ordenados todos del mismo tipo. Un ejemplo de un vector de tipo numérico llamado *vector1* con 5 elementos puede ser:

Un ejemplo de un vector de tipo caracter llamado *vector2* con 4 elementos puede ser:

Los elementos en cada uno de estos vectores ocupan una determinada posición y pueden ser accedidos a través del uso de **índices**, expresados con corchetes al lado del nombre del vector. Por ejemplo, la acción `ESCRIBIR vector1[3]` nos mostrará el valor 2.71.

Como todas las variables que empleamos en nuestros algoritmos, los vectores deben ser declarados en el mismo y su tamaño debe ser especificado al comenzar. Esto se realiza a través de la acción `DIMENSIONAR`. Por ejemplo, el *vector1* visto anteriomente puede ser creado de la siguiente forma:

```
DIMENSIONAR numérico vector1(5)
vector1[1] <- -4.5
vector1[2] <- 12
vector1[3] <- 2.71
```

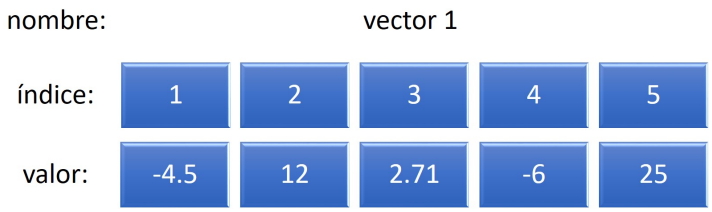


Figura 7.3: Ejemplo vector numérico con posiciones indexadas

```
vector1[4] <- -6
vector1[5] <- 25
```

Cuando declaramos un vector que usaremos más adelante, especificamos cuántos elementos entrarán en él poniendo su dimensión entre paréntesis al lado del nombre.

Podemos asignar valores a las posiciones del vector empleando estructuras de control iterativas, por ejemplo:

```
DIMENSIONAR numérico vector3(30)
PARA i DESDE 1 HASTA 30 HACER
    vector3[i] <- i * 2
FIN PARA
```

En este ejemplo, la variable `i` se usa como índice para el espacio en el vector que será modificado y también para calcular el valor por asignar, resultando en un vector con los números 2, 4, 6, …, 60.

En el próximo ejemplo, se deja que el usuario determine la dimensión del vector y que provea cada uno de los valores para el mismo:

```
VARIABLE numérico tam
LEER tam
DIMENSIONAR numérico vector4(tam)
PARA i DESDE 1 HASTA tam HACER
    LEER vector4[i]
FIN PARA
```

## 7.2. Arreglos bidimensionales

Un **arreglo bidimensional** representa lo que habitualmente conocemos con una **matriz** y también lo podemos llamar de esa forma. A diferencia de los vectores, las matrices requieren dos índices o parámetros para acceder a sus elementos, sobre los cuales nos referimos como **fila** y **columna**. Se pueden utilizar dos estructuras *PARA… FIN PARA* anidadas para recorrer todos los elementos de la matriz, como se muestra en el siguiente ejemplo:

```
DIMENSIONAR numérico matriz1(3, 4)
PARA i DESDE 1 HASTA 3 HACER
    PARA j DESDE 1 HASTA 4 HACER
        matriz1[i, j] <- i * j
    FIN PARA
FIN PARA
```

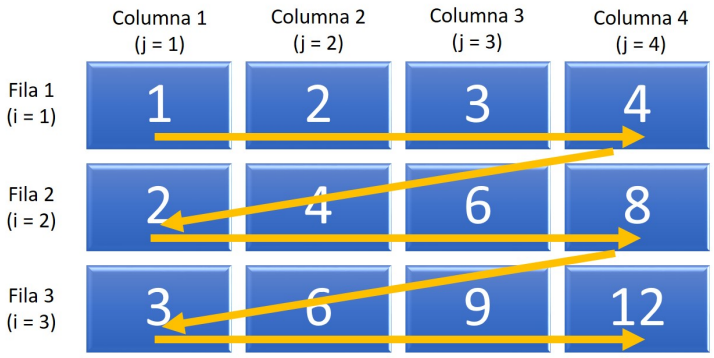


Figura 7.4: Ejemplo: matriz1

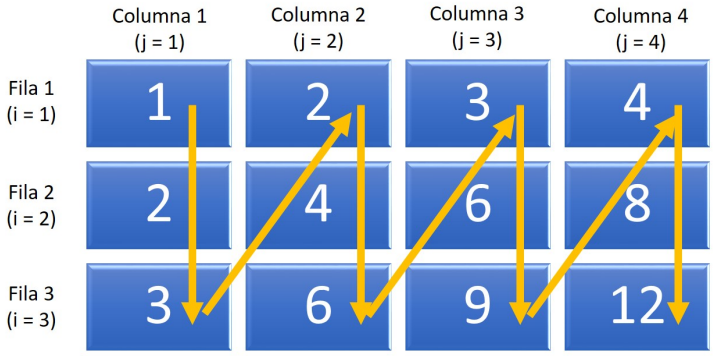


Figura 7.5: Ejemplo: matriz1 recorrida por columnas

En el ejemplo anterior los valores fueron asignados recorriendo la matriz por filas como lo indican las flechas. Otra posibilidad es recorrer la matriz por columna en primer instancia, para lo cual la estructura *PARA… FIN PARA* que representa a los índices de las columnas debe ser la externa y la que representa a los índices de columnas, la interna:

```
DIMENSIONAR numérico matriz1(3, 4)
PARA j DESDE 1 HASTA 4 HACER
    PARA i DESDE 1 HASTA 3 HACER
        matriz1[i, j] <- i * j
    FIN PARA
FIN PARA
```


Figura 7.6: Vector v original


Figura 7.7: Vector v reordenado

## 7.3. Arreglos multidimensionales

Un **arreglo multidimensional** contiene más de dos dimensiones. Aunque los vectores y matrices son los tipos de arreglos más usados, podemos emplear tantos índices para localizar los elementos del arreglo como estimemos necesarios. La representación matemática o visual ya no es tan sencilla. Para interpretarlos o saber cuándo usarlos, pensamos que cada uno de las dimensiones representa una característica, condicionante o parámetro definidor del elemento.

Por ejemplo, si se desea contar el número de autos que ingresaron a una playa de estacionamiento por hora a lo largo de varios años, podríamos utilizar un arreglo donde la primera dimensión indique el año, la segunda el mes, la tercera el día y la cuarta la hora. Si llamamos a este arreglo `numeroAutos`, el elemento `numeroAutos[2, 4, 23, 14]` contendrá el número de autos que ingresaron a la hora 14, del día 23, en el mes 4 del segundo año.

## 7.4. Ejemplo: invertir los elementos de un vector

Nos planteamos el problema de dar vuelta los elementos pertenecientes a un vector, de manera que el primer elemento pase a ser el último, el segundo pase al penúltimo lugar, etcétera.

Por ejemplo, dado el vector `v`:

Queremos modificarlo para obtener:

Si bien podemos pensar en distintas formas para resolver este problema, probablemente la más sencilla requiere que intercambiemos de a dos los valores en ciertas posiciones del vector, por ejemplo, el primero y el último. Para esto podemos emplear una variable auxiliar para guardar el valor de alguna de las

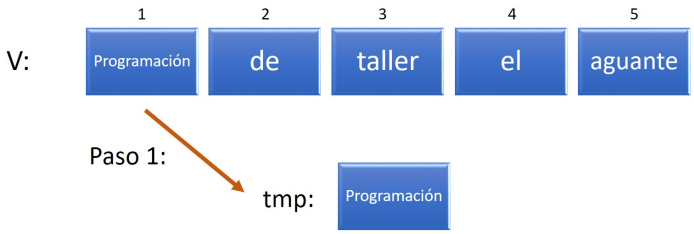


Figura 7.8: Vector v reordenado

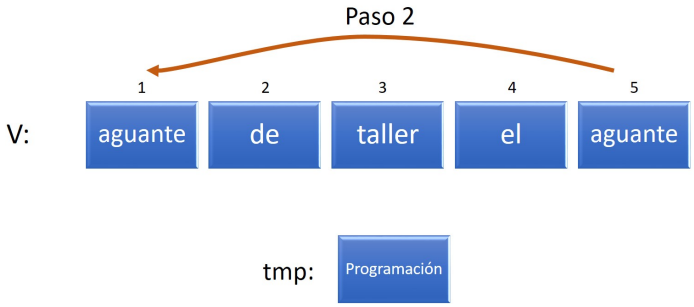


Figura 7.9: Vector v reordenado

celdas temporariamente (por eso lo vamos a llamar `tmp`) y poder realizar el intercambio:

Ahora sólo resta realizar el mismo procedimiento para los valores de las posiciones 2 y 4. Como el número de elementos en el vector es impar, el valor en la posición central queda en su lugar. Podemos definir el siguiente algorimo para resolver este problema de manera general:

```
ALGORITMO: "Invertir (dar vuelta) los elementos de un vector"
COMENZAR
```

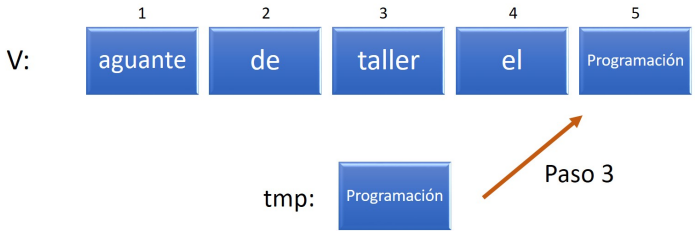


Figura 7.10: Vector v reordenado

```
    \\ Declarar variables
    VARIABLE numérico n
    VARIABLE caracter tmp
    LEER n
    DIMENSIONAR caracter v(n)

    \\ Asignar valores al vector
    PARA i DESDE 1 HASTA n HACER
        LEER v[i]
    FIN PARA

    \\ Reordenar
    PARA i DESDE 1 HASTA ENTERO(n / 2) HACER
        tmp <- v[i]                 \\ Paso 1
        v[i] <- v[n - i + 1]        \\ Paso 2
        v[n - i + 1] <- tmp         \\ Paso 3
    FIN PARA

    \\ Mostrar el vector reodenado
    PARA i DESDE 1 HASTA n HACER
        ESCRIBIR v[i]
    FIN PARA

FIN
```

El código correspondiente en SAS/IML es:

```
/* Invertir (dar vuelta) un vector */
proc iml;
    v = {"programacion" "de" "taller" "el" "aguante"};
    n = ncol(v);
    do i = 1 to int(n / 2);
        tmp = v[i];
        v[i] = v[n - i + 1];
        v[n - i + 1] = tmp;
    end;
    print v;
quit;
```

# Capítulo 8

# Archivos

Como hemos visto, los programas usan variables para guardar información: datos de entrada, resultados calculados, valores intermedios. Sin embargo, la información guardada en las variables es efímera. Cuando los programas paran de correr, el valor almacenado en las variables se pierde. En muchas ocasiones, es necesario guardar información de una forma más permanente.

En estos casos, el enfoque usual es recolectar la información en un todo lógicamente cohesivo y guardarlo en un medio permanente que generalmente se graba en el disco rígido de la máquina, es decir, en un archivo. Un **archivo** o **fichero** es un conjunto de información sobre un mismo tema tratado como una unidad de almacenamiento y organizado de forma estructurada para la búsqueda de un dato individual. Los archivos pueden contener instrucciones de programas o información creada o usada por un programa. Todos los objetos de datos permanentes que guardamos en nuestra computadora (documentos, juegos, programas ejecutables, código, etc.) son guardados en la forma de archivos.

La unidad elemental que compone a un archivo o fichero es un **registro**, el cual es una colección de información relativa a una misma entidad. En general, cada registro de un mismo archivo tiene la misma estructura que los demás. Los datos individuales sobre dicha entidad ocupan **campos** dentro de los registros. Por ejemplo:

```
ARCHIVO: Pasajeros
Campo 1: NOMBRE, tipo caracter
Campo 2: NÚMERO DE VUELO, tipo caracter
Campo 3: FECHA DE VUELO, tipo caracter
Campo 4: NÚMERO DE ASIENTO, tipo caracter
Campo 5: CIUDAD ORIGEN, tipo caracter
Campo 6: CIUDAD DESTINO, tipo caracter
Campo 7: PRECIO, tipo numérico
```

| Nombre | NroVuelo | FechaVuelo | NroAsiento | Origen | Destino | Precio |
|---|---|---|---|---|---|---|
| Pamela Suárez | AR6071 | 12/09/17 | 17A | Rosario | Córdoba | 1532.23 |
| Gonzalo Echarri | AR5423 | 14/09/17 | 31B | Rosario | Buenos Aires | 1424.10 |
| ... | ... | ... | ... | ... | ... | ... |

La forma más común de identificar un registro es eligiendo un campo dentro del registro llamado **clave** (por ejemplo, el nombre del pasajero), que contiene un único valor para cada registro. En otras casos es posible identificar un registro a través del valor de más de uno de sus campos.

## 8.1. Organización de archivos

Existen distintos tipos de organización de los archivos según la forma en la que se pueda acceder a cada uno de sus registros:

- Archivo secuencial

  Los registros se encuentran en cierto orden que debe ser respetado para la lectura de los mismos. Para leer el registro situado en la posición $n$, el programa previamente tiene que pasar por todos los registros que ocupan las posiciones anteriores. Los registros pueden leerse uno por uno hasta llegar al final del archivo. La mayoría de los lenguajes de programación disponen de una función lógica que devuelve un valor **VERDADERO** cuando se alcanza el final del archivo.

- Archivo directo

  Está formado por un conjunto de registros que pueden ser recuperados por su posición dentro del archivo sin necesidad de recorrer los anteriores.

- Archivo indexado

  Dispone de una tabla de índices adicional, es decir, una referencia que permite obtener de forma automática la ubicación de la zona del archivo donde se encuentra el registro buscado. Esto permite localizar un registro por medio de su clave, o del valor de algún campo en particular, sin recorrer previamente los registros que lo preceden.

La organización más sencilla y más comunmnete empleada es la secuencial, aunque no sea la más eficiente. Nosotros trabajaremos en el Taller con este tipo de organización.

## 8.2. Operaciones sobre archivos

Los procedimientos básicos que los programas pueden llevar a cabo sobre los distintos tipos de archivos son:

- Creación de un archivo

  Consiste en la escritura de los registros que van a conformar el archivo. Los datos pueden introducirse por teclado, desde otro archivo o como resultado de algún proceso intermedio.

- Apertura y cierre de un archivo

  Para que un programa pueda operar directamente sobre un archivo, la primera operación que debe realizar es la **apertura** del mismo, que incluye la identificación del archivo a utilizar y el modo (lectura, escritura, etc.). Cuando un programa no vaya a acceder más a un archivo, es necesario indicarlo a través del **cierre** del mismo, ya que se liberan memoria y recursos del sistema, se previene la corrupción de los datos si se detiene el programa mientras se está ejecutando y expresa explícitamente que ya no se hará más uso del mismo.

- Lectura y escritura en un archivo

  La **lectura** consiste en transferir información del archivo a la memoria principal usada por el programa, mientras que la **escritura** es la transferencia de información guardada en las variables del programa al archivo.

Otras operaciones que se pueden realizar sobre los archivos incluyen acciones de:

- Actualización: añadir (dar de alta), modificar o eliminar (dar de baja) algún registro.
- Clasificación: reubicar los registros de tal forma que queden ordenados por algún campo determinado.
- Fusión o mezcla: combinar dos o más archivos para formar uno nuevo.
- Partición: subdividir los registros por el valor que toman en algún campo.

## 8.3. Pseudo-código

Al escribir los algoritmos en pseudo-código, se puede hacer uso de las siguientes expresiones para representar algunas operaciones a realizar sobre los archivos:

- `ABRIR (nombre del archivo) secuencial, de entrada/salida`: indica la acción de apertura de un archivo con organización secuencial, sobre el cual se va a proceder a realizar lectura o escritura de registros.

- `CERRAR (nombre del archivo)`: para indicar que el archivo no será vuelto a utilizar.
- `LEER (nombre del archivo) campo1, campo2, ...`: indica la lectura del próximo registro cuando se trabaja con un archivo secuencial. La expresión `campo1, campo2, ...` son los nombres de los campos que contienen los registros y constituyen los identificadores de las variables que almacenarán temporalmente los datos del registro que se están leyendo.
- `LEER (nombre del archivo, número de registro) campo1, campo2, ...`: indica la lectura de un registro en particular cuando se trabaja con un archivo de acceso directo.
- `LEER (nombre del archivo, campo1 = valor) campo1, campo2, ...`: indica la lectura de cada uno de los registros en los cuales el campo `campo1` es igual a `valor`, para cuando se trabaja con un archivo de acceso indexado.
- Las tres acciones de `LEER` anteriores tienen su acción análoga de `ESCRIBIR`, para modificar o añadir registros al archivo.
- `FINDE (nombre del archivo)`: es la función lógica que devuelve el valor `FALSO` mientras resten más registros por leer en el archivo y el valor `VERDADERO` cuando se llega al final del archivo y ya no quedan más registros por leer.

### 8.3.1. Ejemplo: lectura de un archivo

Siendo `nombreArchivo` el nombre del archivo que se desea leer, compuesto por `m` campos llamados `campo1`, `campo2`, …, `campom`, entonces, el pseudocódigo necesario para leer el archivo es:

```
ABRIR(nombreArchivo) secuencial, de entrada
MIENTRAS FINDE(nombreArchivo) = FALSO ENTONCES
    LEER(nombreArchivo) campo1, campo2, ..., campom

    ... realizar operaciones con las variables campo1, campo2, ..., campom...

FIN MIENTRAS
CERRAR(nombreArchivo)
```

En IML, podemos seguir los siguientes pasos:

1. Haciendo uso de un `proc import`, se importa el archivo en la librería temporal `Work` (podría guardarse en otra), creando un dataset cuyo nombre se indica en `out` (en este caso, `nombreDataset`):

```
* Si se trata de un archivo de texto;
```

```
proc import out = nombreDataset
            datafile = "C:\direccion\hasta\la\carpeta\nombreArchivo.txt"
            dbms = TAB REPLACE;
run;

* Si se trata de un archivo de Excel;
proc import out = nombreDataset
            datafile = "C:\direccion\hasta\la\carpeta\nombreArchivo.txt"
            dbms = EXCEL REPLACE;
run;
```

2. Dentro del `proc iml`, leer línea por línea los registros del data set que hemos creado (en este ejemplo, `nombreDataset`) a través de un `do data;`:

```
proc iml;

    ...

    use nombreDataset;
    do data;

        * Se lee de a una línea (registro) por vez, los campos del archivo son usados como
            variables que tienen los respectivos valores para la línea en cuestión;

       ... realizar operaciones con los valores de cada registro del archivo...

    end;
    close nombreDataset;

    ...

quit;
```

### 8.3.2. Ejemplo: escritura de un archivo

Supongamos que la información que se desea guardar en un nuevo archivo se encuentra en una matriz llamada `tabla`, que tiene `n` filas correspondientes a `n` registros y `m` columnas referidos a `m` campos llamados `campo1`, `campo2`, ..., `campom`. Entonces, el pseudocódigo necesario para crear y escribir el nuevo archivo es:

```
CREAR(nombreArchivo)
ABRIR(nombreArchivo) secuencial, de salida
PARA i DESDE 1 HASTA n HACER
    campo1 <- tabla[i, 1]
```

```
    campo2 <- tabla[i, 2]
    ...
    campom <- tabla[i, m]
    ESCRIBIR(nombreArchivo) campo1, campo2, ..., campom
FIN PARA
CERRRAR(nombreArchivo)
```

En IML, a partir de la matriz que tiene la información que se desea guardar en un archivo, se deben seguir dos pasos:

1. Dentro del mismo `proc iml`, crear un dataset de SAS a partir de la matriz. En este ejemplo, la matriz se llama `tabla`, el nuevo dataset `datos` y se guarda en la librería temporal `Work` (podría guardarse en otra):

```
proc iml;

    ... todo el código necesario ...

    create datos from tabla[colname = ("campo1" || "campo2" || ... || "campom")];
    append from tabla;
    close datos;
quit;
```

2. Haciendo uso de un `proc export`, se exporta el dataset desde la librería Work hasta una carpeta de la computadora, en este ejemplo, como archivo de texto:

```
proc export data = datos
            outfile = "C:\direccion\hasta\la\carpeta\deseada\nombreArchivo.txt"
            dbms = TAB REPLACE;
run;
```

## 8.4. Archivos de texto

Archivos como el del ejemplo de los pasajeros suelen ser guardados como **archivos de texto**. Un **archivo de texto** (también conocido como *texto llano* o *texto simple*), es un archivo informático que contiene únicamente texto formado por una secuencia ordenada de caracteres. El texto almacenado en este tipo de archivo carece de cualquier tipo de formato tipográfico (negrita, cursiva, colores, subrayado, fuente, etc.), lo cual permite que una gran variedad de programas pueda leer y editar el contenido.

Los archivos de texto están compuestos por caracteres ordinarios, como las letras, números y signos de puntuación, y por caracteres especiales que indican,

por ejemplo, saltos de línea y tabulaciones. Como las computadoras solamente entienden números, cada caracter es codificado internamente con un representación numérica binaria. Distinas maneras de hacer esta representación dan lugar a diferentes formatos de codificación de caracteres (como *ASCII*, *ISO-8859-1* o *UTF-8*).

La información contenida en los archivos que utilizaremos en el Taller puede ser almacenada de manera práctica en archivos de texto.

# Capítulo 9

# Subalgortimos

Un principio fundamental en la resolución de un problema es intentar descomponerlo en partes más pequeñas, que puedan ser más fáciles de afrontar. Este concepto también se aplica en la programación. Nuestros algoritmos pueden descomponerse en **subalgoritmos** que den solución a un aspecto del problema, de menor extensión. Este proceso se conoce como **descomposición algorítmica** o **descomposición modular**. Cada subalgoritmo debe ser independiente de los demás y a su vez podría seguir descomponiéndose en partes más sencillas en lo que se conoce como **refinamiento sucesivo**. Si un programa es muy largo se corre el riesgo de que sea muy difícil de entender como un todo, pero siempre se lo puede dividir en secciones más simples y manejables. Un subalgoritmo se escribe una vez y luego es utilizado por todos aquellos algoritmos que lo necesiten.

Observemos el siguiente ejemplo, que presenta un algoritmo para el cálculo de un número combinatorio. Recordemos que el número combinatorio entre $n$ y $k$ se define como:

$$\binom{n}{k} = \frac{n!}{(n-k)!k!} \qquad k \leq n, k \in \mathbb{N}_0, n \in \mathbb{N}$$

```
ALGORITMO: Ej 1. Cálculo de números combinatorios
COMENZAR
    VARIABLE numerica n, k, fact1, fact2, fact3, comb
    LEER n, k \\ Asumimos que n y k cumplen con los requisitos

    \\ Calcular el factorial de n
    fact1 <- 1
    PARA i DESDE 1 HASTA n HACER
        fact1 <- fact1 * i
    FIN PARA
```

```
    \\ Calcular el factorial de n-k
    fact2 <- 1
    PARA i DESDE 1 HASTA n - k HACER
        fact2 <- fact2 * i
    FIN PARA

    \\ Calcular el factorial de k
    fact3 <- 1
    PARA i DESDE 1 HASTA k HACER
        fact3 <- fact3 * i
    FIN PARA

    \\ Calcular el nro combinatorio
    comb <- fact1 / (fact2 * fact3)

    ESCRIBIR "El nro combinatorio de " n " tomado de a " k " es " comb
FIN
```

Como se puede observar, el cálculo del factorial requiere siempre la misma estructura y se repite tres veces. Esto constituye una parte del problema cuya resolución puede plantearse por separado, dando lugar a un subalgoritmo. El algoritmo quedaría mejor expresado de la siguiente manera:

```
ALGORITMO: Ej 2. Cálculo de números combinatorios
COMENZAR
    VARIABLE numérica n, k, comb
    LEER n, k
    comb <- factorial(n) / (factorial(n - k) * factorial(k))
    ESCRIBIR "El nro combinatorio de " n " tomado de a " k " es " comb
FIN
```

Aquí se puede ver cómo se simplificó la estructura del algoritmo, al hacer uso de un subalgoritmo llamado `factorial` que toma entre paréntesis un valor para el que procede a calcular y devolver el factorial. Para que esto funcione, debemos definir aparte dicho subalgoritmo, como se muestra a continuación. Más adelante veremos los detalles de esta definición y por qué, en particular, se dice que este subalgoritmo es una *función*:

```
FUNCIÓN factorial(n: numérico): numérico
    VARIABLE numérica fact
    fact <- 1
    PARA i DESDE 1 HASTA n HACER
        fact <- fact * i
    FIN PARA
```

```
    DEVOLVER fact
FIN FUNCIÓN
```

El empleo de subalgoritmos, desarrollando por separado ciertas partes del problema, resulta especialmente ventajoso en los casos siguientes:

- **En algoritmos complejos**: si el algoritmo, y luego el programa, se escribe todo seguido resulta muy complicado de entender, porque se pierde la visión de su estructura global dada la gran cantidad de operaciones que lo conforman. Aislando ciertas partes como subalgoritmos separados se reduce la complejidad.
- **Cuando se repiten operaciones análogas**: si la resolución de un problema requiere realizar una tarea que se repite varias veces en el algoritmo, podemos definir dicha tarea como un subalgoritmo por separado. De esta manera, su código se escribirá sólo una vez aunque se use en muchos puntos del programa.

Los subalgoritmos se clasifican en **funciones** y **procedimientos**. Las **funciones** devuelven como resultado un solo valor al algoritmo principal. Los procedimientos, en cambio, pueden devolver cero, uno o varios valores.

## 9.1. Funciones

Una **función** es un subalgoritmo que devuelve un único resultado. Ya hemos trabajado con funciones que asumimos predefinidas, como por ejemplo las funciones módulo (`MOD`), valor absoluto (`ABS`) o raiz cuadrada (`RAIZ`), pero ahora veremos que podemos definir nuestras propias funciones. El valor que la función devuelve como resultado define su tipo, de modo que una función puede ser de tipo numérica, caracter o lógica. En programación, la noción de función se asemeja a la idea matemática de función de una o más variables. Por ejemplo, podemos pensar en la función $f(x, y) = x^2 + 3y$. En pseudocódigo, el subalgoritmo que se encargaría de implementarla es:

```
FUNCIÓN f(x: numérico, y: numérico): numérico
COMENZAR
    DEVOLVER x ** 2 + 3 * y
FIN FUNCIÓN
```

Dado que esta función devuelve un valor numérico, decimos que la misma es de tipo numérico, lo cual se indica al final del encabezado. Se dice que $x$ e $y$ son los **parámetros formales** o **ficticios** y son los que permiten expresar la "ley" o "forma" de la función. También se aclara en el encabezado que estos parámetros son de tipo numérico. Los valores en los cuales se quiere evaluar la

función se llaman **parámetros actuales** o **reales**. Por ejemplo, si nos interesa calcular $f(4, 5)$, los valores 4 y 5 son los parámetros actuales y se establece una correspondencia entre el parámetro formal $x$ y el real 4, así como entre la $y$ y el 5. Como veremos más adelante, dicha correspondencia puede establecerse de distintas formas. En este ejemplo, el resultado que se obtiene es 31. A los parámetros también se les dice **argumentos**.

También puede ser expresada como:

```
FUNCIÓN f(x: numérico, y: numérico): numérico
COMENZAR
    VARIABLE numérica rtdo
    rtdo <- x ** 2 + 3 * y
    DEVOLVER rtdo
FIN FUNCIÓN
```

De manera general, la definición de una función es:

```
FUNCIÓN nombre(lista de parámetros formales): tipo de resultado
COMENZAR
    Declaración de variables
    Acciones
    DEVOLVER valor
FIN FUNCIÓN
```

La palabra clave `DEVOLVER` provoca la inmediata finalización de la ejecución de la función e indica cuál es el resultado de la misma, cuyo tipo debe coincidir con el tipo de función declarado antes. La acción `DEVOLVER` se puede insertar en cualquier punto de la parte ejecutable de la función y además es posible utilizar más de una sentencia `DEVOLVER` en una misma función, aunque sólo una llegue a ejecutarse. Esto puede verse en el siguiente ejemplo:

```
FUNCIÓN maximo(num1: numérico, num2: numérico): numérico
COMENZAR
    SI num1 >= num2
        ENTONCES
            DEVOLVER num1
        SI NO
            DEVOLVER num2
    FIN SI
FIN FUNCIÓN
```

Para usar una función en un algoritmo, se la invoca escribiendo su nombre seguida por los valores actuales entre paréntesis, separados por coma. Esta invocación representa un valor del tipo de la función que puede ser usado como operando

en otra expresión. Al invocar una función es obligatorio que los valores suministrados para los argumentos reales correspondan en cantidad, tipo y orden con los argumentos formales de la definición de la función. Por ejemplo:

```
ALGORITMO: Ej 3. Hallar el máximo entre dos valores
COMENZAR
    ESCRIBIR "El máximo entre 5 y 10 es " maximo(5, 10)
FIN
```

O más general:

```
ALGORITMO: Ej 3. Hallar el máximo entre dos valores
COMENZAR
    VARIABLE numérica x, y
    LEER x, y
    ESCRIBIR "El máximo es " maximo(x, y)
FIN
```

## 9.2. Procedimientos

Un **procedimiento** es un subalgoritmo que agrupa una acción o conjunto de acciones, dándoles un nombre por el que se las puede identificar posteriormente. Se diferencia de la función en que no tiene como objetivo, en general, devolver un valor, pudiendo devolver ninguno, uno o varios. Esto quiere decir que tampoco se declara de qué *tipo* es. El objetivo principal de los procedimientos es ayudar en la modularidad del programa y evitar la repetición de acciones.

Como en las funciones, desde el algoritmo principal se pasan valores al procedimiento utilizando **parámetros** o **argumentos**, aunque también puede haber procedimientos que carezcan de los mismos. Para usar un procedimiento hay que invocarlo, escribiendo su nombre y a continuación, si los hay, los valores de los argumentos actuales para esa llamada, separados por comas. Aquí también los argumentos reales deben ser compatibles en cuanto a la cantidad, tipo y orden que los argumentos formales declarados en la definición del procedimiento.

Por ejemplo, podemos definir un procedimiento que se encargue de escribir un título para la salida de nuestro algoritmo y otro para escribir una línea que separe los resultados:

```
ALGORITMO: Ej 4. Procedimientos con y sin argumentos
COMENZAR
    ...
    colocarTitulo("Primer grupo de resultados")
    ESCRIBIR 1
```

```
    colocarLinea()
    ESCRIBIR 2
    colocarLinea()
    ESCRIBIR 3

    colocarTitulo("Segundo grupo de resultados")
    ESCRIBIR 4
    colocarLinea()
    ESCRIBIR 5
    colocarLinea()
    ESCRIBIR 6
FIN

PROCEDIMIENTO colocarTitulo(titulo: caracter)
    ESCRIBIR "================================="
    ESCRIBIR titulo
    ESCRIBIR "================================="
FIN PROCEDIMIENTO

PROCEDIMIENTO colocarLinea()
    ESCRIBIR "_________________________________"
FIN PROCEDIMIENTO
```

Como resultado la salida mostrará:

```
=================================
Primer grupo de resultados
=================================
1

---------------------------------
2

---------------------------------
3

=================================
Segundo grupo de resultados
=================================
4

---------------------------------
5

---------------------------------
6

---------------------------------
```

En el siguiente ejemplo podemos identificar los argumentos reales `a` (con el valor 5), `b` (con el valor 2), `c` y `d` (sin valores asignados inicialmente). Cuando

el procedimiento `proced1` es invocado, se establece una correspondencia con los argumentos formales `n1`, `n2`, `n3` y `n4`, respectivamente. `n1` toma el valor 5, `n2` toma el valor 2 y el procedimiento le asigna los valores 7 a `n3` y 1 a `n4`. Al finalizar, este procedimiento habrá dejado sin cambios a las variables `a` y `b`, mientras que le habrá asignado los valores 7 a `c` y 1 a `d`. Como resultado, el algoritmo escribe "5 2 7 1".

```
ALGORITMO: Ejemplo 5
COMENZAR
    VARIABLE numérica a, b, c, d
    a <- 5
    b <- 2
    proced1(a, b, c, d)
    ESCRIBIR a b c d
FIN

PROCEDIMIENTO proced1(n1: numérico, n2: numérico, n3: numérico, n4: numérico)
    n3 <- n1 + n2
    n4 <- n2 - 1
FIN PROCEDIMIENTO
```

En el siguiente ejemplo, el procedimiento `proced2` modifica las variables que actúan como argumentos reales. Al ser invocado, se establece una correspondencia entre los argumentos reales `a` (con el valor 5) y `b` (con el valor 2), y los argumentos formales `n1` y `n2`, respectivamente. De esta forma, la primera acción del procedimiento le asigna el valor 7 a `n1` y 1 a `n2`. De esta manera, al finalizar `a` vale 7 y `b` vale 1 y el algoritmo escribe "7 1".

```
ALGORITMO: Ejemplo 6
COMENZAR
    VARIABLE numérica a, b
    a <- 5
    b <- 2
    proced2(a, b)
    ESCRIBIR a b
FIN

PROCEDIMIENTO proced2(n1: numérico, n2: numérico)
    n1 <- n1 + n2
    n2 <- n2 - 1
FIN PROCEDIMIENTO
```

## 9.3. Pasaje de argumentos

Los algoritmos y subalgoritmos comunican información entre sí a través de los parámetros o argumentos y existen distintas formas de realizar esta comunicación.

### 9.3.1. Pasaje por valor

En este caso, los argumentos representan valores que se transmiten **desde** el algoritmo **hacia** el subalgoritmo. Las funciones, además, cuentan con un valor de retorno, que es el valor que se transmite desde el subalgoritmo hacia el algorimo que lo llamó.

El **pasaje por valor** implica que los objetos del algoritmo provistos como argumentos en la llamada al subalgoritmo no serán modificados por la ejecución del mismo. Este sistema funciona de la siguiente forma:

1. Se evalúan los argumentos reales usados en la llamada.
2. Los valores obtenidos se copian en los argumentos formales dentro del subalgoritmo.
3. Los argumentos formales se usan como variables dentro del subalgoritmo. Aunque los mismos sean modificados (por ejemplo, se les asignen otros valores), no se modifican los argumentos reales en el algoritmo, sólo sus copias dentro del subalgoritmo.

En general, se desalienta la reasignación de valor a un argumento pasado por valor por resultar confuso.

### 9.3.2. Pasaje por referencia

En otras situaciones es deseable que el subalgoritmo pueda modificar las variables del algoritmo que se usen como argumentos. De esta manera se puede producir más de un resultado. De esto se encarga el **pasaje por referencia**. Si un parámetro se pasa por referencia, esta variable será empleada en el subalgoritmo como si fuera suya, es decir, las modificaciones que sufra dentro del subalgoritmo la modificarán permanentemente. Este sistema funciona de la siguiente forma:

1. Se seleccionan las variables usadas como argumentos reales.
2. Se asocia cada variable con el argumento formal correspondiente.
3. Los cambios que experimenten los argumentos formales se reflejan también en los argumentos reales de origen.

### 9.3.3. Ejemplos

Algunos lenguajes de programación permiten que el programador elija el modo en el que se realiza el pasaje. En el siguiente ejemplo veremos la diferencia entre ambos modos.

```
ALGORITMO: Ejemplo 7
COMENZAR
    VARIABLE numérica a, b, c
    a <- 3
    b <- 5
    c <- fun(a, b - a)
    ESCRIBIR a b c
FIN

FUNCIÓN fun(x: numérico, y: numérico): numérico
    x <- x + 1
    DEVOLVER x + y
FIN FUNCIÓN
```

Si el pasaje de argumentos se hace por valor, los cambios producidos en el cuerpo de la función sobre los parámetros formales no son transmitidos a los parámetros actuales. Esto significa que los formales son una "copia" de los actuales. Los pasos que sigue el algoritmo son:

- `a = 3, b = 5`
- Al invocar la función: `x = 3, y = 5 - 3 = 2`
- Primera línea de la función: `x = 3 + 1 = 4`
- La función devuelve el valor `x + y = 4 + 2 = 6`
- De regreso en el algoritmo principal: `c` recibe el valor `6`
- Se escribe: `3 5 6`

Si el pasaje de argumentos se hace por referencia, cualquier modificación realizada sobre los parámetros formales es automáticamente realizada también a los actuales. Los pasos que sigue el algoritmo son:

- `a = 3, b = 5`
- Al invocar la función: `x = 3, y = 5 - 3 = 2`
- Primera línea de la función: `x = 3 + 1 = 4`. El parámetro actual asociado con `x`, `a`, sufre el mismo cambio y recibe el valor `4` (`a = 4`).
- La función devuelve el valor `x + y = 4 + 2 = 6`
- De regreso en el algoritmo principal: `c` recibe el valor `6`
- Se escribe: `4 5 6`

Analicemos ahora el tipo de pasaje en el contexto de un procedimiento:

```
ALGORITMO: Ejemplo 8
COMENZAR
    VARIABLE numérica a, b
    a <- 8
    b <- 4
    miProc(a, b)
    ESCRIBIR a b
FIN

PROCEDIMIENTO miProc(x: numérico, y: numérico)
    x <- x * 2
    y <- x - y
FIN PROCEDIMIENTO
```

Si el pasaje es por referencias, los pasos que sigue el algoritmo serían:

- `a = 8, b = 4`
- Al invocar la función: `x = 8, y = 4`
- Primera línea de la función: `x = 8 * 2 = 16`. Lo mismo sucede con el parámetro actual `a`: `a = 16`.
- Segunda línea de la función: `y = 16 - 4 = 12`. Lo mismo sucede con el parámetro actual `b`: `b = 12`.
- Al regresar al algoritmo principal, la sentencia `ESCRIBIR` produce: `16 12`.

Si el pasaje hubiese sido por valor, `a` y `b` no hubiesen cambiado y la sentencia `ESCRIBIR` mostraría `8, 4`. Como en un procedimiento los resultados regresan en los mismos parámetros, no pueden ser todos pasados por valor, porque en ese caso el procedimiento nunca realizaría ninguna acción.

Si el parámetro `x` se pasa por valor mientras que `y` se pasa por referencia, los pasos serían:

- `a = 8, b = 4`
- Al invocar la función: `x = 8, y = 4`
- Primera línea de la función: `x = 8 * 2 = 16`.
- Segunda línea de la función: `y = 16 - 4 = 12`. Lo mismo sucede con el parámetro actual `b`: `b = 12`.
- Al regresar al algoritmo principal, la sentencia `ESCRIBIR` produce: `8 12`.

## 9.4. Variables locales y globales

Como ya sabemos, en los algoritmos definimos variables que son de ayuda para la resolución de los problemas. De la misma forma, también se pueden definir variables dentro de los subalgoritmos. Por esta razón, podemos distinguir entre

variables **locales** y **globales**, haciendo referencia a cuál es su alcance o en qué ámbito existen:

- **Variable local**: es aquella que está declarada dentro de un subalgoritmo, en el sentido de que "existe" dentro de ese subalgoritmo. No tiene nada que ver con las variables que puedan ser declaradas con el mismo nombre en cualquier parte del algoritmo principal o de otros subalgoritmos. Cuando otro subalgoritmo utiliza el mismo nombre se refiere a una posición diferente en memoria.
- **Variable global**: es aquella que está declarada en el algoritmo principal. Es accesible para todos los subalgoritmos que de él dependen, sin ser pasada como argumento.

La parte del algoritmo en que una variable se define se conoce como **ámbito** (*scope*, en inglés).

El uso de variables locales tiene muchas ventajas. Las variables locales permiten independizar al subalgoritmo del algoritmo principal. Las variables definidas localmente en un subalgoritmo no son reconocidas fuera de él. La comunicación entre el subalgoritmo y el algoritmo principal se da exclusivamente a través de la lista de parámetros. Esta característica hace posible dividir grandes proyectos en piezas más pequeñas independientes. Cuando diferentes programadores están implicados, pueden trabajar independientemente.

Las variables globales tienen la ventaja de compartir información entre diferentes subalgoritmos y el algoritmo principal sin tener que hacer menciones en la lista de parámetros de los subalgoritmos.

Analicemos el siguiente ejemplo:

```
ALGORITMO: Ejemplo 9
COMENZAR
    VARIABLE GLOBAL numérica z
    VARIABLE LOCAL numérica x, y
    x <- 2
    z <- 10
    y <- fcn(x)
    ESCRIBIR x z y
FIN

FUNCIÓN fcn(l: numérico): numérico
    VARIABLE GLOBAL numérica z
    VARIABLE LOCAL numérica x
    z <- 5
    x <- 7
    DEVOLVER l + 4
FIN FUNCIÓN
```

`z` es una variable global, es decir, puede ser accedida desde el algoritmo o desde la función. `x` es el nombre con el que se indican dos variables. Una es local al algoritmo principal y, por lo tanto, puede ser accedida sólo desde él. La otra es local a la función.

Cuando se realiza la ejecución de este algoritmo, se dan los siguientes pasos:

- `x` recibe el valor 2
- `z` recibe el valor 10
- se llama a la función y se establece la correspondencia entre el parámetro formal `l` y el actual `x` (`l` = 2) y se ejecuta la función:
  - la variable global `z` toma el valor 5 (deja de valer 10)
  - la variable local `x` toma el valor 7, pero la `x` del algoritmo principal no se modifica, siguen con valor 2.
  - se devuelve el valor 2 + 4 = 6
- de regreso en el algoritmo principal, `y` recibe el valor 6
- se escribe "2 5 6", es decir, los valores de `x`, `z` e `y`

### 9.4.1. Transparencia referencial

Como ya hemos mencionado, cuando se escribe un algoritmo o un programa siempre debe buscarse mantener cierta claridad. En lo que respecta a los subalgoritmos, un principio deseable de claridad se denomina **transparencia referencial**, que se logra cuando el subalgoritmo sólo utiliza elementos mencionados en la lista de argumentos o definidos localmente, sin variables globales. Esto garantiza que, cada vez que se la invoque con los mismos valores en los argumentos de entrada, el subalgoritmo produzca el mismo resultado.

El uso de variables globales permite escribir subalgoritmos que carecen de transparencia referencial. Si un subalgoritmo modifica alguna variable externa, se dice que produce *efectos secundarios*, debe realizarse con precaución y generalmente es desaconsejable.

## 9.5. Código en IML de los ejemplos vistos

```
/* Crear una librería en mi computadora */
libname subalg "C:\direccion\hasta\una\carpeta";

/*
Observaciones:
```

```
- En su defincion, las funciones deben hacer uso de la sentencia "return",
  los procedimientos no.
- En su invocación, los procedimientos se corren usando run nombreProc(...);
  o call nombreProc(...); pero las funciones se corren directo sin usar call o run.
- Cuando definimos los módulos, hay que decir dónde se guardarán (con "reset"). Yo los voy a
  guardar dentro de la librería creada arriba "subalg", en un tipo especial de archivo
  de SAS llamado "catálogo". Al catálogo le puedo poner cualquier nombre, en este caso, "ejemplos
- Después de crear el/los módulo/s, debo guardarlos. Puedo guardarlos todos juntos si están
  dentro del mismo proc iml con store module = _all_ o uno por uno con store module = nombreDelMo
- SAS IML usa PASAJE POR REFERENCIA!!!! Cuidado!!! Todo lo que se haga dentro del módulo
  modifican los respectivos objetos del programa principal.
*/

/* Definir todos los módulos (subalgoritmos) que usaré en un mismo proc */
proc iml;

    reset storage = subalg.ejemplos;

    * Ej 2. Función factorial;
    start factorial(n);
        fact = 1;
        do i = 1 to n;
            fact = fact * i;
        end;
        return fact;
    finish factorial;

    * Ej 3. Función maximo;
    start maximo(num1, num2);
        if num1 >= num2 then return num1;
        else return num2;
    finish maximo;

    * Ej 4. Procedimiento colocarTitulo;
    start colocarTitulo(titulo);
        print "================================";
        print titulo;
        print "================================";
    finish colocarTitulo;

    * Ej 4. Procedimiento colocarLinea;
    start colocarLinea;
        print "________________________________";
    finish colocarLinea;

    * Ej 5. Procedimiento proced1;
```

```
    start proced1(n1, n2, n3, n4);
        n3 = n1 + n2;
        n4 = n2 - 1;
    finish proced1;

    * Ej 6. Procedimiento proced1;
    start proced2(n1, n2);
        n1 = n1 + n2;
        n2 = n2 - 1;
    finish proced2;

    * Ej 7. Pasaje por referencia en una función;
    start fun(x, y);
        x = x + 1;
        return x + y;
    finish fun;

    * Ej 8. Pasaje por referencia en un procedimiento;
    start miProc(x, y);
        x = x * 2;
        y = x - y;
    finish miProc;

    * Ej 9. Variables globales;
    start fcn(l) global(z);
        z = 5;
        x = 7;
        return l + 4;
    finish fcn;

    store module = _all_;
quit;

/* Ej 1. Cálculo de números combinatorios (sin subalgoritmos) */
proc iml;
    n = 10;
    k = 2;

    fact1 = 1;
    do i = 1 to n;
        fact1 = fact1 * i;
    end;

    fact2 = 1;
    do i = 1 to n - k;
        fact2 = fact2 * i;
```

```
    end;

    fact3 = 1;
    do i = 1 to k;
        fact3 = fact3 * i;
    end;

    comb = fact1 / (fact2 * fact3);
    print "El nro combinatorio de " n " tomado de " k " es " comb;
quit;

/* Ej 2. Cálculo de números combinatorios (con subalgoritmos) */
proc iml;
    reset storage = subalg.ejemplos;
    load module = factorial; * Esta línea no es necesaria;
    n = 10;
    k = 2;
    comb = factorial(n) / (factorial(n - k) * factorial(k));
    print "El nro combinatorio de " n " tomado de " k " es " comb;
quit;

/* Ej 3. Hallar el máximo entre dos valores */
proc iml;
    reset storage = subalg.ejemplos;
    a = 5;
    b = 10;
    c = 2;
    d = -1;
    print "El maximo entre " a " y " b " es " (maximo(a, b));
    print "El maximo entre " b " y " c " es " (maximo(b, c));
    print "El maximo entre " c " y " c " es " (maximo(c, c));
    print "El maximo entre " d " y " c " es " (maximo(d, c));
quit;

/* Ej 4. Procedimientos para colocar títulos o líneas */
* Se ve mejor en el output (sólo caracteres) que en el Results Viewer (ya tiene formato);
proc iml;
    reset storage = subalg.ejemplos;

    * Cualquier cosa para mostrar;
    a = shape(1:6, 2, 3);
    b = shape(7:10, 2, 2);

    run colocarTitulo("Matrices");
    print a;
    run colocarLinea;
```

```
    print b;
    run colocarLinea;
    print a b;
    run colocarTitulo("Matrices traspuestas");
    print (a`);
    run colocarLinea;
    print (b`);
quit;

/* Ej 5. Procedimiento proced1 */
proc iml;
    reset storage = subalg.ejemplos;
    a = 5;
    b = 2;
    run proced1(a, b, c, d);
    print a b c d;
quit;

/* Ej 6. Procedimiento proced2 */
proc iml;
    reset storage = subalg.ejemplos;
    a = 5;
    b = 2;
    run proced2(a, b);
    print a b;
quit;

/* Ej 7. Pasaje por referencia en una función */
proc iml;
    reset storage = subalg.ejemplos;
    a = 3;
    b = 5;
    c = fun(a, b - a);
    print a b c;
quit;

/* Ej 8. Pasaje por referencia en un procedimiento */
proc iml;
    reset storage = subalg.ejemplos;
    a = 8;
    b = 4;
    run miProc(a, b);
    print a b ;
quit;

/* Ej 9. Variables globales */
```

```
proc iml;
    reset storage = subalg.ejemplos;
    x = 2;
    z = 10;
    y = fcn(x);
    print x z y;
quit;
```