El arte de programar en R: un lenguaje para la estadística

Julio Sergio Santana
Efraín Mateos Farfán

5 de junio de 2014

Capítulo 1

Introducción

1.1. El arte de programar en R

Un artista, típicamente un pintor, tiene en sus manos un conjunto de recursos artísticos: materiales y herramentales, que al combinarlos de acuerdo con su sensibilidad y habilidades, transforma en una obra de arte, estéticamente atractiva o repulsiva a sus destinatarios. Aunque escencialmente tecnológica, en el caso de la programación, el programador o programadora juega ese mismo papel: los distintos ambientes de programación ponen delante de él/ella un conjunto de recursos que de acuerdo con sus conocimientos, habilidades y sensibilidades, combinará para producir obras: programas y sistemas, que, en la superficie, serán funcionales o no funcionales; pero que, a un nivel más profundo, podrían también ser juzgadas como estéticamente atractivas o repulsivas. Uno de los factores que más decisivamente influyen en el *cómo* un creador combina los elementos a su mano, es el gusto y la pasión que imprime en su tarea. Por consiguiente, si de la lectura de este texto se logra encender en ti, querid@ lector@, una pasión que te lleve a producir verdaderas obras de arte, los autores estaremos más que satisfechos por haber cumplido el propósito de esta obra.

1.2. ¿Que es R?

Si este es tu primer acercamiento a R, es muy probable que te cuestiones sobre la ventaja y la utilidad de R sobre otras paqueterías de estadística; como veremos adelante R es más que eso. La intención de este primer capítulo, es responder algunas dudas y animarte a que explores este *software* poderoso, que puede ser aplicado ampliamente en el procesamiento de datos en ciencias.

Empezaremos diciendo que R es un lenguaje de programación interpretado, de distribución libre, bajo Licencia GNU, y se mantiene en un ambiente para el cómputo estadístico y grafico. Este *software* corre en distintas plataformas Linux, Windows, MacOS, e incluso en PlayStation 3. El término ambiente

pretende caracterizarlo como un sistema totalmente planificado y coherente, en lugar de una acumulación gradual de herramientas muy específicas y poco flexibles, como suele ser con otro *software* de análisis de datos. El hecho que R sea un lenguaje y un sistema, es porque forma parte de la filosofía de creación[1], como lo explica John Chambers, cito: "*Buscamos que los usuarios puedan iniciar en un entorno interactivo, en el que no se vean, concientemente, a ellos mismos como programadores. Conforme sus necesidades sean más claras y su complejidad se incremente, deberían gradualmente poder profundizar en la programación, es cuando los aspectos del lenguaje y el sistema se vuelven más importantes*". Por esta razón, en lugar de pensar de R como un sistema estadístico, es preferible verlo como un ambiente en el que se aplican técnicas estadísticas. Por ejemplo, en este libro nos inclinaremos hacia el lado de la programación (lenguaje) más que tocar los aspectos estadísticos. Esto con la finalidad de ampliar la gamma de aplicaciones en el tratamiento de datos.

1.3. Historia de R y S

R fue creado en 1992 en Nueva Zelanda por Ross Ihaka y Robert Gentleman. La intención inicial con R, era hacer un lenguaje didáctico, para ser utilizado en el curso de Introducción a la Estadística de la Universidad de Nueva Zelanda. Para ello decidieron adoptar la sintáxis del lenguaje S desarrollado por Bell Laboratories. Como consecuencia, la sintáxis es similar al lenguaje S, pero la semántica que aparentemente es parecida a la de S, en realidad es sensiblemente diferente, sobre todo en los detalles un poco más profundos de la programación.

A modo de broma Ross y Robert, comienzan a llamar "R" al lenguaje que implementaron, por las iniciales de sus nombres, y desde entonces así se le conoce en la muy extendida comunidad amante de dicho lenguaje. Debido a que R es una evolución de S, a continuación daremos una breve reseña histórica de este lenguaje, para entender los fundamentos y alcances de R.

S es un lenguaje que fue desarrollado por John Chambers y colaboradores en Laboratorios Bell (AT&T), actualmente Lucent Tecnologies, en 1976. Este lenguaje, originalmente fue codificado e implementado como unas bibliotecas de fortran. Por razones de eficiencia, en 1988 S fue reescrito en lenguaje C, dando origen al sistema estadístico S, Versión 3. Con la finalidad de impulsar comercialmente a S, Bell Laboratories dio a StatSci (ahora Insightful Corporation) en 1993, una licencia exclusiva para desarrollar y vender el lenguaje S. En 1998, S ganó el premio de la *Association for Computing Machinery* a los Sistemas de *Software*, y se liberó la versión 4, la cual es prácticamente la versión actual.

El éxito de S fue tal que, en 2004 *Insghtful* decide comprar el lenguaje a *Lucent* (*Bell Laboratories*) por la suma de 2 millones de dólares, convirtiéndose hasta la fecha en el dueño. Desde entonces, *Insightful* vende su implementación del lenguaje S bajo el nombre de S-PLUS, donde le añade un ambiente gráfico

[1]Desde la codificación del lenguaje S, lenguaje progenitor de R, como se verá en la sección siguiente.

amigable. En el año 2008, TIBCO compra *Insightful* por 25 millones de dólares y se continúa vendiendo S-PLUS, sin modificaciones. R, que define su sintaxis a partir de esa versión de S, no ha sufrido en lo fundamental ningún cambio dramático desde 1998.

Regresando a R, luego de la creación de R (en 1992) dan un primer anuncio al público del software R en 1993. En el año de 1995 Martin Mächler, de la Escuela Politécnica Federal de Zúrich, convence a Ross y Robert a usar la Licencia GNU para hacer a R un software libre. A partir de 1997, R forma parte del proyecto GNU.

En 1996 se crea una lista pública de correos, sin embargo debido al gran éxito de R, los creadores fueron rebasados por la continua llegada de correos. Por lo que tuvieron que crear, en 1997, 2 listas de correos R-help y R-devel, que son las que actualmente fucionan. Además se consolida el grupo núcleo de R,donde se involucran personas asociadas con S-PLUS, con la finalidad de administrar el código fuente de R.

Fue hasta febrero de 29 del 2000, que se considera al software completo y lo suficientemente estable, para liberar la versión 1.0.

1.4. Algunas características importantes de R

El sistema R esta dividido en dos partes conceptuales: 1) El sistema base de R, que es el que puedes bajar de CRAN[2]; y, 2) en todo lo demás. La funcionalidad de R consta de paquetes modulares. El sistema base de R contiene el paquete básico que se requiere para su ejecución y la mayoría de las funciones fundamentales. Los otros paquetes contenidos en la "base" del sistema incluye a ***utils***, ***stats, datasets, graphics, grDevices, grid, tools, parallel, compiler, splines, tcltk, stats4***.

La capacidad de graficos de R es muy sofisticada y mejor que la de la mayoria de los paquetes estadísticos. R cuenta con varios paquetes gráficos especializados, por ejemplo, hay paquetería para graficar, crear y manejar los *shapefiles*[3], para hacer contornos sobre mapas en distintas proyecciones, graficado de vectores, contornos, etc. También existen paqueterías que permiten manipular y crear datos en distintos formatos como netCDF, Matlab, Excel entre otros. Cabe señalar que, además del paquete base de R, existen más de 4000 paquetes en CRAN (http://cran.r-project.org), que han sido desarrollados por usuarios y programadores alrededor del mundo, esto sin contar los paquetes disponibles en redes personales.

R es muy útil para el trabajo interactivo, pero también es un poderoso lenguaje de programación para el desarrollo de nuevas herramientas, por ejemplo rclimdex, cliMTA-R, etc. Otra ventaja muy importante es que tiene una comunidad muy activa, por lo que, haciendo las preguntas correctas rápidamente

[2]Por sus siglas en inglés: *The Comprehensive R Archive Network*. Su página *Web* es: http://cran.r-project.org/.

[3]Formato común de sistemas de información geográfica (GIS), introducido por la compañía ESRI en su sistema de *software* ArcGIS.

encontrarás la solución a los problemas que se te presenten en el ámbito de la programación con R. Estas características han promovido que el número de sus usuarios en el área de las ciencias se incremente enormemente.

Al ser *software* libre lo hace un lenguaje atractivo, debido a que no hay que preocuparse por licencias y cuenta con la libertad que garantiza GNU. Es decir con R se tiene la libertad de: 1) correrlo para cualquier propósito, 2) estudiar como trabaja el programa y adaptarlo a sus necesidades, pues se tiene acceso al código fuente, 3) redistribuir copias, y 4) mejorar el programa y liberar sus mejoras al público en general.

Es importante mencionar que, debido a su estructura, R consume mucho recurso de memoria, por lo tanto si se utilizan datos de tamaño enorme, el programa se alentaría o, en el peor de los casos, no podría procesarlos. En la mayoría de los casos, sin embargo, los problemas que pudieran surgir con referencia a la lentitud en la ejecución del código, tienen solución, principalmente teniendo cuidado de vectorizar el código; ya que esto permitiría particionarlo y aprovechar en procesamiento paralelo en equipos con multi núcleos.

1.5. Ayuda en R

R cuenta con una muy buena ayuda en el uso de funciones de manera muy similar al *man* de UNIX. para obtener información de cualquier función en específico, por ejemplo *lm*, el comando es:

```
help(lm)

# Una forma abreviada sería

?lm # -- comentario
```

El código anterior, muestra dos formas de invocar la ayuda en el intérprete de R, la función `help()` y el operador `'?'`. En ambos casos el resultado es el mismo. Aparte de esto, el lenguaje omite interpretar, en un renglón, cualquier texto que siga al símbolo `'#'`; esta es la provisión del lenguaje para incorporar comentarios en el código.

Cuando se desea información sobre caracteres especiales de R, el argumento se debe encerrar entre comillas sencillas o dobles, con la finalidad que lo identifique como una cadena de caracteres. Esto también es necesario hacer para unas cuantas palabras con significado sintáctico incluyendo al `if`, `for`, y `function`. Por ejemplo:

```
help("[[")

help('if')
```

Por otra parte, el sistema puede mostrar un listado de contenidos acerca de algún tópico cualquiera invocando la función `help.search()`, que abreviadamente se invoca con `??`. Por ejemplo, si se desea saber acerca del tópico *split*, lo que puede incluir, funciones, paquetes, variables, etc., se hace de la siguiente manera:

```
help.search("split")

# O abreviadamente:
??"split"
```

Además de esto, existen foros muy activos en diversos temas de R (*Mailing Lists*), donde seguramente podrás encontrar las respuestas apropiadas. La dirección de esos foros la puedes se encontrar en http://www.r-project.org[4].

R en las ciencias

[4]Una interfaz en la *Web* al grupo básico de R, conocido como *R-help*, se puede encontrar en http://dir.gmane.org/gmane.comp.lang.r.general.

Capítulo 2

Los datos y sus tipos

Todas las cosas que manipula R se llaman objetos. En general, éstos se construyen a partir de objetos más simples. De esta manera, se llega a los objetos más simples que son de cinco clases a las que se denomina *atómicas* y que son las siguientes:

- `character` (cadenas de caracteres)
- `numeric` (números reales)
- `integer` (números enteros)
- `complex` (números complejos)
- `logical` (lógicos o booleanos, que sólo toman los valores True o False)

En el lenguaje, sin embargo, cada uno de estas clases de datos no se encuentran ni se manejan de manera aislada, sino encapsulados dentro de la clase de objeto más básica del lenguaje: el `vector`. Un `vector` puede contener cero o más objetos, pero todos de la misma clase. En contraste, la clase denominada `list`, permite componer objetos también como una secuencia de otros objetos, pero, a diferencia del `vector`, cada uno de sus componentes puede ser de una clase distinta.

2.1. Los datos numéricos

Probablemente el principal uso de R es la manipulación de datos numéricos. El lenguaje agrupa estos datos en tres categorías, a saber: `numeric, integer` y `complex`, pero cuando se introduce algo que puede interpretar como un número, su inclinación es tratarlo como un dato de tipo `numeric`, es decir, un número de tipo real, a no ser que explícitamente se indique otra cosa. Veamos algunos ejemplos:

```
x <- 2  # Se asigna el valor 2 a x
print(x)  # Se imprime el valor de x

## [1] 2

class(x)  # Muestra cual es la clase de x

## [1] "numeric"

x <- 6/2  # Se asigna el valor de la operacion dividir 6/2 a x
print(x)

## [1] 3

class(x)

## [1] "numeric"
```

Aparentemente las dos asignaciones que se hacen, mediante el operador de asignación, `<-`, a la *variable* `x`, es de los enteros 2 y 3 respectivamente. Sin embargo, al preguntar, mediante la *función* `class()`, cuál es la clase de `x`, la respuesta es `numeric`, esto es, un número real. Para asignar explícitamente un entero, `integer`, a una variable, se agrega la letra L al final del número, como sigue:

```
x <- 23L; print(x)

## [1] 23

class(x)

## [1] "integer"
```

Aquí la variable `x` tendrá como valor el entero 23. Como una nota adicional del lenguaje, nótese que se han escrito dos expresiones de R en un mismo renglón. En este caso, las expresiones se separan mediante ';'.

Para lograr que una expresión, como la operación de división `6/2`, arroje como resultado un entero, se tiene que hacer una *conversión*; ello se logra mediante la función `as.integer`, como sigue:

```
x <- as.integer(6/2); print(x)

## [1] 3

class(x)

## [1] "integer"
```

Por su parte, los números complejos, `complex` en el lenguaje, tienen una sintaxis muy particular; misma que se tiene que emplear para indicar explícitamente que un número introducido corresponde a ese tipo:

```
x <- 21 + 2i
y <- 2i + 21 # El mismo valor que x
z <- -1 + 0i # Corresponde a -1
tt <- sqrt(z) # raiz cuadrada de -1
print(x); print(y); print(z); print(tt)

## [1] 21+2i
## [1] 21+2i
## [1] -1+0i
## [1] 0+1i

class(tt)

## [1] "complex"
```

En los ejemplos anteriores a la variable `tt` se le asigna el resultado de una función, `sqrt()`, que es la raíz cuadrada de el número `-1`. Nótese que ésta es la forma correcta de calcular esa raíz, por ejemplo, `sqrt(-1)`, hubiera arrojado como resultado un error.

También, existe un valor numérico especial, `Inf`, que representa el infinito y que puede resultar en algunas expresiones, por ejemplo:

```
x <- 1/0  # Division por cero
x

## [1] Inf

# Tambien dividir un número por Inf da cero:
y <- 1/Inf
y

## [1] 0
```

Finalmente, algunas operaciones pueden resultar en algo que no es un número, esto se representa por el valor `NaN`. Veamos un ejemplo:

```
x <- 0/0
x

## [1] NaN
```

2.2. Vectores

Se ha dicho con anterioridad que las clases atómicas de datos no se manejan de manera individual. En efecto, en todos los ejemplos anteriores, el lenguaje ha creado implícitamente vectores de longitud 1, y son esos los que se han asignado a las variables. Tomemos el caso más sencillo:

```
x <- 2  # Se asigna el valor 2 a x
print(x)  # Se imprime el valor de x

## [1] 2
```

Aquí, la impresión del valor de `x` tiene una forma muy particular: "`[1] 2`". El '`[1]`' que precede al valor, indica que se trata del primer elemento y único, en este caso, del vector que se muestra.

Hay diversas maneras de crear vectores de otras longitudes, que, como se ha dicho antes, son secuencias de objetos de la misma clase atómica. En las siguientes secciones se verán algunos casos.

2.2.1. El uso de la función `c()` para crear vectores

La primer manera de crear vectores es a partir de los elementos individuales que compondrán el vector. Para esto se utiliza la función `c()` como se muestra a continuación.

```
c(4,2,-8)  # Creacion de un vector sin asignarlo a una variable

## [1]  4  2 -8

## -----------------
## Distintas formas de asignar un vector a una variable
u <- c(4,2,-8) # Usando el operador <-
c(4, 2, -8) -> v # Usando el operador ->
# Usando la funcion assign:
assign("w", c(4, 2, -8))
p = c(4, 2, -8) # Usando el operador =
print(u); print(v); print(w); print(p)

## [1]  4  2 -8
## [1]  4  2 -8
## [1]  4  2 -8
## [1]  4  2 -8
```

La función `c()` sirve para concatenar varios elementos del mismo tipo. En todos los ejemplos mostrados, la impresión del vector se hace en un renglón que comienza con el símbolo '`[1]`', indicando con ello que el primer elemento del renglón corresponde al primer elemento del vector.

Un caso muy particular de asignación, es el de la función `assign()`. A diferencia de los otros casos vistos en el ejemplo anterior, el nombre de la variable aparece entre comillas.

Más adelante, en la página 12, se verá como la función `c()` también se puede utilizar para la creación de vectores a partir de otros vectores.

2.2.2. Creación de vectores a partir de archivos de texto - la función `scan()`

Otra manera de crear un vector es a partir de un archivo de texto. Sea, por ejemplo, el caso del archivo UnVec.txt, que se contiene la siguiente información:

```
  12 15.5 3.1
  -2.2 0 0.0007
```

Supóngase ahora que a partir de esos datos se quiere crear un vector. Para eso se usa la función `scan()`, como se muestra a continuación:

```
vec <- scan("UnVec.txt")
print(vec)

## [1] 12.0000 15.5000  3.1000 -2.2000  0.0000  0.0007
```

Desde luego que hay otras funciones para lectura de archivos, más complejas, pero baste por el momento con el uso de esta función, tal como se muestra, para permitir la creación de un vector a partir de los datos contenidos en un archivo de texto. Por el ahora, la única nota adicional es que la función `scan()` ofrece la posibilidad de indicarle explícitamente el tipo de vector que se quiere crear. Así por ejemplo, la creación de un vector de enteros se hace de la siguiente manera:

```
vec <- scan("IntVec.txt", integer())
print(vec); class(vec) # El vector y su clase

## [1]   4   3  -2   1   0 200  -8  20
## [1] "integer"
```

Por supuesto que en este caso, se debe prever que el archivo leído contenga datos que puedan ser interpretados como números enteros.

La función inversa, en este caso, de la función scan(), es la función write. Así, un vector cualquiera fácilmente se puede escribir en un archivo de texto, como se muestra a continuación:

```
vv <- c(5, 6.6, -7.7)
write(vv, "OtroArchivo.txt")
```

```
# Ahora recuperemos el contenido del archivo
v1 <- scan("OtroArchivo.txt")
v1

## [1]  5.0  6.6 -7.7
```

2.2.3. Creación de vectores a partir de secuencias y otros patrones

Un vector, inicializado en ceros, o `FALSE`, y de longitud determinada, se puede crear con la función `vector()`. Es esta misma función la que permite crear vectores sin elementos. En seguida se muestran algunos ejemplos:

```
v <- vector("integer", 0)
v  # Un vector de enteros sin elementos

## integer(0)

w <- vector("numeric", 3)
w  # Un vector de tres ceros

## [1] 0 0 0

u <- vector("logical", 5)
u  # Un vector de 5 FALSE

## [1] FALSE FALSE FALSE FALSE FALSE
```

El operador `':'` permite generar un vector entero a partir de una secuencia creciente o decreciente de enteros, cuyos extremos se indican, tal como se muestra en seguida:

```
1:3

## [1] 1 2 3

v <- 40:13
print(v)

##  [1] 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23
## [19] 22 21 20 19 18 17 16 15 14 13

class(v)  # El vector y su clase

## [1] "integer"
```

Nótese que el desplegado o impresión del vector `v`, se ha tenido que hacer en dos renglones. Cada uno de esos renglones comienza, indicando entre corchetes `[ ]`, el índice del primer elemento en el renglón. Por otra parte, el operador `':'` es un caso particular de la función `seq()` que permite generar una mayor variedad de secuencias numéricas. Veamos aquí algunos ejemplos:

```
v <- seq(from = 5, to = 15, by = 2)
print(v)  # secuencia desde 5 hasta 15 de 2 en 2

## [1]  5  7  9 11 13 15
```

Debe notarse aquí, no obstante, que la clase del resultado de esta secuencia es `numeric` y no `integer`; esto es, el vector resultante es de números reales, que puede, a conveniencia, ser convertido a enteros, mediante la función `as.integer()`, como se vio anteriormente, en la página 7.

```
class(v)

## [1] "numeric"
```

La función `seq()` tiene varios argumentos más cuya documentación se puede consultar mediante `?seq` o `help('seq')` en el intérprete de R. En seguida se muestra sólo otra forma bastante común de utilizar esta función, que tiene que ver con la producción de un vector o una secuencia de una longitud determinada.

```
v <- seq(from = 4, by = 2, length.out = 8)
print(v)  # secuencia de 8 números iniciando desde  4 y de 2 en 2

## [1]  4  6  8 10 12 14 16 18
```

Algunas veces es necesario repetir una secuencia de números varias veces para generar un vector deseado. La función `rep()` sirve para ese propósito. Supóngase, por ejemplo, que se desea crear un vector con la repetición de la secuencia 4, 8, -3, cinco veces. Eso se logra como se muestra a continuación:

```
v <- c(4, 8, -3)
w <- rep(v, times = 5)
print(w)

##  [1]  4  8 -3  4  8 -3  4  8 -3  4  8 -3  4  8 -3
```

Finalmente, aveces se requiere construir un vector a partir de dos o más vectores ya existentes. La forma simple de lograr esto es con la función `c()` como se muestra a continuación:

```
u <- c(3, 4, 5)
v <- c(5, 4, 3)
w <- c(u, v)
print(w)  # La concatenacion de u y v

## [1] 3 4 5 5 4 3
```

2.2.4. Acceso a los elementos individuales de un vector

Aunque este tema está comprendido dentro de la selección de subconjuntos o porciones, que se verá más adelante en el capítulo 3, se dará aquí un adelanto para permitir operar con los elementos individuales de los vectores. Dentro de un vector, sus elementos se pueden identificar mediante un *índice* entero, que en el caso de este lenguaje empieza con el 1. Así, por ejemplo:

```
v <- c(8, 7, -3, 2, 182)
v[5] # El quinto elemento

## [1] 182

print(v[1]); print(v[3])

## [1] 8
## [1] -3

v[4]+v[2] # La suma del cuarto y segundo elementos de v

## [1] 9
```

Primeramente se ha accedido al quinto elemento, mediante `v[5]` ; si el intérprete de R se está usando interactivamente, el valor de ese elemento, 182 , se imprime implícitamente. Luego se manda imprimir explícitamente, los elementos 1 y 3 del vector. Finalmente, se suman los elementos 4 y 2 del vector; el resultado de esa operación se imprime implícitamente, es decir, de manera automática, si la operación se solicita al intérprete en su modo interactivo.

El acceso a los elementos individuales de un vector no solamente es para *consulta* o *lectura*, sino también para su *modificación* o *escritura*. Por ejemplo:

```
v[1] <- v[2] - v[5]
v  # Note que el resultado de la operacion se ha guardado en v[1]

## [1] -175    7   -3    2  182
```

Esta misma operación puede hacer crecer un vector. Por ejemplo, el vector `v` tiene 5 elementos. Si se asigna un valor al elemento 8, el vector *crecerá* hasta esa longitud, de la manera siguiente:

```
v[8] <- 213
v  # v tiene ahora 8 elementos con espacios vacios: NA

## [1] -175    7   -3    2  182   NA   NA  213
```

La nota aquí es que para aumentar el vector a esa longitud se tuvieron que introducir elementos ausentes o vacíos que se indican con el valor `NA` (del inglés: *Not Available*) en los espacios correspondientes.

Otra característica interesante de este lenguaje, es que permite dar nombre y acceder por medio de ese nombre a los elementos individuales de un vector. Supóngase por ejemplo que se tiene el registro de cantidades de ciertas frutas en un vector:

```
frutas <- c(15, 100, 2, 30)
frutas

## [1]  15 100   2  30
```

Supóngase ahora que se quiere asociar esos valores con el nombre de la fruta correspondiente:

```
names(frutas) <- c("naranja", "pera", "manzana", "durazno")
```

Si ahora se manda desplegar el vector:

```
frutas

## naranja    pera manzana durazno
##      15     100       2      30
```

Otra manera más directa de nombrar los elementos de un vector, es en el momento mismo de la creación con la función `c()`, con una sintaxis semejante a la siguiente:

```
frutas <- c(naranja = 15, pera = 100, manzana = 2, durazno = 30)
```

Además se puede acceder a los elementos individuales del vector mediante su nombre:

```
frutas["durazno"]

## durazno
##      30

frutas["manzana"] <- 8
frutas
```

```
## naranja    pera manzana durazno
##      15     100       8      30

# El acceso a traves de indices se sigue permitiendo:
frutas[2]

## pera
##  100
```

2.2.5. Operaciones sencillas con vectores

Las operaciones aritméticas más comunes están definidas para vectores: la suma, la resta, la división y la exponenciación, todas ellas se definen elemento a elemento entre dos vectores. Por ejemplo:

```
v <- 2 + 3  # Resulta en un vector de longitud 1
v

## [1] 5

v <- c(2, 3) - c(5, 1)  # Resulta en un vector de longitud 2
v

## [1] -3  2

v <- c(2, 3, 4) * c(2, 1, 3)  # Resulta en un vector de longitud 3
v

## [1]  4  3 12

v <- c(2, 3, 4)^(3:1)  # Eleva a potencias 3,2,1
v

## [1] 8 9 4
```

En todos los casos, la operación indicada se aplica elemento a elemento entre los dos vectores operandos. En el último ejemplo, debido al orden de precedencia de aplicación de los operadores, es necesario encerrar entre paréntesis la expresión `3:1`.

En muchas ocasiones es necesario saber la longitud de una vector. La función `length()` aplicada a un vector regresa precisamente ese valor:

```
u <- 2:33
v <- c(4, 5, 6)
w <- c(u, v)
w
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
## [19] 20 21 22 23 24 25 26 27 28 29 30 31 32 33  4  5  6

length(w)

## [1] 35
```

Aprovecharemos el vector `w`, creado en el ejemplo anterior, para ilustrar también el uso de las operaciones lógicas. ¿Qué pasa si probamos este vector para saber cuáles de sus elementos son menores o iguales a 10?

```
w <= 10  # Prueba elementos menores o iguales a 10

##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [10] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [19] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [28] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

El resultado es un vector de lógicos, de la misma longitud que el original y *paralelo* a ese, en el que se indica, elemento a elemento cuál es el resultado de la prueba lógica: *"menor o igual que diez"*, en este caso. Otros operadores lógicos son: <, >, >=, ==, y !=.

En el asunto de las operaciones aritméticas que se han ilustrado anteriormente, surge una pregunta: ¿qué pasa cuando los vectores operandos no son de la misma longitud? En esos casos, el intérprete del lenguaje procede a completar la operación *reciclando* los elementos del operador de menor longitud. Así, por ejemplo:

```
v <- c(4, 5, 6, 7, 8, 9, 10) * c(1, 2)

## Warning: longitud de objeto mayor no es múltiplo de la longitud
de uno menor

v

## [1]  4 10  6 14  8 18 10
```

es lo mismo que:

```
v <- c(4, 5, 6, 7, 8, 9, 10) * c(1, 2, 1, 2, 1, 2, 1)
v

## [1]  4 10  6 14  8 18 10
```

Notemos, sin embargo, que en el primer caso el sistema ha arrojado un mensaje de advertencia, *Warning*, indicando la diferencia en las longitudes de los operandos. La eliminación de estos mensajes se hace por medio de la función `options()`, como sigue:

```
options(warn = -1)
v <- c(4, 5, 6, 7, 8, 9, 10) * c(1, 2)
v

## [1]  4 10  6 14  8 18 10
```

Es esta funcionalidad la que permite hacer de manera muy simple algunas operaciones vectoriales, como por ejemplo:

```
v <- c(2, -3, 4)
w <- 2 * (v^2)  # Dos veces el cuadrado de v
w

## [1]  8 18 32
```

Además, algunas funciones pueden recibir como argumento un vector y producir a su salida un vector de la misma longitud que el de entrada. Tal es el caso de las funciones trigonométricas como `sin()`, `cos()`, y la raíz cuadrada: `sqrt()`. Por ejemplo:

```
# Se desea la raiz cuadrada de los siguientes valores:
v <- c(9, 8, 31)
sqrt(v)

## [1] 3.000 2.828 5.568

# El sin de 30, 45 y 60 grados: Primero se hace la conversion
# a radianes:
angulos <- c(30, 45, 60) * (pi/180)
angulos  # En radianes

## [1] 0.5236 0.7854 1.0472

senos <- sin(angulos)
senos

## [1] 0.5000 0.7071 0.8660
```

Para ilustrar la utilidad de estos conceptos en los siguientes párrafos se da un ejemplo de aplicación.

Ejemplo de aplicación

De un edificio, a una altura de 15 m, se ha lanzado con un ángulo de 50 grados, un proyectil a una velocidad de 7 m/s. ¿Cuáles serán las alturas (coordenadas y) del proyectil a cada 0.5 m de distancia horizontal desde donde se lanzó y hasta los 11 m?

Las ecuaciones que gobiernan este fenómeno son las siguientes:

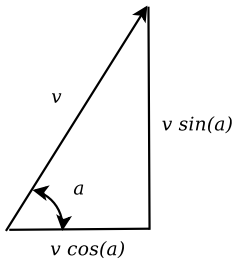

Figura 2.1: Las componentes de la velocidad

$$x = v_{0x}t + x_0$$

$$y = -\frac{1}{2}gt^2 + v_{0y}t + y_0$$

Aquí, g es la aceleración de la gravedad, el parámetro t se refiere al tiempo, y la velocidad está descompuesta en sus componentes: v_{0x} y v_{0y}. Tal como se muestra en la Fig. 2.1, éstas se pueden obtener a partir de la velocidad inicial y el ángulo, usando las funciones trigonométricas `sin()` y `cos()`, y considerando que en R, los argumentos de esas funciones deben estar dados en radianes, y por tanto el ángulo debe convertirse a esa unidad. Así, los datos de partida son como sigue:

```
g <- 9.81  # aceleracion gravedad
x0 <- 0  # x inicial
y0 <- 15  # y inicial
vi <- 7  # velocidad inicial
alphaD <- 50  # angulo-grados
```

y para encontrar las componentes de la velocidad:

```
# Se convierte a radianes
alpha <- (pi/180) * alphaD  # angulo-radianes
vox <- vi * cos(alpha)  # componente x de velocidad inicial
vox

## [1] 4.5

voy <- vi * sin(alpha)  # componente y de velocidad inicial
voy

## [1] 5.362
```

Con esto es suficiente para proceder con el problema. Primeramente obtenemos las x para las que se desea hacer el cálculo, como sigue:

```
# desde 0 hasta 11 de 0.5 en 0.5:
las.x <- seq(from = 0, to = 11, by = 0.5)
```

En este ejemplo, la secuencia de valores de x se ha guardado en una variable de nombre "`las.x`". En este lenguaje, en los nombres de las variables, el punto (.), así como el guión bajo (_), se pueden utilizar simplemente como separadores, para darles mayor claridad.

Nótese que en las fórmulas que gobiernan el fenómeno, dadas anteriormente, no se tiene y en función de x, sino que las dos coordenadas dependen del parámetro t, esto es, del tiempo. Para resolver este asunto simplemente se despeja en parámetro t, en la ecuación de x, y obtenemos:

$$t = (x - x_0)/v_{0x}$$

Así, obtenemos los valores de t correspondientes a las x, usando esta fórmula:

```
las.t <- (las.x - x0)/vox
```

Finalmente, encontramos las y correspondientes a las t, justamente encontradas, aplicando la fórmula para y:

```
las.y <- -(g/2) * las.t^2 + voy * las.t + y0
# Los resultados:
las.x

##  [1]  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0
## [12]  5.5  6.0  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0 10.5
## [23] 11.0

las.y

##  [1] 15.0000 15.5353 15.9495 16.2425 16.4144 16.4652 16.3948
##  [8] 16.2033 15.8906 15.4568 14.9019 14.2258 13.4286 12.5103
## [15] 11.4708 10.3102  9.0285  7.6256  6.1015  4.4564  2.6901
## [22]  0.8026 -1.2059
```

Se han encontrado los valores buscados, y en la Fig. 2.2 se muestra un gráfico con la trayectoria del proyectil.

2.2.6. Otras clases de datos basadas en vectores

Los vectores sirven como base para la definición de otras clases de datos, a saber: las matrices y los arreglos. En la sección siguiente se da una breve introducción al tema de las matrices. El tema de los arreglos, sin embargo, se abordará en un capítulo posterior.

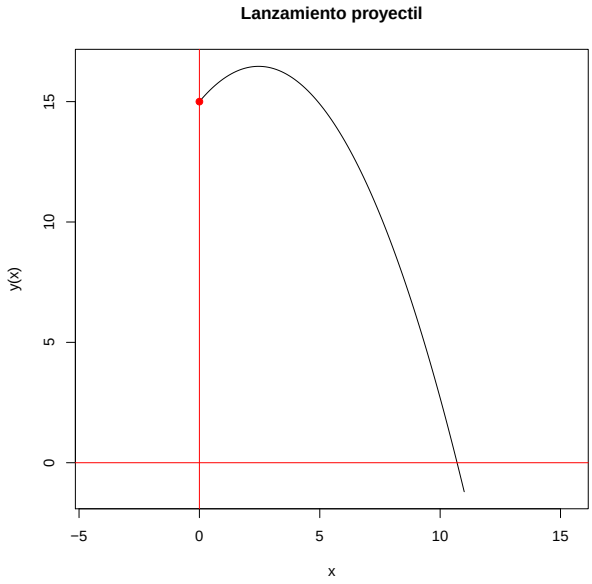

Figura 2.2: Gráfico de la trayectoria del proyectil lanzado desde una altura de 15 m.

2.3. Matrices

Desde el punto de vista del lenguaje, una matriz es un vector con un atributo adicional: `dim`. Para el caso de las matrices, este atributo es un vector entero de dos elementos, a saber: el número de renglones y el número de columnas que componen a la matriz.

2.3.1. Construcción de matrices

Una de las formas de construir una matriz es a partir de un vector, como sigue:

```
(m <- 11:30)  # Un vector con 20 numeros

##  [1] 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
## [19] 29 30

# Para convertirla en matriz simplemente se especifica el
# atributo dim
dim(m) <- c(4, 5)  # 4 renglones y 5 columnas
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   11   15   19   23   27
## [2,]   12   16   20   24   28
## [3,]   13   17   21   25   29
## [4,]   14   18   22   26   30

class(m)

## [1] "matrix"
```

Debe notarse que, mediante la construcción mostrada, el armado de la matriz se hace por columnas. Por otra parte, las dimensiones de la matriz pueden cambiarse en cualquier momento, y el acceso a un elemento particular de la matriz se hace ahora mediante dos índices: el renglón y la columna, aunque, el acceso a los elementos de la matriz como un vector, es decir, con un solo índice, sigue siendo posible, como se muestra en seguida:

```
dim(m) <- c(5, 4)  # ahora 5 renglones y 4 columnas
m

##      [,1] [,2] [,3] [,4]
## [1,]   11   16   21   26
## [2,]   12   17   22   27
## [3,]   13   18   23   28
## [4,]   14   19   24   29
## [5,]   15   20   25   30

# Y el elemento en el renglon 3 y columna 2 es:
m[3, 2]

## [1] 18

m[8]  # acceso al mismo elemento, como vector, con un solo indice

## [1] 18
```

Una ventaja del lenguaje es que permite hacer referencia a una columna o a un renglón de la matriz, como si se tratara de un sólo objeto, o sea como un vector. Para ello, se omite alguno de los dos índices en la expresión de acceso a la matriz, como se muestra más adelante. En el ejemplo que se viene examinando, esos vectores estarían compuestos por números enteros, aunque los componentes de una matriz pueden ser también reales (`numeric`) o complejos (`complex`).

```
# El renglon 3 y la columna 2 de la matriz:
m[3, ]
```

```
## [1] 13 18 23 28

m[, 2]

## [1] 16 17 18 19 20

# La clase las columnas o renglones:
class(m[3, ])

## [1] "integer"
```

Las matrices también se pueden crear de manera flexible por medio de la función primitiva `matrix()`, que permite alterar la secuencia por *default* de armado de la matriz; esto es, ahora, si se quiere, se puede armar la matriz por renglones en vez de columnas:

```
(m <- matrix(11:30, nrow = 5, ncol = 4, byrow = TRUE))

##      [,1] [,2] [,3] [,4]
## [1,]   11   12   13   14
## [2,]   15   16   17   18
## [3,]   19   20   21   22
## [4,]   23   24   25   26
## [5,]   27   28   29   30
```

Adicionalmente, a los renglones y las columnas de una matriz se les pueden asignar nombres, que pueden ser después consultados o usados como índices:

```
rownames(m) <- c("uno", "dos", "tres", "cuatro", "cinco")
colnames(m) <- c("UNO", "DOS", "TRES", "CUATRO")
m

##        UNO DOS TRES CUATRO
## uno     11  12   13     14
## dos     15  16   17     18
## tres    19  20   21     22
## cuatro  23  24   25     26
## cinco   27  28   29     30

# Consulta de los nombres de las columnas
colnames(m)

## [1] "UNO"    "DOS"    "TRES"   "CUATRO"

# Una columna:
m[, "DOS"]

##    uno    dos   tres cuatro  cinco
##     12     16     20     24     28
```

Las funciones `rbind()` y `cbind()`, son otras que se pueden utilizar para construir matrices, dando, ya sea los renglones individuales o las columnas individuales, respectivamente.

```
m1 <- rbind(c(1.5, 3.2, -5.5), c(0, -1.1, 60))
m1

##      [,1] [,2] [,3]
## [1,]  1.5  3.2 -5.5
## [2,]  0.0 -1.1 60.0

class(m1[1, ])  # ahora compuesta de numeros reales

## [1] "numeric"

(m2 <- cbind(c(1.5, 3.2, -5.5), c(0, -1.1, 60)))

##      [,1] [,2]
## [1,]  1.5  0.0
## [2,]  3.2 -1.1
## [3,] -5.5 60.0
```

2.3.2. Acceso a los elementos individuales de una matriz

Como en los casos anteriores, el lenguaje también provee de mecanismos para acceder a los elementos individuales de una matriz. Para ello se emplea el operador `[]`. Supongamos que en la matriz `m`, del ejemplo anterior se quiere tener acceso al elemento que se encuentra en el renglón 2 y en la columna 1 de la matriz. Eso se logra de la siguiente manera:

```
m[2, 1]

## [1] 15
```

Y también se pueden utilizar los nombres de renglón y columna, si es que la matriz los tiene:

```
m["dos", "UNO"]

## [1] 15
```

Otras formas para tener acceso a porciones de la matriz, se verán con detalle más adelante, en el capítulo 3.

2.3.3. Operaciones sencillas con matrices

Todas las operaciones aritméticas válidas para vectores, son validas para las matrices, siempre y cuando, las matrices operando tengan las mismas di-

mensiones y se aplican elemento a elemento, esto es, la operación se aplica entre cada columna, con su correspondiente, como si fueran vectores. (véase la sección correspondiente: 2.2.5). En seguida, se muestra un ejemplo con la multiplicación, que no debe ser confundido con la multiplicación matricial.

```
(m <- matrix(1:15, nrow = 5, ncol = 3))

##      [,1] [,2] [,3]
## [1,]    1    6   11
## [2,]    2    7   12
## [3,]    3    8   13
## [4,]    4    9   14
## [5,]    5   10   15

(mm <- rbind(1:3, 3:1, c(1, 1, 1), c(2, 2, 2), c(3, 3, 3)))

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    3    2    1
## [3,]    1    1    1
## [4,]    2    2    2
## [5,]    3    3    3

m * mm

##      [,1] [,2] [,3]
## [1,]    1   12   33
## [2,]    6   14   12
## [3,]    3    8   13
## [4,]    8   18   28
## [5,]   15   30   45
```

La multiplicación matricial se hace con el operador %*%. Para entender esta operación, pondremos un ejemplo con dos matrices, como sigue:

```
(A <- matrix(1:6, 3, 2))

##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6

(B <- rbind(7:9, 10:12))

##      [,1] [,2] [,3]
## [1,]    7    8    9
## [2,]   10   11   12
```

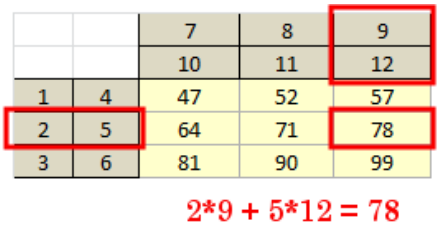

Figura 2.3: La multiplicación matricial

En el ejemplo, la matriz `A` será multiplicada por la matriz `B`, y debe notarse que, en este caso, el número de columnas de la matriz `A`, es igual al número de renglones de la matriz `B`. La multiplicación de estas dos matrices la podemos visualizar en la Fig. 2.3. En esta figura, la matriz `A` se pone a la izquierda y la matriz `B` se pone en la parte superior. Los elementos de la matriz producto, estarán en las intersecciones de un renglón de la matriz `A` con una columna de la matriz `B`, y se calculan como se muestra en el ejemplo: el primer elemento del renglón de `A` por el primer elemento de la columna de `B` más el segundo elemento del renglón de `A` por el segundo elemento de la columna de `B`, etc. Este procedimiento es igual, para dimensiones mayores, siempre y cuando coincida el número de columnas de `A` con el número de renglones de `B`. En R, esta operación se hace así:

```
A %*% B

##      [,1] [,2] [,3]
## [1,]   47   52   57
## [2,]   64   71   78
## [3,]   81   90   99
```

Otra operación muy utilizada e implementada en R como una función, `t()`, es la traspuesta de una matriz. Esta es una operación en la que los renglones se cambian a columnas y viceversa, tal como se muestra en el siguiente ejemplo:

```
# Se usa la misma matriz A del ejemplo anterior:
A

##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6

t(A)

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

Hay otras operaciones matriciales, pero baste con éstas en el presente capítulo, que las otras se introducirán más adelante en el texto.

Ejemplo de aplicación

Las transformaciones lineales se representan por medio de matrices, y su aplicación involucra la multiplicación matricial de la matriz que representa la transformación por el vector o secuencia de vectores que representan el punto o puntos en el espacio que se quieren transformar. Como un ejemplo, la rotación en dos dimensiones es una transformación lineal: si se quiere rotar el punto (x, y) por un ángulo α, la operación está dada por:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

donde el punto (x', y'), es el punto transformado, es decir, al que se ha aplicado la rotación. Si la operación se quiere hacer a una secuencia de puntos que pudieran representar los vértices de alguna figura geométrica, bastará con armar la matriz de puntos correspondiente y aplicar a ella la transformación, de la siguiente manera:

$$\begin{bmatrix} x'_1 & x'_2 & \dots & x'_n \\ y'_1 & y'_2 & \dots & y'_n \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \dots & x_n \\ y_1 & y_2 & \dots & y_n \end{bmatrix}$$

Supóngase ahora, que se tiene un triángulo, cuyos vértices son (1.0, 0.0), (2.0, 1.0), y (1.0, 1.0), y se quieren encontrar los vértices del triángulo resultante de una rotación de 32°. Tómese en cuenta que el lenguaje R, provee de las funciones trigonométricas `sin()`, `cos()`, así como del número `pi`.

```
# Triangulo original:
m <- cbind(c(1, 0), c(2, 1), c(1, 1))
# Se convierte el angulo a radianes
alpha <- 32 * pi/180
# La matriz para esa rotacion es:
tr <- rbind(c(cos(alpha), -sin(alpha)), c(sin(alpha), cos(alpha)))
# El triangulo transformado
mt <- tr %*% m  # multiplicacion matricial
# Los vertices del triangulo transformado
mt
```

```
##        [,1]  [,2]   [,3]
## [1,] 0.8480 1.166 0.3181
## [2,] 0.5299 1.908 1.3780
```

En la Fig. 2.4 se muestran tanto el triangulo original, como el triangulo resultante, en rojo, después de la rotación.

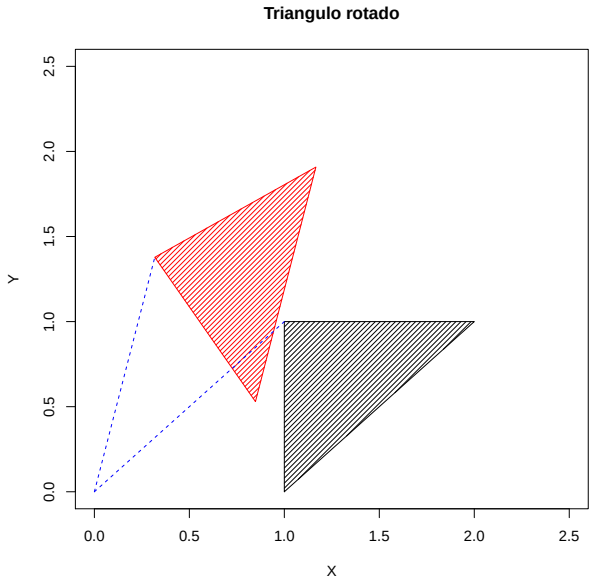

Figura 2.4: Rotación de un triángulo; el triángulo rotado se muestra en rojo

2.4. Factores y vectores de caracteres

Los caracteres, o más apropiadamente, las *cadenas de caracteres*, se utilizan para nombrar cosas u objetos del *mundo*. Igual que en el caso de los números, en R la clase `character` no se refiere a una cadena de caracteres aislada sino a un vector que contiene cero o más cadenas de caracteres. De este modo podríamos tener por ejemplo, una lista (o vector) con los nombres de personas, y otra, paralela a la primera, con sus meses de nacimiento:

```
persona <- c("Hugo", "Paco", "Luis", "Petra", "Maria", "Fulano",
             "Sutano", "Perengano", "Metano", "Etano", "Propano")
mes.nacimiento <- c("Dic", "Feb", "Oct", "Mar", "Feb", "Nov",
                    "Abr", "Dic", "Feb", "Oct", "Dic")
persona
```

```
##  [1] "Hugo"      "Paco"      "Luis"      "Petra"
##  [5] "Maria"     "Fulano"    "Sutano"    "Perengano"
##  [9] "Metano"    "Etano"     "Propano"

mes.nacimiento
```

```
##  [1] "Dic" "Feb" "Oct" "Mar" "Feb" "Nov" "Abr" "Dic" "Feb"
## [10] "Oct" "Dic"
```

Así, si se quiere imprimir el nombre de la persona 7 con su mes de nacimiento se puede hacer con:

```
print(persona[7]); print(mes.nacimiento[7])

## [1] "Sutano"
## [1] "Abr"

# De una manera mas "pulcra":
print( c(persona[7], mes.nacimiento[7]) )

## [1] "Sutano" "Abr"
```

La función `paste()` permite concatenar cadenas de caracteres y por medio de ella se puede dar incluso una mejor apariencia a la salida:

```
paste(persona[7], "nacio en el mes de", mes.nacimiento[7])

## [1] "Sutano nacio en el mes de Abr"
```

2.4.1. Los factores y su estructura

Los dos vectores anteriores pueden considerarse como una *estructura de información*, a la que se puede someter a algún tipo de procesamiento estadístico. El lenguaje tiene muchas herramientas para ese propósito. Considérese, por ejemplo, el problema de determinar la frecuencia de aparición de ciertos meses en el vector `mes.nacimiento`. En este caso, el lenguaje provee de una clase que facilita este tipo de análisis, a saber: la clase `factor`. Para entender esta clase, procedamos primeramente a transformar el vector `mes.nacimiento` a un factor, mediante la función de conversión `as.factor()`, como sigue:

```
Fmes.nacimiento <- as.factor(mes.nacimiento)
Fmes.nacimiento

##  [1] Dic Feb Oct Mar Feb Nov Abr Dic Feb Oct Dic
## Levels: Abr Dic Feb Mar Nov Oct

# y generamos la impresion ahora con el factor:
paste(persona[7], "nacio en el mes de", Fmes.nacimiento[7])

## [1] "Sutano nacio en el mes de Abr"
```

Si se compara la impresión del factor `Fmes.nacimiento` con la del vector `mes.nacimiento`, se podría pensar que "no ha pasado mucho". De hecho, la impresión *bonita* con la función `paste()`, ha resultado igual. Sin embargo, el factor exhibe una estructura adicional denominada `Levels`, en la que se han

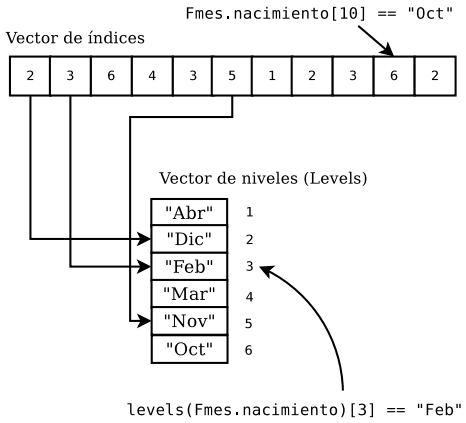

Figura 2.5: Estructura interna de los factores

registrado e identificado los elementos del vector sin repetición; esto es, los nombres únicos de los meses, en este caso. La estructura interna de esta clase se puede descubrir:

```
unclass(Fmes.nacimiento)

##  [1] 2 3 6 4 3 5 1 2 3 6 2
## attr(,"levels")
## [1] "Abr" "Dic" "Feb" "Mar" "Nov" "Oct"
```

Como se puede ver, el núcleo de la clase son dos vectores. El primero, es un vector de índices enteros, que sustituye al vector de caracteres original, y el segundo es un vector de caracteres, que contiene los niveles (*Levels*) o categorías, a los que hace referencia el primer vector. La Fig. 2.5 muestra esta disposición, en la que, con motivo de no tener un desplegado confuso, se grafican sólo tres de las referencias del vector de índices al vector de niveles.

Abordemos ahora el problema que motivó la presente discusión: la frecuencia de aparición de ciertos elementos en un vector. La función `table()` toma típicamente como argumento un factor y regresa como resultado justamente la frecuencia de aparición de los niveles en el vector de índices:

```
table(Fmes.nacimiento)

## Fmes.nacimiento
## Abr Dic Feb Mar Nov Oct
##   1   3   3   1   1   2
```

La interpretación de estos resultados en el contexto de la estructura de información original, es que, por ejemplo, 3 personas del vector `persona`, nacieron en el mes de Dic. En el ejemplo mostrado, los niveles o `Levels` aparecen

ordenados alfabéticamente. La creación de factores en los que se establezca un orden determinado en los niveles, se puede hacer con la función `factor()`, como se muestra:

```
meses <- c("Ene","Feb","Mar","Abr","May","Jun","Jul","Ago",
           "Sep","Oct","Nov","Dic")
# Se incluyen meses que no estan el el vector original
FFmes.nacimiento <- factor(mes.nacimiento, levels=meses)
FFmes.nacimiento

##  [1] Dic Feb Oct Mar Feb Nov Abr Dic Feb Oct Dic
## 12 Levels: Ene Feb Mar Abr May Jun Jul Ago Sep Oct ... Dic

# Ahora la tabla de frecuencias es:
table(FFmes.nacimiento)

## FFmes.nacimiento
## Ene Feb Mar Abr May Jun Jul Ago Sep Oct Nov Dic
##   0   3   1   1   0   0   0   0   0   2   1   3
```

Debe notarse que la función `table()` pudiera haber recibido como argumento directamente el vector de caracteres original, y hubiera producido el resultado deseado, como se muestra:

```
table(mes.nacimiento)

## mes.nacimiento
## Abr Dic Feb Mar Nov Oct
##   1   3   3   1   1   2
```

La razón es simple: el intérprete del lenguaje *sabe* que la función está esperando recibir un factor y en consecuencia trata de convertir, en automático, el argumento que recibe, a esa clase. Como la conversión de vectores de caracteres a factores es trivial, la función no tiene ningún problema en desarrollar su tarea.

2.4.2. Acceso a los elementos de un factor

El acceso a cada uno de los dos vectores que le dan estructura al factor se hace como se muestra a continuación y se ha ilustrado también en la Fig. 2.5:

```
# Un elemento individual del factor:
Fmes.nacimiento[10]

## [1] Oct
## Levels: Abr Dic Feb Mar Nov Oct
```

```
# Un elemento individual de los niveles:
levels(Fmes.nacimiento)[3]

## [1] "Feb"
```

Incluso es posible modificar todos o algunos de los niveles del factor. Por ejemplo:

```
levels(Fmes.nacimiento)[3] <- "febrero"
Fmes.nacimiento

##  [1] Dic     febrero Oct     Mar     febrero Nov     Abr
##  [8] Dic     febrero Oct     Dic
## Levels: Abr Dic febrero Mar Nov Oct
```

Si se quiere tener acceso al factor como un vector de índices, se convierte a entero:

```
as.integer(Fmes.nacimiento)

##  [1] 2 3 6 4 3 5 1 2 3 6 2
```

2.5. Listas

Una lista, de la clase `list`, es una clase de datos que puede contener cero o más elementos, cada uno de los cuales puede ser de una clase distinta. Por ejemplo, se puede concebir una lista para representar una familia: la mamá, el papá, los años de casados, los hijos, y las edades de los hijos, de la manera siguiente:

```
familia <- list("Maria", "Juan", 10, c("Hugo", "Petra"), c(8,
    6))
familia

## [[1]]
## [1] "Maria"
##
## [[2]]
## [1] "Juan"
##
## [[3]]
## [1] 10
##
## [[4]]
## [1] "Hugo"  "Petra"
```

```
##
## [[5]]
## [1] 8 6
```

Nótese que la lista contiene cinco elementos; los tres primeros son a su vez de un sólo elemento: el nombre de la mamá, el nombre del papá, y los años de casados. Los siguientes dos, son dos vectores de dos elementos cada uno: los hijos y sus respectivas edades.

Al igual que en el caso de los vectores, como se vio en la sección 2.2.4 en la página 14, los elementos de las listas pueden ser nombrados, lo que añade mayor claridad a su significado dentro de la lista. La forma de hacer esto se muestra a continuación:

```
familia <- list(madre="Maria", padre="Juan", casados=10,
                hijos=c("Hugo", "Petra"), edades=c(8, 6))
familia
```

```
## $madre
## [1] "Maria"
##
## $padre
## [1] "Juan"
##
## $casados
## [1] 10
##
## $hijos
## [1] "Hugo"  "Petra"
##
## $edades
## [1] 8 6
```

2.5.1. Acceso a los elementos individuales de una lista

Al igual que en el caso de los vectores, las listas no serían de mucha utilidad sin la posibilidad de tener acceso a sus elementos individuales. El lenguaje, provee de este acceso mediante tres operadores, a saber: `[ ]`, `[[ ]]`, y `$`. El primero de estos operadores se revisará a detalle en el capítulo 3. Aquí se explicarán los otros dos operadores en su forma de uso más simple.

Cuando los elementos de la lista tienen nombre, se puede acceder a ellos con cualquiera de los dos operadores. Usando los ejemplos anteriores, esto se puede hacer de la manera siguiente:

```
# Acceso de lectura
familia$madre

## [1] "Maria"

familia[["madre"]]

## [1] "Maria"

# Acceso de escritura
familia[["padre"]] <- "Juan Pedro"
familia$padre  # para checar el nuevo valor

## [1] "Juan Pedro"
```

Nótese que al emplear el operador `$`, no se han usado las comillas para mencionar el nombre del elemento, pero, este operador también admite nombres con comillas. Por otra parte, el operador `[[ ]]`, sólo admite los nombres de elementos con comillas, o de cualquier expresión que al evaluarse dé como resultado una cadena de caracteres. En seguida se muestran algunos ejemplos:

```
familia$"madre" <- "Maria Candelaria"
mm <- "madre"
familia[[mm]]

## [1] "Maria Candelaria"

familia[[ paste("ma", "dre", sep="") ]]

## [1] "Maria Candelaria"
```

En el último caso, el operador ha recibido como argumento la función `paste()`, que, como se ha dicho anteriormente, en la página 28, sirve para concatenar cadenas de caracteres. Esta función supone de inicio que las cadenas irán separadas por un espacio en blanco. Por ello es que, en el ejemplo se indica que el separador es vacío mediante `sep=""`. Alternativamente, para este último caso, se puede usar la función `paste0()`, que de entrada supone que tal separador es vacío.

2.6. *Data frames*

Un *data frame*[1] es una lista, cuyos componentes pueden ser vectores, matrices o factores, con la única salvedad de que las longitudes, o número de

[1] *Usamos aquí el témino anglosajón, "data frames", y no su traducción al castellano, "marco o estructura de datos", dado que estos nombres sólo introducirían confusión, pues ninguno de ellos da una pista de lo que es. Probablemente un término apropiado sería algo como "tabla de datos".*

renglones, en el caso de matrices, deben coincidir en todos los componentes. La apariencia de un *data frame* es la de una tabla y una forma de crearlos es mediante la función `data.frame()` . Veamos un ejemplo:

```
(m <- cbind(ord=1:3, edad=c(30L, 26L, 9L)) )

##      ord edad
## [1,]   1   30
## [2,]   2   26
## [3,]   3    9

(v <- c(1.80, 1.72, 1.05) )

## [1] 1.80 1.72 1.05

ff <- data.frame(familia=c("Padre", "Madre", "Hijo"),
                 m, estatura=v)
ff

##   familia ord edad estatura
## 1   Padre   1   30     1.80
## 2   Madre   2   26     1.72
## 3    Hijo   3    9     1.05
```

Una gran ventaja de los *data frames*, es que R tiene diversas funciones para leer y guardar las tablas que representan, en archivos de texto, y otros formatos. Como un ejemplo, supongamos que se tiene un archivo, denominado "Rtext.txt", la siguiente información:

	Precio	Piso	Area	Cuartos	Edad	Calentador
01	52.00	111.0	830	5	6.2	no
02	54.75	128.0	710	5	7.5	no
03	57.50	101.0	1000	5	4.2	no
04	57.50	131.0	690	6	8.8	no
05	59.75	93.0	900	5	1.9	si

La lectura de esta tabla hacia un *data frame*, es muy sencilla y se hace mediante la función `read.table()`[2], como sigue:

```
mi.tabla <- read.table("Rtext.txt")
mi.tabla
```

[2]Aparte de la función `read.table()`, existen otras funciones que permiten leer datos de algún tipo de archivo y vaciarlos en una estructura de tipo data frame. Probablemente, una de las más útiles es la función `read.csv()`, que permite hacer esta operación a partir de archivos que contienen valores separados por comas, uno de los formatos de intercambio de información entre manejadores de hojas de cálculo, como Excel, más usados.

```
##    Precio Piso Area Cuartos Edad Calentador
## 01  52.00  111  830       5  6.2         no
## 02  54.75  128  710       5  7.5         no
## 03  57.50  101 1000       5  4.2         no
## 04  57.50  131  690       6  8.8         no
## 05  59.75   93  900       5  1.9         si
```

Nótese que el primer renglón y la primera columna no son parte de los datos de la tabla; ellos son, respectivamente, los nombres de las columnas y renglones de la tabla o *data frame*, lo que podemos constatar mediante las funciones `colnames()` y `rownames()`:

```
colnames(mi.tabla)

## [1] "Precio"     "Piso"       "Area"       "Cuartos"
## [5] "Edad"       "Calentador"

rownames(mi.tabla)

## [1] "01" "02" "03" "04" "05"
```

Como se mencionó anteriormente, un *data frame* es una lista muy particular, pero, ¿cuáles son los elementos de esa lista? Los elementos de la lista, y que obedecen a todas las reglas sintácticas dadas anteriormente (ver sección 2.5.1), son las columnas de la tabla. Así, por ejemplo, al segundo elemento de la lista podemos tener acceso de las siguientes formas:

```
mi.tabla$Piso

## [1] 111 128 101 131  93

mi.tabla[[2]]

## [1] 111 128 101 131  93

mi.tabla[2]

##    Piso
## 01  111
## 02  128
## 03  101
## 04  131
## 05   93
```

En el último caso, los datos se despliegan junto con el nombre de la columna y cada uno de los nombres de los renglones. Ello se debe a que en realidad, el operador `[]` extrae una *rebanada* del dato o variable sobre la cuál opera, un *data*

frame en este caso, y que podríamos denominarlo como un *sub-data frame* aquí; esto es, se trata otra vez de un *data frame* pero más *chiquito* que el original. Los detalles de este operador se discutirán a detalle más adelante en el texto.

Para tener acceso a un elemento individual de un data frame, se utiliza el operador `[]`, con la misma sintaxis que se utilizó para las matrices. Por ejemplo, el elemento en el renglón 3 y la columna 2, se puede revisar, o incluso cambiar con:

```
mi.tabla[3, 2]

## [1] 101

# modificamos el elemento con:
mi.tabla[3, 2] <- 106
mi.tabla

##    Precio Piso Area Cuartos Edad Calentador
## 01  52.00  111  830       5  6.2         no
## 02  54.75  128  710       5  7.5         no
## 03  57.50  106 1000       5  4.2         no
## 04  57.50  131  690       6  8.8         no
## 05  59.75   93  900       5  1.9         si
```

Otra característica importante de los *data frames* es que, salvo que se indique otra cosa, las columnas de tipo `character` se convierten automáticamente a tipo `factor`:

```
mi.tabla$Calentador

## [1] no no no no si
## Levels: no si

class(mi.tabla$Calentador)

## [1] "factor"
```

La posibilidad de operar con *rebanadas* de los data frames, es una de las cosas que hacen más atractivas a esta estructura. Si, por ejemplo, se quiere añadir, una nueva columna o componente del *data frame*, y ésta calcularla como el resultado de multiplicar el Precio por el Area, se puede hacer de la siguiente manera:

```
mi.tabla$Total <- mi.tabla$Precio * mi.tabla$Area
mi.tabla

##    Precio Piso Area Cuartos Edad Calentador Total
## 01  52.00  111  830       5  6.2         no 43160
```

```
## 02  54.75  128  710       5  7.5         no 38872
## 03  57.50  106 1000       5  4.2         no 57500
## 04  57.50  131  690       6  8.8         no 39675
## 05  59.75   93  900       5  1.9         si 53775
```

2.7. Funciones

A diferencia de otros lenguajes de programación procedurales, como C, Java, y PHP, en R las funciones constituyen una *clase*. Por ejemplo, los objetos de esa *clase* pueden ser asignados a variables; podría darse el caso, incluso, de armar una lista cuyos elementos fueran funciones.

Aunque la escritura de funciones es parte de la programación que se verá más adelante, se indicará aquí la forma de crear funciones como una herramienta para agrupar varias operaciones.

La sintaxis para la creación de una función es como sigue:

variable <- **function**(*arg_1*, *arg_2*, ..., *arg_n*) *expresion*

Como se puede ver, se trata de una asignación de un valor: la función, a una variable. A partir de esa definición, la variable se puede utilizar como el *nombre* de la función. En R, toda expresión tiene un valor, así que el valor de la *expresión* será lo que la función regresará cuando se aplique. En seguida se muestra un ejemplo.

```
hipotenusa <- function(x, y) {
    sqrt(x^2 + y^2)
}

class(hipotenusa)

## [1] "function"
```

En este caso, la función de biblioteca `sqrt()`, entrega la raíz cuadrada, y el operador `^`, eleva un valor a la potencia indicada como segundo argumento. La función entrega como resultado el último valor calculado que encuentre, aunque esta entrega se pude hacer explícita mediante la instrucción `return`, con lo cual la función anterior podría alternativamente ser codificada como:

```
hipotenusa <- function(x, y) {
    return(sqrt(x^2 + y^2))
}
```

Para utilizar esta función lo hacemos con:

```
hipotenusa(3, 4)

## [1] 5
```

Los argumentos de la función tienen nombres, y esos se pueden usar en el llamado a la función, cambiando incluso el orden en que aparecen en la definición de la función.

```
hipotenusa(y = 4, x = 3)

## [1] 5
```

Otra característica es que las funciones, en su definición, pueden tener valores asignados por defecto o en ausencia cuando es llamada la función:

```
hipotenusa <- function(x=3, y=4) { # valores por ausencia
    return( sqrt( x^2 + y^2 ) )
}
# Llamamos a la funcion con argumentos "ausentes"
hipotenusa()

## [1] 5
```

Las funciones toman sus datos de los argumentos dados o de las variables que "le están al alcance"[3] a la función, así por ejemplo, la siguiente función:

```
ff <- function(r) {
    return(PI * r^2)
}
```

si se ejecuta esta función, por ejemplo, con `ff(3)`, puede disparar un error, ya que no se ha definido el valor de PI. Pero si se ejecuta en la siguiente secuencia, se obtendrá un valor, aun cuando la función se haya definido con anterioridad:

```
PI <- 3.1416
ff(3)

## [1] 28.27
```

La noción de función que se ha presentado aquí es muy básica. En un capítulo posterior se presenta la creación de funciones enriquecida por las estructuras de control que se discutirán también más adelante.

[3] En la sección 5.2.2 en la página 91, se aborda con más detalle este tema.

2.8. Coerción

Se han abordado en este capítulo, no de una manera exhaustiva, pero sí para tener una idea clara de su potencial, los principales tipos de datos del lenguaje R. Estos tipos de datos son el fundamento para la construcción de otras *clases* de datos más complejas. Algunos de los tipos de datos admiten su conversión a otros tipos; para ello, el lenguaje provee de un conjunto de funciones de la forma: `as.<tipo>()`. En seguida se muestran algunos ejemplos.

Distintas conversiones entre datos numéricos:

```
x <- 1.03
x

## [1] 1.03

y <- as.integer(x)  # conversion a entero
y

## [1] 1

z <- as.complex(y)  # conversion a complejo
z

## [1] 1+0i

a <- c("1000", "2013.1", "0")
class(a)

## [1] "character"

b <- as.numeric(a)  # conversion de character a otro tipo
b

## [1] 1000 2013    0

class(b)

## [1] "numeric"

c <- as.logical(b)  # conversion a logico
# 0 es FALSE y distinto de 0 es TRUE
c

## [1]  TRUE  TRUE FALSE
```

También, puede haber conversiones entre clases de datos más estructuradas. Una que se antoja inmediata es la conversión de una matriz a un *data frame*:

```
(m <- matrix(1:20, nrow = 5, ncol = 4))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
```

```
ff <- as.data.frame(m)
ff
```

```
##   V1 V2 V3 V4
## 1  1  6 11 16
## 2  2  7 12 17
## 3  3  8 13 18
## 4  4  9 14 19
## 5  5 10 15 20
```

Nótese, en este último caso, que la función de conversión automáticamente ha asignado nombres a los renglones y las columnas del *data frame* creado. Para tener acceso a los elementos del *data frame*, lo podemos hacer mediante los nombres asignados a las columnas:

```
ff$V2
```

```
## [1]  6  7  8  9 10
```

Existen muchas más conversiones posibles; pero baste por ahora con las que se han visto, que ellas dan una idea de los mecanismos usados en el lenguaje para este tipo de operaciones.

Capítulo 3

Acceso a porciones o subconjuntos de datos

Una de las riquezas del lenguaje R, es la posibilidad de extraer porciones o subconjuntos de los distintos tipos de datos, mediante mecanismos diversos, algunos de los cuales se revisarán en el presente capítulo.

En las secciones 2.2.4, 2.3.2, y 2.5.1, se ha introducido el tema del acceso a los elementos individuales de varios de los datos estructurados empleados en R. En el presente capítulo, se extenderá ese concepto para cubrir ya no solamente elementos individuales, sino también porciones o subconjuntos de los datos.

3.1. Los operadores de acceso o selección

Los operadores de acceso a los datos estructurados: vectores, matrices, factores, listas y *data frames*, son:

- `[]` . (Ejemplo: `mtx[2,3]`) Este operador siempre regresa un objeto de la misma clase que el original[1] y se puede emplear para seleccionar más de un elemento.
- `[[]]` . (Ejemplo: `ff[['Nombre']]`) Este operador se emplea para extraer elementos de una lista; esto incluye a los data frames que, como se ha dicho anteriormente, son un tipo particular de lista. Sólo permite la extracción o el acceso a un sólo elemento, aunque el elemento en sí mismo puede ser compuesto, y el objeto resultante no necesariamente es de la misma clase que la lista o *data frame* original.
- `$` . (Ejemplo: `ff$Nombre`) Este operador se emplea para extraer o acceder a los elementos de una lista o un *data frame*, a partir del nombre del elemento. Este operador hace más o menos lo mismo que el operador `[[]]`.

[1] Hay una excepción a esta regla, que se discutirá más adelante en la sección 3.2.2.2 en la página 48.

De los anteriores, el primer operador, `[]`, es quizá el más poderoso, pero, por lo mismo, también el que involucra una sintaxis más compleja, como se verá en la siguiente sección.

3.2. El operador `[]`

En el capítulo anterior se ha visto el uso de este operador para seleccionar o tener acceso a elementos individuales de distintos tipos de datos estructurados: vectores, matrices, factores, listas y *data frames*. Se visitarán nuevamente esos tipos de datos estructurados, pero para un uso más elaborado del operador.

3.2.1. Vectores y factores

Tanto los vectores como los factores son estructuras unidimensionales, de este modo, el uso del operador `[]` es semejante en ambos. Anteriormente se ha visto como seleccionar un elemento de ambas estructuras. Veremos ahora como seleccionar grupos de elementos.

3.2.1.1. Selección de una secuencia de elementos, o elementos particulares

Un ejemplo de selección de una secuencia de elementos contenida en un vector podría ser con una *indexación* como la que se muestra a continuación:

```
# P.ej. para un vector de 20 numeros aleatorios, generados
# con la funcion rnorm(), que genera numeros aleatorios con
# una distribucion normal:
(v <- rnorm(20))

##  [1]  0.89588 -0.34744  0.27485  0.10492 -0.37921  1.55139
##  [7]  0.02214 -0.12507  1.57380  0.23438 -0.42632 -0.60389
## [13]  1.55622  0.06278 -0.31224  0.43291 -1.65316  1.53361
## [19] -0.07183 -1.52539

# Si queremos seleccionar de esos, solo los numeros en las
# posiciones de la 5 a la 15:
(subv <- v[5:15])

##  [1] -0.37921  1.55139  0.02214 -0.12507  1.57380  0.23438
##  [7] -0.42632 -0.60389  1.55622  0.06278 -0.31224
```

Nótese que efectivamente, como se ha dicho anteriormente, la clase de dato resultante de la operación es la misma que la de entrada, un vector numérico en este caso:

```
class(v)

## [1] "numeric"

class(subv)

## [1] "numeric"
```

El caso de los factores es similar. Se tomará un ejemplo introducido en la sección 2.4: un factor con ciertos meses de nacimiento:

```
(Fmes.nacimiento <- factor(c("Dic", "Feb", "Oct", "Mar", "Feb",
                    "Nov", "Abr", "Dic", "Feb", "Oct", "Dic"),
                    levels=c("Ene","Feb","Mar","Abr","May","Jun",
                     "Jul","Ago","Sep","Oct","Nov","Dic")))

##  [1] Dic Feb Oct Mar Feb Nov Abr Dic Feb Oct Dic
## 12 Levels: Ene Feb Mar Abr May Jun Jul Ago Sep Oct ... Dic
```

Si se quiere extraer de ahí solamente los elementos comprendidos entre el dos y el cinco, se hace así:

```
(sub.Fmes.nacimiento <- Fmes.nacimiento[2:5])

## [1] Feb Oct Mar Feb
## 12 Levels: Ene Feb Mar Abr May Jun Jul Ago Sep Oct ... Dic
```

Nótese que la salida sigue siendo un factor, con los mismos `Levels` (niveles), que el factor original, pero recortado de acuerdo con la operación indicada en el operador.

Recuérdese que el código `2:5`, en realidad es un vector entero, a saber: `2, 3, 4, 5`; se antoja entonces natural extender la *indexación*, al uso de vectores de enteros arbitrarios. Esto es, para seleccionar subconjuntos arbitrarios de los vectores o factores originales. Así por ejemplo, para seleccionar los elementos, 2, 3, y del 5 al 8, tanto en el vector como en el factor de los ejemplos anteriores, se puede hacer de la siguiente manera:

```
(sub1.v <- v[c(2, 3, 5:8)])

## [1] -0.34744  0.27485 -0.37921  1.55139  0.02214 -0.12507

(sub1.Fmes.nacimiento <- Fmes.nacimiento[c(2, 3, 5:8)])

## [1] Feb Oct Feb Nov Abr Dic
## 12 Levels: Ene Feb Mar Abr May Jun Jul Ago Sep Oct ... Dic
```

Por otra parte, en esta mismo tema, los índices negativos, tienen el significado de excluir los elementos señalados por el índice. Así, el complemento de los subconjuntos antriores se puede obtener así:

```
(sub2.v <- v[-c(2, 3, 5:8)])

##  [1]  0.89588  0.10492  1.57380  0.23438 -0.42632 -0.60389
##  [7]  1.55622  0.06278 -0.31224  0.43291 -1.65316  1.53361
## [13] -0.07183 -1.52539

(sub2.Fmes.nacimiento <- Fmes.nacimiento[-c(2, 3, 5:8)])

## [1] Dic Mar Feb Oct Dic
## 12 Levels: Ene Feb Mar Abr May Jun Jul Ago Sep Oct ... Dic
```

3.2.1.2. Selección de elementos de acuerdo con una condición

En la sección 2.2.5 en la página 16, se han introducido los operadores lógicos. Haciendo uso de esa noción, se puede por ejemplo, distinguir en el vector `v`, los elementos negativos de aquellos que no lo son, obteniendo un vector de lógicos, de la siguiente manera:

```
v < 0

##  [1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
## [10] FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE
## [19]  TRUE  TRUE
```

Surge la pregunta: ¿cómo se pueden extraer, con esta información, los elementos negativos o los positivos del vector original? El lenguaje permite utilizar como índice del operador `[]`, un vector de lógicos en el que se obtendrá como salida un vector compuesto de los elementos cuyo índice sea `TRUE`. Entonces, la respuesta a la pregunta original se puede dar de las siguientes maneras:

```
# Los negativos:
v[v < 0]

## [1] -0.34744 -0.37921 -0.12507 -0.42632 -0.60389 -0.31224
## [7] -1.65316 -0.07183 -1.52539

# Primera forma de obtener los positivos, mediante la
# negacion logica, con el operador !
v[!(v < 0)]

##  [1] 0.89588 0.27485 0.10492 1.55139 0.02214 1.57380 0.23438
##  [8] 1.55622 0.06278 0.43291 1.53361
```

```
# Segunda forma de obtener los positivos, mediante el
# operador >=
v[v >= 0]

##  [1] 0.89588 0.27485 0.10492 1.55139 0.02214 1.57380 0.23438
##  [8] 1.55622 0.06278 0.43291 1.53361
```

Para factores solo la operaciones logicas para probar la igualdad o desigualdad tienen significado:

```
Fmes.nacimiento == "Mar"  # igualdad

##  [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
## [10] FALSE FALSE

Fmes.nacimiento != "Mar"  # desigualdad

##  [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
## [10]  TRUE  TRUE
```

No obstante, dado que, como se ha señalado en la sección 2.4.1, los factores tienen implícito un vector de enteros, que indexan el orden establecido por los `Levels` (niveles) del factor, se puede usar ese hecho para descubrir, en el caso de ejemplo, cuáles son los meses menores o iguales que ''`Abr`'' (número de orden 4, en `Levels`), de la siguiente manera:

```
# El factor convertido a enteros:
as.integer(Fmes.nacimiento)

##  [1] 12  2 10  3  2 11  4 12  2 10 12

# El vector de logicos:
as.integer(Fmes.nacimiento) <= 4

##  [1] FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
## [10] FALSE FALSE

# .. y usado como indice:
Fmes.nacimiento[as.integer(Fmes.nacimiento) <= 4]

## [1] Feb Mar Feb Abr Feb
## 12 Levels: Ene Feb Mar Abr May Jun Jul Ago Sep Oct ... Dic
```

Una nota importante es que los operadores de selección no solamente se usan para consultar los valores, sino que también se pueden emplear para cambiar los valores de los elementos seleccionados mediante el operador. Así por ejemplo, podríamos cambiar cada uno los elementos negativos del vector `v`, a su correspondiente positivo, mediante la siguiente asignación:

```
v[v < 0] <- -v[v < 0]
# .. y el vector v ahora es:
v

##  [1] 0.89588 0.34744 0.27485 0.10492 0.37921 1.55139 0.02214
##  [8] 0.12507 1.57380 0.23438 0.42632 0.60389 1.55622 0.06278
## [15] 0.31224 0.43291 1.65316 1.53361 0.07183 1.52539
```

3.2.2. Matrices y *data frames*

Las matrices y los *data frames*, son estructuras bidimensionales; es decir, tienen renglones y columnas, y por consiguiente, su comportamiento bajo el operador `[]` es similar. Como se ha visto en el capítulo 2, el acceso a los elementos individuales a estas dos estructuras consta de dos índices, separados por una coma, en el operador, así: `x[i, j]`; donde, `x`, es la estructura en cuestión, `i`, representa el número o identificador[2] de renglón y `j`, el número o identificador de columna.

Para las explicaciones y ejemplos que siguen se usarán las matrices y *data frames* que se generan a continuación:

```
(mt <- matrix(11:30, nrow = 4, ncol = 5))

##      [,1] [,2] [,3] [,4] [,5]
## [1,]   11   15   19   23   27
## [2,]   12   16   20   24   28
## [3,]   13   17   21   25   29
## [4,]   14   18   22   26   30

# Se convierte la matriz a un data frame:
df.mt <- as.data.frame(mt)

# Se le asignan nombres a renglones y columnas de df.mt
rownames(df.mt) <- c("uno", "dos", "tres", "cuatro")
colnames(df.mt) <- c("UNO", "DOS", "TRES", "CUATRO", "CINCO")
df.mt

##        UNO DOS TRES CUATRO CINCO
## uno     11  15   19     23    27
## dos     12  16   20     24    28
## tres    13  17   21     25    29
## cuatro  14  18   22     26    30
```

[2]Recuérdese que tanto los renglones como las columnas pueden tener nombres y que éstos pueden ser utilizados en vez de los índices numéricos correspondientes (cf. p. 22).

3.2.2.1. El operador [] con un solo índice

Tanto para matrices como para *data frames*, el lenguaje permite utilizar el operador [] con un solo índice, si bien el significado es diferente para cada uno de los casos. En el caso de las matrices el uso de un solo índice, provoca el tratamiento de la matriz como si fuera un vector constituído por la concatenación de sus columnas y el índice provisto se referirá entonces al elemento correspondiente a esa posición en el vector, mientras que en el caso de los *data frames*, dado que éstos son listas cuyos elementos son las columnas, ese uso del operador invoca justamente ese tratamiento; esto es, la operación regresará la columna correspondiente al índice dado. Veamos sendos ejemplos de esta operación:

```
# La matriz con indice 5
mt[5]

## [1] 15

# El data frame con indice 5
df.mt[5]

##        CINCO
## uno       27
## dos       28
## tres      29
## cuatro    30
```

3.2.2.2. Omisión de índices en el operador

El caso anterior no se debe confundir con la omisión de índices en el operador, ya que en este caso, el operador, mediante una coma señala el espacio para dos índices, uno de los cuales se omite. Aquí, la semántica del operador es similar para matrices y para *data frames*, ya que en ambos casos, el operador regresa, o bien renglones, o bien columnas, de la estructura sobre la cual se opera. Veamos unos ejemplos:

```
# El tercer renglón:
mt[3, ]

## [1] 13 17 21 25 29

df.mt[3, ]

##      UNO DOS TRES CUATRO CINCO
## tres  13  17   21     25    29

# La tercer columna:
mt[, 3]
```

```
## [1] 19 20 21 22

df.mt[, 3]

## [1] 19 20 21 22
```

Se había dicho en la sección 3.1 en la página 41 que el operador `[]`, *siempre* regresa un objeto del mismo de la misma clase que el original. Los ejemplos anteriores manifiestan una excepción a esta regla, ya que:

```
class(mt[3, ])

## [1] "integer"

class(df.mt[3, ])  # solo en este caso se cumple

## [1] "data.frame"

class(mt[, 3])

## [1] "integer"

class(df.mt[, 3])

## [1] "integer"
```

El lenguaje tiende a simplificar, cuando puede, a la clase más básica de dato, es decir, a vector; lo que podría resultar útil en algunos casos, pero que, en otros casos, podría complicar la programación. Este *extraño* comportamiento, no ortogonal[3], sin embargo, se puede modificar mediante la opción `drop=FALSE`, al usar el operador, como se muestra a continuación:

```
# El tercer renglón:
mt[3, , drop = FALSE]

##      [,1] [,2] [,3] [,4] [,5]
## [1,]   13   17   21   25   29

class(mt[3, , drop = FALSE])

## [1] "matrix"

df.mt[3, , drop = FALSE]
```

[3]En el argot computacional el término *ortogonal* se refiere a que a una misma estructura sintáctica le corresponde una misma estructura semántica o de significado. Esto quiere decir que el operador se comportaría uniformemente, entregando el mismo tipo de resultados, para una misma sintáxis.

```
##      UNO DOS TRES CUATRO CINCO
## tres  13  17   21     25    29

class(df.mt[3, , drop = FALSE])

## [1] "data.frame"

# La tercer columna:
mt[, 3, drop = FALSE]

##      [,1]
## [1,]   19
## [2,]   20
## [3,]   21
## [4,]   22

class(mt[, 3, drop = FALSE])

## [1] "matrix"

df.mt[, 3, drop = FALSE]

##        TRES
## uno      19
## dos      20
## tres     21
## cuatro   22

class(df.mt[, 3, drop = FALSE])

## [1] "data.frame"
```

En este mismo tema, el operador permite la selección simultánea de varios renglones o varias columnas. A continuación se muestra el caso de varias columnas, y, dado que es muy similar, se deja al lector investigar cuál sería la forma de seleccionar varios renglones.

```
# Selección de las columnas 4,3,2, en ese orden:
mt[, 4:2]

##      [,1] [,2] [,3]
## [1,]   23   19   15
## [2,]   24   20   16
## [3,]   25   21   17
## [4,]   26   22   18

df.mt[, 4:2]
```

```
##        CUATRO TRES DOS
## uno        23   19  15
## dos        24   20  16
## tres       25   21  17
## cuatro     26   22  18
```

Nótese que en el caso de múltiples columnas o renglones, dado que en su conjunto no pueden ser simplificados a vectores, el comportamiento del operador `[]`, es ortogonal y obedece a la regla de entregar siempre un objeto de la misma clase que el original.

Los mecanismos que habilita el operador `[]` y que se han descrito aquí, también permiten la selección de una *ventana* en en interior de ambas estructuras.

```
mt[1:3, 4:2]

##      [,1] [,2] [,3]
## [1,]   23   19   15
## [2,]   24   20   16
## [3,]   25   21   17

df.mt[1:3, 4:2]

##      CUATRO TRES DOS
## uno      23   19  15
## dos      24   20  16
## tres     25   21  17
```

En este caso, como en el ejemplo anterior, se ha cambiado el orden de la secuencia de columnas en la *ventana* resultante, lo que, por cierto, resulta más evidente en el caso del *data frame* por el uso de nombres tanto para columnas como para renglones[4].

3.2.2.3. El uso de índices lógicos o condiciones

Al igual que en el caso de los vectores y los factores, este operador admite índices de tipo lógico, que resultan de la expresión de condiciones, que pueden ser tan complicadas como se quiera. Se verán aquí algunos ejemplos sencillos tanto para el caso de matrices como para el caso de *data frames*.

En la matriz y el data frame del ejemplo anterior, el segundo rengón se puede obtener fácilmente de la siguiente manera:

```
mt[2, ]

## [1] 12 16 20 24 28
```

[4]Esto no tiene que ver con la clase del objeto, sino con el hecho de que no se han asignado nombres ni a los renglones, ni a las columnas de la matriz `mt`, en el caso del ejemplo.

```
df.mt[2, ]

##     UNO DOS TRES CUATRO CINCO
## dos  12  16   20     24    28
```

Ahora, supóngase que se quiere obtener ese mismo renglón, pero basado en el hecho de que el valor en la columna 2 es 16. Esto es: obtener el renglón, o renglones, cuyo valor en la columna 2 es 16. Primeramente, obsérvese que todos los valores en la columna 2 de la matriz o del *data frame*, se pueden comparar contra el valor 16, de la siguientes maneras:

```
mt[, 2] == 16

## [1] FALSE  TRUE FALSE FALSE

df.mt[, 2] == 16

## [1] FALSE  TRUE FALSE FALSE
```

Como se puede ver el resultado es un vector de lógicos, cada uno de los cuales corresponde a la comparación por igualdad de cada uno de los elementos en la columna dos contra el valor 16; esto es, en *cada uno de los renglones* de esa columna. Esta comparación se puede utilizar como índice en el espacio correspondiente a los renglones en el operador `[]`, para obtener los renglones que cumplen con la condición establecida. En este caso el arreglo de lógicos resultante de la condición actúa como una máscara o un filtro que sólo deja *pasar*, del la matriz, aquellos elementos para los cuales hay un valor `TRUE`, como se muestra a continuación:

```
# Se usan parentesis, (), para enfatizar la condicion, aunque
# se podria prescindir de ellos:
mt[(mt[, 2] == 16), ]

## [1] 12 16 20 24 28

df.mt[(df.mt[, 2] == 16), ]

##     UNO DOS TRES CUATRO CINCO
## dos  12  16   20     24    28

# En el caso de la matriz, si se quiere obtener como salida
# una matriz (de un solo renglon), se hace asi:
mt[(mt[, 2] == 16), , drop = FALSE]

##      [,1] [,2] [,3] [,4] [,5]
## [1,]   12   16   20   24   28
```

Modifiquemos ahora la matriz y el *data frame*, para tener más de un renglón que cumple con esta condición:

```
mt[4, 2] <- 16L
df.mt[4, 2] <- 16L
mt  # (El data frame es semejante)

##      [,1] [,2] [,3] [,4] [,5]
## [1,]   11   15   19   23   27
## [2,]   12   16   20   24   28
## [3,]   13   17   21   25   29
## [4,]   14   16   22   26   30
```

Y ahora, si se aplica nuevamente la operación de prueba, lo que se obtiene es un conjunto de renglones:

```
mt[(mt[, 2] == 16), ]

##      [,1] [,2] [,3] [,4] [,5]
## [1,]   12   16   20   24   28
## [2,]   14   16   22   26   30

df.mt[(df.mt[, 2] == 16), ]

##        UNO DOS TRES CUATRO CINCO
## dos     12  16   20     24    28
## cuatro  14  16   22     26    30
```

Las expresiones o pruebas lógicas usadas como índices pueden ser más complejas si se quiere. Supongamos que se quiere obtener, todas las columnas que en su renglón 2 no son múltiplos de 8; esto se hace como se muestra en seguida.

```
# La prueba logica hace uso del operador modulo o residuo: %%
mt[2, ]%%8 != 0  # (Para el data frame es semejante)

## [1]  TRUE FALSE  TRUE FALSE  TRUE

# Ahora usemos la expresion como indice:
mt[, (mt[2, ]%%8 != 0)]

##      [,1] [,2] [,3]
## [1,]   11   19   27
## [2,]   12   20   28
## [3,]   13   21   29
## [4,]   14   22   30

df.mt[, (df.mt[2, ]%%8 != 0)]
```

```
##        UNO TRES CINCO
## uno     11   19    27
## dos     12   20    28
## tres    13   21    29
## cuatro  14   22    30
```

El uso y el potencial que este tipo de operaciones pueden tener en la práctica, es algo que sin duda sólo tiene como límite la imaginación del programador o usuario del lenguaje, y que definitivamente cae dentro de lo que en este libro se llama *el arte de programar en R*.

3.3. Los operadores `[[]]` y `$`

Los operadores `[[]]` y `$`, son más o menos semejantes, aunque éste último limita su aplicación a las listas y, por consiguiente, también a los *data frames*. Estos operadores establecen formas de tener acceso a los elementos de las distintas estructuras mediante los nombres o identificadores asignados a esos elementos. Si bien es cierto que el operador `[]`, admite también nombres de los elementos como índices, como se ha mostrado en la sección 2.2.4 en la página 14, los operadores discutidos en esta sección habilitan formas más flexibles de tener acceso a los elementos de las estructuras mediante sus nombres, amén de que ese acceso va a un nivel más *profundo*. Para comprender esto, piénsese en el siguiente simil: sea la estructura sobre la que actúan los operadores como una bodega que contiene cajas, que a su vez contienen distintos objetos. En general, el operador `[]`, podría considerarse como un dispositivo que *mueve* algunas de las cajas a otra, por así decirlo, *sub-bodega* o bodega *más pequeña*, y ese sería el resultado de la operación; es decir, entrega la *sub-bodega* con las cajas seleccionadas, mientras que los otros operadores entregan las cajas o incluso el contenido de éstas.

A manera de comparación, se dotará a la matriz `mt`, utilizada anteriormente, de nombres para sus columnas y renglones:

```
rownames(mt) <- c("uno", "dos", "tres", "cuatro")
colnames(mt) <- c("UNO", "DOS", "TRES", "CUATRO", "CINCO")
mt

##        UNO DOS TRES CUATRO CINCO
## uno     11  15   19     23    27
## dos     12  16   20     24    28
## tres    13  17   21     25    29
## cuatro  14  16   22     26    30
```

Aquí, el acceso a renglones y columnas por nombres, mediante el operador `[]`, da resultados *semejantes* tanto en matrices como en *data frames*:

```
mt[, "TRES"]

##    uno    dos   tres cuatro
##     19     20     21     22

df.mt[, "TRES"]

## [1] 19 20 21 22

mt["dos", ]

##    UNO    DOS   TRES CUATRO  CINCO
##     12     16     20     24     28

df.mt["dos", ]

##     UNO DOS TRES CUATRO CINCO
## dos  12  16   20     24    28

# Para comparacion con el operador [[]]
mt["dos", "TRES", drop = F]

##     TRES
## dos   20

class(mt["dos", "TRES", drop = F])  # La clase del objeto

## [1] "matrix"

df.mt["dos", "TRES", drop = F]  # F es lo mismo que FALSE

##     TRES
## dos   20

class(df.mt["dos", "TRES", drop = F])  # La clase del objeto

## [1] "data.frame"
```

Como se ha dicho anteriormente, un *data frame* es una lista muy particular, cuyos componentes son las columnas del *data frame*, mismas que siempre tienen la misma longitud, y que en el caso del ejemplo, `df.mt`, es exactamente 4. De este modo, lo que se diga aquí para los *data frames*, con referencia a los operadores estudiados, en general es también válido para las listas.

El operador `[[]]`, permite el acceso a esos componentes, ya sea mediante índices numéricos o los nombres provistos a los elementos de las estructuras. Asi, para los dos últimos casos mostrados anteriormente:

```
mt[[2, 3]]

## [1] 20

mt[["dos", "TRES"]]

## [1] 20

class(mt[["dos", "TRES"]])

## [1] "integer"

df.mt[[2, 3]]

## [1] 20

class(df.mt[[2, 3]])

## [1] "integer"

df.mt[["dos", "TRES"]]

## [1] 20
```

Nótese que a diferencia con el operador `[]`, el operador `[[]]`, no entrega en ni una matriz, ni un data frame, sino un vector entero de un solo elemento, en este caso. Compárese con los dos últimos casos del ejemplo anterior.

Otra diferencia importante de este operador es que no admite ni rangos, ni conjuntos de índices; esto es, para cada espacio en el operador sólo admite ya sea un índice entero o una cadena de caracteres que identifica el nombre de algún elemento de la estructura sobre la que opera.

El operador `$`, es semejante pero sólo actúa sobre la estructura unidimensional de una lista o de un *data frame*. La diferencia de este operador con el operador `[[]]`, es que los nombres de los elementos no necesitan ir entrecomillados, y pueden estar incompletos, cuando no hay ambigüedad en la identificación de los elementos a los cuales se refieren. Esta característica resulta más útil cuando se trabaja con el lenguaje directamente desde la consola, o sea, interactivamente, ya que puede representar alguna economía en la escritura de las expresiones.

```
df.mt[["TRES"]]

## [1] 19 20 21 22

df.mt$TRES

## [1] 19 20 21 22
```

```
df.mt$"TRES"

## [1] 19 20 21 22

df.mt$T

## Warning: Name partially matched in data frame

## [1] 19 20 21 22
```

El operador `[[]]`, también admite nombres incompletos, pero ese comportamiento tiene que ser señalado explícitamente por medio de la opción `exact = FALSE`, en el operador:

```
df.mt[["TR", exact = F]]  # Recuerde F es FALSE

## [1] 19 20 21 22
```

La riqueza en el uso de estos operadores se irá descubriendo a medida que se incorporen otras caracteristicas del lenguaje que se estudiarán en los capítulos siguientes.

Capítulo 4

Estructuras de control y manejo de datos

En los lenguajes de programación, se entiende por estructuras de control aquellas construcciones sintácticas del lenguaje que dirigen el flujo de la ejecución de un programa en *una dirección* o en otra dentro de su código. Por ejemplo, prácticamente todos los lenguajes tienen una construcción "IF", que permite ejecutar o saltar un conjunto, bloque o secuencia de instrucciones dentro del código de un programa. R también cuenta con un conjunto de estructuras de control, si bien, mucho de lo que éstas implementan se puede también hacer mediante

4.1. La construcciones IF-ELSE

Estas construcciones son semejantes a las de otros lenguajes de programación, con una salvedad que puede ser capitalizada por los usuarios del lenguaje: la construcción en sí misma regresa un valor, que puede, si se quiere, ser asignado a una variable o utilizado de otras maneras. Los siguientes ejemplos muestran la sintaxis y el uso de estas construcciones.

```
aa <- 15
if (aa > 14) # if sin else
  print("SI MAYOR")

## [1] "SI MAYOR"

if (aa > 14) print ("SI MAYOR")

## [1] "SI MAYOR"

if (aa > 14) { # Instruccion compuesta
```

```
  print ("PRIMER RENGLON")
  print ("SI MAYOR")
}

## [1] "PRIMER RENGLON"
## [1] "SI MAYOR"

# Usando el valor que regresa el if
y <- 10
y <- if (aa > 14) 50
y

## [1] 50
```

La construcción IF admite una sola expresión, pero ésta puede ser la expresión compuesta, que se construye mediante los paréntesis de llave `{ }`, y las expresiones en su interior, separadas ya sea por el cambio de renglón o por `';'`. En los casos anteriores, la expresión señalada por el `if` se ejecuta u omite dependiendo del valor de la condición, `TRUE` o `FALSE`, respectivamente. En el caso del ejemplo la condición esta dada por la expresión `aa > 14`, que prueba si la variable `aa` es mayour que `14`. La siguientes construcciones, redirigen la ejecución del código a distintos bloques o conjuntos de instrucciones dependiendo de que se cumplan o no las condiciones establecidas:

```
if (10 > aa) { # 1er. bloque
    print("RANGO MENOR")
} else if ( 10 <= aa && aa <= 20) { # 2o. bloque
    print("primer renglon"); print("RANGO MEDIO")
} else { # 3er. bloque
    print("RANGO MAYOR")
}

## [1] "primer renglon"
## [1] "RANGO MEDIO"
```

Nótese que el segundo bloque de expresiones en el ejemplo, es una expresión compuesta de dos expresiones; pero como ellas se han dado en un solo renglón se han separado mediante el caracter `;`. La condición que da lugar a este mismo bloque de expresiones, introduce dos nuevos operadores: `<=` y `&&`; el primero de ellos es el operador de comparación *menor o igual que* y el segundo es el operador lógico *and*. Entonces la condición es equivalente a la expresión matemática: $10 \leq \texttt{aa} \leq 20$[1]. Finalmente, el tercer y último bloque de expresiones, se ejecuta en caso de que ninguna de las condiciones correspondientes a los otros dos bloques se haya satisfecho.

[1]Si desea saber más acerca de los distintos operadores disponibles en el lenguaje, introduzca '??operator' en la consola del intérprete del lenguaje

4.2. Los ciclos

El lenguaje cuenta con varios tipos de ciclos o repeticiones, a saber: repeticiones por un número determinado de veces, repeticiones mientras se cumple una condición y repeticiones *infinitas*. En seguida se discutirá cada uno de estos casos.

4.2.1. Repeticiones por un número determinado de veces

La construcción que habilita esta operación es la instrucción `for`. El número de veces que se repite la expresión o expresiones englobadas en la instrucción, puede estár explícita en ella misma, como se muestra a continuación:

```
letras <- c("c", "l", "i", "M", "T", "A")
for (i in 1:6) {
    print(letras[i])
}

## [1] "c"
## [1] "l"
## [1] "i"
## [1] "M"
## [1] "T"
## [1] "A"
```

El número de veces que se repite la expresión, puede quedar implícito en las estructuras de las cuales toma elementos el `for`. Así, el mismo resultado que se obtuvo en el ejemplo anterior, se puede obtener con cualquiera de las siguientes dos contstrucciones:

```
for (i in seq_along(letras)) {
    print(letras[i])
}

for (letra in letras) {
    print(letra)
}
```

En el primer caso se llamó a la función `seq_along()`, que genera una secuencia de enteros de acuerdo con el número de elementos que tenga el objeto que se le de como argumento. El segundo caso, tipifica la escencia de la construcción `for`, ya que se trata de ir tomando uno a uno los elementos del objeto consignado después de la partícula `in` del `for`.

4.2.2. Repeticiones mientras se cumple una condición

La construcción que habilita esta operación es la instrucción `while`, que se puede ejemplificar como sigue:

```
# Para la misma tarea de los ejemplos anteriores
i <- 1
while (i <= 6) {
    print(letras[i])
    i <- i + 1
}
```

La salida de este ejemplo es la misma que la de los ejemplos anteriores. En este caso, si no se tiene cuidado en el manejo del índice `i`, involucrado en la condición, se puede dar lugar a un ciclo sin salida.

4.2.3. Repeticiones *infinitas*

La construcción que habilita esta operación es la instrucción `repeat`. Aunque en realidad no se quiere significar que las repeticiones sean infinitas o interminables, como la instrucción no tiene condición de salida o interrupción, el resultado que la instrucción produciría en sí misma sería una repetición interminable; pero, el lenguaje provee facilidades para que desede el interior del bloque de expresiones que se repiten, se obligue la interrupción del ciclo, por ejemplo, mediante la instrucción `break`, que se detalla más adelante. Así, para producir los mismos resultados que en los ejemplos anteriores, se puede hacer así:

```
i <- 1
repeat {
    print(letras[i])
    i <- i + 1
    if (i > 6)
        break
}
```

En este caso, mediante un `if` que prueba una condición de salida, se *dispara* una instrucción `break` que interrumpe el ciclo.

4.2.4. Interrupciones del flujo normal de los ciclos

El flujo normal de los ciclos se puede interrumpir básicamente por medio de tres instrucciones diferentes: `break`, `next` y `return`.

En el último ejemplo de la sección anterior, 4.2.3, se ha utilizado la instrucción `break` para obligar la salida de un ciclo infinito que se realiza mediante la instrucción `repeat`. No es éste, sin embargo, el único ciclo que puede interrumpir la instrucción `break`, ya que ella se puede utilizar en el interior de cualquier

ciclo para forzar su interrupción. Como un ejemplo, se presenta un ciclo en que simula lo que podría ser un procedimiento iterativo para encontrar el valor de una variable, cuyo valor converge hacia un cierto valor con cada nueva iteración. Para no caer en ciclos verdaderamente infinitos, este tipo de procedimientos limitan el número de iteraciones a un valor suficientemente grande, lo que aquí se hace mediante una instrucción `for` limitada a 1000 repeticiones:

```
# se usara un generador de numeros aleatorios,
# la siguiente funcion asegura su repetibilidad:
set.seed(140) # el argumento puede ser cualquier numero
aprox <- 0.003 # Valor determinante para la salida del ciclo

Y_ini <- 2.7 # Supuesto valor inicial de Y
for (iter in 1:1000) { # aseguro no mas de 1000 iteraciones
    # Procedimiento para calcular la siguiente Y, que
    # en este caso simularemos mediante generador aleatorio:
    Y <- Y_ini + 0.008*rnorm(1)
    # La condicion de salida:
    if (abs(Y - Y_ini) <= aprox)
        break # Uso del break para salir del ciclo
    # Preparamos para la siguiente iteracion
    Y_ini <- Y
}
# Veamos que ha resultado:
paste0("Y_ini: ", Y_ini, ", Y: ", Y, ", Num.iter: ", iter)

## [1] "Y_ini: 2.76443400590741, Y: 2.76582777768031, Num.iter: 8"
```

En este ejemplo, el objetivo se ha alcanzado en 8 iteraciones. Se ha utilizado la función `abs()`, que entrega el valor absoluto de su argumento, y se ha utilizado un generador de números aleatorios con distribución normal, implementado mediante la función `rnorm()`, que se inicializa mediante la función `set.seed()`[2].

Un caso parecido de salida o interrupción de un ciclo es la instrucción `return`. Esta instrucción está asociada con las funciones y su propósito es interrumpir u obligar la salida de la función en la cuál se invoca, entregando, opcionalmente, como resultado de la función un valor si se da como argumento del `return`. De esta manera, la interrupción de un ciclo es realmente colateral, pero igualmente efectiva, solamente que la salida de ciclo no es exactamente *afuera* de él, sino *afuera* de la ejecución de la función en la que se ha invocado. Como un ejemplo, se creará y ejecutará una función: la función generadora de

[2]Tal como se ha llamado en el ejemplo, la función rnorm(), entregará valores que tendrán un valor medio 0, un gran porcentaje de los cuáles (68 %) estará entre -1 y 1. Para mayor información sobre estas funciones, introduzca "?rnorm" y "?set.seed", en la consoloa de su intérprete de R. En el caso del ejemplo, la convergencia está simulada por el hecho de que la media de los números aleatorios es precisamente 0; entonces el incremento entre un valor y el siguiente tenderá a ser 0.

los números de fibbonacci, que, tenidos o dados los dos primeros números de fibbonacci, F_0 y F_1, definidos ambos como 1, calcula cada uno de los siguientes como la suma de los dos anteriores.

```
# Primero se crea la funcion:
fibbonacci <- function(n) {
    if (n %in% c(0,1))
        return (1)

    F0 <- 1; F1 <- 1; i <- 2
    repeat {
        s <- F0 + F1 # Suma de los fib anteriores
        if (i == n) # Ya es el que se busca
            return (s) # Sale hasta afuera de la funcion

        # recorremos los ultimos dos proximos numeros
        F0 <- F1
        F1 <- s
        i <- i+1 # incrementamos el indice
    }
}

# El octavo numero de fibbonacci se genera
# llamando a la funcion asi:
fibbonacci(8)

## [1] 34
```

Esta función utiliza el operador `%in%`, que identifica si un cierto elemento está dentro de un conjunto representado por un vector. La instrucción `return`, es una función primitiva que termina la ejecución de una función y entrega como resultado de la función el argumento que se le pase. En el caso del ejemplo, se ha usado dos veces: la primera, simplemente detecta si el argumento es `0` o `1`, en cuyo caso termina la función y entrega `1` como resultado; la segunda vez, que es la que nos interesa, se usa dentro de la instrucción `repeat` para determinar si ya se ha llegado al número fibbonacci correspondiete, en cuyo caso termina allí la función y entrega como resultado el número correspondiente.

La instrucción `next` interrumpe el flujo normal de ejecución de un programa de una manera diferente: en vez de salir de un ciclo, solamente impide la ejecución de las instrucciones *siguientes* y regresa al principio del ciclo para ejecutar la siguiente iteración. El siguiente ejemplo ilustra esta operación:

```
for (i in 1:7) {
    if (3 <= i && i <= 5)
        next
    print(i)
```

```
}

## [1] 1
## [1] 2
## [1] 6
## [1] 7
```

Hasta aquí, se han visto diferentes estructuras de control que, al igual que otros lenguajes de programación, permiten definir el flujo de ejecución de las instrucciones de algún programa. A través de estas estructuras de control se pueden manipular los elementos de las clases de datos compuestas. La riqueza de este lenguaje, sin embargo, está en el manejo de cada una de las distintas estructuras de información, implementadas a través de las clases de datos estructuradas, como vectores, factores, *data frames*, etc., como un todo a través de funciones que las contemplan de esa manera. Este es el tema que se desarrollará en las siguientes secciones.

4.3. Funciones de clasificación, transformación y agregación de datos

En R hay diversas funciones que permiten atravesar las estructuras de información compuestas, ejecutando operaciones sobre los elementos o componentes de éstas.

4.3.1. Motivación

Para comprender mejor las funciones de transformación, clasificación y agregación de datos, se proponen aquí unos ejemplos sencillos: el cálculo del módulo o magnitud de un vector de números reales y el cálculo del promedio, o media, de un conjunto de datos provistos como un vector numérico.

En el primer caso, recuérdese que dado un vector n-dimensional:

$$\mathbf{v} = \langle v_1, v_2, \ldots, v_n \rangle,$$

su módulo o magnitud está dado por:

$$|\mathbf{v}| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

Dadas las estructuras de control aprendidas previamente, y sabiendo que el operador `^`, sirve para elevar a una potencia dada y la función `sqrt()`, extrae la raíz cuadrada, la tentación inmediata es resolver esta operación como se hace en la mayoría de los lenguajes *procedurales*:

```
# Sea el vector:
vv <- c(-2, 3, 4)
# Se creara una funcion que haga el trabajo
modulo <- function(v) {
    s <- 0  # Contendra la suma de cuadrados
    for (elt_v in v) {
        s <- s + elt_v^2  # Incrementamos la suma
    }
    # El resultado es la raiz de la suma:
    sqrt(s)
}
# y el modulo que queremos es:
modulo(vv)

## [1] 5.385
```

En la sección 2.2.5, sin embargo, se aprendió que las operaciones aritméticas se pueden distribuir a lo largo de los elementos de un vector, y esta es una característica del lenguaje de la que se puede sacar provecho en este momento; recurriendo adicionalmente a la función de agregación `sum()`, que toma como argumento un vector y suma uno a uno sus elementos y entrega esa suma como resultado. Entonces en R, la función `modulo()`, se puede programar así:

```
modulo0 <- function(v) {
    sqrt(sum(v^2))
}
# Puede quedar en una linea, asi:
modulo0 <- function(v) sqrt(sum(v^2))
# y la utilizamos igual:
modulo0(vv)

## [1] 5.385
```

Nótese cómo, de manera admirable, esta forma de programación se apega mucho más a la fórmula matemática mostrada previamente.

El problema de la media de los valores en un vector numérico es incluso más sencillo. Para el mismo vector, **v**, definido con anterioridad, la siguiente fórmula sirve para obtener la media:

$$\bar{v} = \frac{1}{n}\sum_{i=1}^{n} v_i$$

En un lenguaje de programación usual, esto se haría más o menos así:

```
# Primeramente construyamos un vector con elementos numericos
# arbitrarios, con un generador de numeros aleatorios
# (distribucion uniforme), entre 10.5 y 40.8, generamos 32
# numeros, asi:
nums <- runif(32, 10.5, 40.8)
suma <- 0  # Iniciamos la suma como cero
for (elt in nums) {
    suma <- suma + elt  # se agrega cada elemento
}
# Se calcula e imprime el promedio
(promedio <- suma/length(nums))

## [1] 28.66
```

Esta misma operación se puede hacer con la función de agregación `sum()`, cuyo funcionamiento se explicó anteriormente:

```
(promedio <- sum(nums)/length(nums))

## [1] 28.66

# Podemos convertir estas operaciones en una funcion, asi:
haz_promedio <- function(v) {
    sum(v)/length(v)
}
# o en una sola linea:
haz_promedio <- function(v) sum(v)/length(v)
# Y llamamos a la funcion:
haz_promedio(nums)

## [1] 28.66
```

Desde luego que un lenguaje como R, cuenta ya con una función que permite hacer este cálculo en un solo *disparo*: la función `mean()`.

```
mean(nums)

## [1] 28.66
```

La función `sum()` es un ejemplo de funciones que toman un objeto estructurado de R y operan sobre los elementos de éste. Otro ejemplo es la función `prod()`, cuya operación es similar a la anterior pero usando la multiplicación en vez de la suma. La pregunta inmediata que surge aquí es si existe en el lenguaje alguna función que permita construir funciones de este tipo pero para cualquier operador o función, y la respuesta es que sí: la función `Reduce()` tiene precisamente este propósito. En seguida se muestra como se reprograma-

rían las funciones `sum()` y `prod()`, por medio de `Reduce()`[3]:

```
miSum <- function(v) Reduce("+", v)
miProd <- function(v) Reduce("*", v)
# Para comparar:
prod(vv)

## [1] -24

miProd(vv)

## [1] -24
```

La aplicación de la función Reduce(), no está limitada a operadores, de hecho, la operación puede estar definida por cualquier función de dos argumentos, y, opcionalmente, puede además entregar el arreglo de resultados parciales, cada vez que se aplica la operación. Veamos el siguiente ejemplo:

```
mif <- function(a, b) {
    return(a * b/(a + b))
}
#
miOp <- function(v) Reduce(mif, v)  # Note: mif va sin comillas
# Apliquemos:
miOp(vv)

## [1] 12

# Otra version, con resultados parciales y en la que incluso
# la funcion se da 'en linea':
miOpV2 <- function(v) Reduce(function(a, b) a * b/(a + b), v,
    accumulate = TRUE)
miOpV2(vv)

## [1] -2 -6 12
```

Nótese que con el argumento `accumulate=TRUE`, la función entrega lo que la operación da cada vez que se añade un elemento del objeto a la operación.

Esta misma idea de operar con los elementos de los vectores, puede ser llevada más allá, a operar con los elementos de datos más estructurados como pueden ser las listas y los *data frames*. Para ello, el lenguaje cuenta con un conjunto de funciones de transformación, clasificación y agregación de datos que se irán revisando en los siguientes párrafos con ejemplos ilustrativos de su uso.

[3]En su forma más simple, la función Reduce(), toma una función u operación binaria y un objeto compuesto, y la aplica consecutivamente entre el resultado anterior y el siguiente elemento, tomando inicialmente como primer resultado el primer elemento del objeto.

4.3.2. Las funciones `sapply()` y `lapply()`

Para empezar, supóngase que se tiene un *data frame*, exclusivamente de columnas numéricas y se quiere conocer el promedio de cada una de estas columnas. En este caso, la función `sapply()` permite aplicar una operación o una función a cada uno de los elementos la lista o data frame, dado como argumento. Así, la operación deseada se obtiene de la siguiente manera.

```
(misDatos <- data.frame(uno = runif(5, 10.5, 40.3), dos = runif(5),
    tres = runif(5, 155, 890)))

##     uno    dos  tres
## 1 17.66 0.7023 433.0
## 2 33.04 0.8633 875.6
## 3 38.24 0.9655 283.5
## 4 27.80 0.1671 156.1
## 5 11.53 0.2866 464.6

sapply(misDatos, haz_promedio, simplify = TRUE)

##     uno     dos    tres
##  25.652   0.597 442.572

# aqui se podria haber usado la funcion 'mean' en vez de
# 'haz_promedio'
```

El argumento opcional `simplify`, especificado aquí como `TRUE`, obliga a que el resultado, si se puede, sea entregado como un vector, con un elemento correspondiente a cada una de las columnas en este caso, de otra manera, el resultado es entregado como una lista. El resultado obtenido con la función lapply es más o menos similar, pero el resultado se entrega siempre en una lista:

```
lapply(misDatos, mean)

## $uno
## [1] 25.65
##
## $dos
## [1] 0.597
##
## $tres
## [1] 442.6

# notese que aqui se uso la funcion 'mean' en vez de
# 'haz_promedio'
```

El uso de las funciones `sapply()` y `lapply()` mostrado arriba, es equivalente a la siguiente versión más *procedural*.

```
r <- numeric()  # vector numerico vacio para resultados
for (elt in misDatos) {
    r <- c(r, haz_promedio(elt))  # note el uso de c()
}
names(r) <- names(misDatos)  # Esto solo es 'azucar' estetica
r

##     uno     dos    tres
##  25.652   0.597 442.572
```

4.3.3. Operaciones marginales en matrices y la función `apply()`

Como objetos numéricos, las matrices resultan útiles para hacer diversidad de cálculos. Una de sus principales características es que tanto sus renglones como sus columnas pueden ser tratados como elementos individuales. De esta forma, hay operaciones que se efectúan para todas sus columnas o para todos sus renglones; a estas se les denominará *operaciones marginales*. El lenguaje tiene algunas de estas operaciones implementadas directamente, entre ellas están las funciones: `rowSums()`, `colSums()`, `rowMeans()` y `colMeans()`. Para ejemplificar, se calcularán las sumas de las columnas de una matriz y los promedios de sus renglones.

```
# Se hace una matriz arbitraria de 3 renglones y 5 columnas,
# con rbind, que la construye por renglones:
(mm <- rbind(5:9, runif(5, 10, 20), c(2, -2, 1, 6, -8)))

##       [,1]  [,2]  [,3]  [,4]  [,5]
## [1,]  5.00  6.00  7.00  8.00  9.00
## [2,] 18.76 17.41 10.05 18.43 17.36
## [3,]  2.00 -2.00  1.00  6.00 -8.00

colSums(mm)

## [1] 25.76 21.41 18.05 32.43 18.36

rowMeans(mm)

## [1]  7.0 16.4 -0.2
```

Estas operaciones, que se tienen implementadas directamente, evitan mucha programación que se tendría que hacer con ciclos y operaciones individuales con los elementos de la matriz.

El lenguaje provee además la posibilidad de construir el mismo tipo de operación marginal para el caso general de cualquier función, mediante la función `apply()`. Por ejemplo, la función `sd()`, calcula la desviación estándar para un conjunto de números dados como un vector numérico. Si se quisiera aplicar

esta función a todos los renglones o a todas las columnas de la matriz, y considerando que la matriz tiene dos margenes: el de los renglones -primer índice: número de margen 1-, y el de las columnas -segundo índice: número de margen 2-, el asunto se puede resolver así:

```
# para los renglones
apply(mm, 1, sd)  # el 1 es el margen: renglones

## [1] 1.581 3.603 5.215

# para las columnas
apply(mm, 2, sd)  # el 2 es el margen: columnas

## [1]  8.940  9.754  4.606  6.673 12.925
```

Para la función `apply()`, no es necesario que la función a aplicar esté predefinida, ni que el resultado de esa función sea un único número. Para clarificar esto, se propone el siguiente ejemplo:

```
# una funcion que toma un vector arbitrario y multiplica cada
# elemento, por su posicion en el vector:
ff <- function(v) v * (1:length(v))
# probemos la funcion:
ff(c(2, 4, 6, 8))

## [1]  2  8 18 32

# Ahora la aplicaremos a todas las columnas de la matriz
apply(mm, 2, ff)

##       [,1]  [,2]  [,3]  [,4]   [,5]
## [1,]  5.00  6.00  7.00  8.00   9.00
## [2,] 37.53 34.82 20.11 36.85  34.73
## [3,]  6.00 -6.00  3.00 18.00 -24.00
```

El resultado ha sido una matriz, pero cada una de las columnas de esta nueva matriz es el resultado de aplicar la funcion `ff()`, recién creada, a cada una de las columnas de la matriz original.

4.3.4. Clasificaciones y uso de la función `split()`

Generalmente, aunque los datos con los que se quiere trabajar están dados en tablas, no están organizados de la manera que se pretende trabajar sobre ellos. La función `split()`, permite clasificar los datos, tipicamente dados como un vector, o como un *data frame*. Para explicar el uso de esta función se recurrirá a un ejemplo. Supóngase que los datos, que se muestran en la Fig. 4.1, se tienen en un archivo separado por comas: Precipitaciones.csv. Este archivo contiene

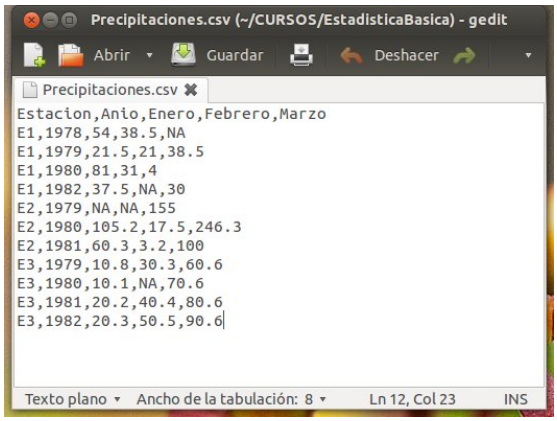

Figura 4.1: Archivo que contiene datos de precipitación en distintas estaciones

los datos de precipitación para distintas estaciones, para distintos años y para distintos meses.

La lectura de un archivo de esta naturaleza hacia un *data frame*, se hace por medio de la función `read.csv()`, como se muestra a continuación:

```
tt <- read.csv("Precipitaciones.csv")
tt

##    Estacion Anio Enero Febrero Marzo
## 1        E1 1978  54.0    38.5    NA
## 2        E1 1979  21.5    21.0  38.5
## 3        E1 1980  81.0    31.0   4.0
## 4        E1 1982  37.5      NA  30.0
## 5        E2 1979    NA      NA 155.0
## 6        E2 1980 105.2    17.5 246.3
## 7        E2 1981  60.3     3.2 100.0
## 8        E3 1979  10.8    30.3  60.6
## 9        E3 1980  10.1      NA  70.6
## 10       E3 1981  20.2    40.4  80.6
## 11       E3 1982  20.3    50.5  90.6
```

Supóngase ahora que se desea tener la misma información pero clasificada por años. Con este propósito se puede emplear la función `split()`, que recibe básicamente dos argumentos principales: el objeto a ser clasificado -típicamente, un *data frame* o un vector-, y una serie de datos que servirán para la clasificación -típicamente, un factor o un vector numérico o de cadenas de caracteres-. Este último argumento debe tener la misma longitud que una columna del *data frame*, dado como primer argumento, o que el vector dado como argumento. según el caso, y no es necesario que este argumento sea parte del objeto dado como primer argumento. Si, como primer argumento de la función

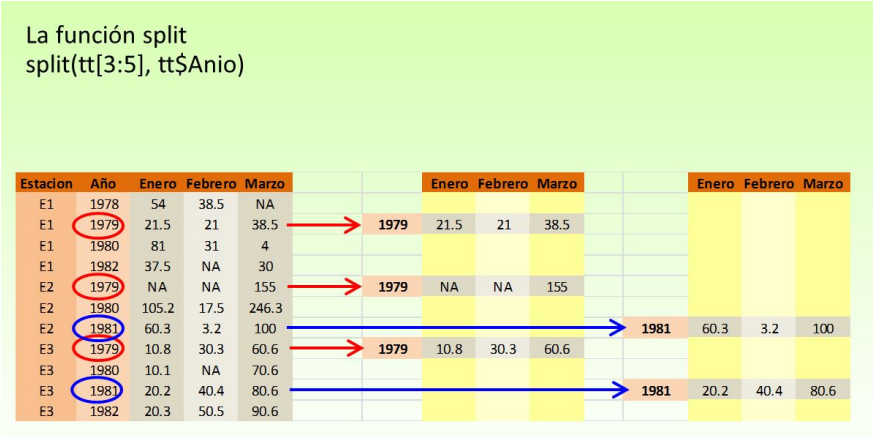

Figura 4.2: Operación de la función `split()`

pasamos una porción de la tabla contenida en el *data frame* justamente leído -sólo las columnas 3 a la 5 del *data frame*-, y como segundo argumento pasamos la columna correspondiente a los años, la función operaría como se muestra en la Fig 4.2.

El resultado de la función, en el caso del ejemplo, es una lista de tablas, cada una correspondiente a cada uno de los años registrados en la columna `tt$Anio`, como se muestra a continuación:

```
mls <- split(tt[, 3:5], tt$Anio)  # Notese: tt[, 3:5] == tt[3:5]
mls

## $`1978`
##   Enero Febrero Marzo
## 1    54    38.5    NA
##
## $`1979`
##   Enero Febrero Marzo
## 2  21.5    21.0  38.5
## 5    NA      NA 155.0
## 8  10.8    30.3  60.6
##
## $`1980`
##   Enero Febrero Marzo
## 3  81.0    31.0   4.0
## 6 105.2    17.5 246.3
## 9  10.1      NA  70.6
##
## $`1981`
##    Enero Febrero Marzo
```

```
## 7   60.3     3.2 100.0
## 10  20.2    40.4  80.6
##
## $`1982`
##    Enero Febrero Marzo
## 4   37.5      NA  30.0
## 11  20.3    50.5  90.6
```

A cada una de las tablas correspondientes a cada año se tiene acceso, de la manera habitual para listas, por ejemplo, la tabla correspondiente al año 1980 es:

```
mls$"1980"
```

```
##   Enero Febrero Marzo
## 3  81.0    31.0   4.0
## 6 105.2    17.5 246.3
## 9  10.1      NA  70.6
```

Solo para mostrar la potencia del lenguaje con estas operaciones, suóngase que se quiere aplicar una operación a cada una de las tablas de la lista resultante, el promedio por columnas o la suma por renglones, por ejemplo. Dado que se tata de una lista, podemos utilizar cualquiera de las funciones `lapply()` o `sapply()`, como se muestra a continuación.

```
lapply(mls, colMeans)
```

```
## $`1978`
##   Enero Febrero   Marzo
##    54.0    38.5      NA
##
## $`1979`
##   Enero Febrero   Marzo
##      NA      NA    84.7
##
## $`1980`
##   Enero Febrero   Marzo
##   65.43      NA  106.97
##
## $`1981`
##   Enero Febrero   Marzo
##   40.25   21.80   90.30
##
## $`1982`
##   Enero Febrero   Marzo
##    28.9      NA    60.3
```

```
lapply(mls, rowSums)

## $`1978`
##  1
## NA
##
## $`1979`
##     2     5     8
##  81.0    NA 101.7
##
## $`1980`
##   3   6   9
## 116 369  NA
##
## $`1981`
##     7    10
## 163.5 141.2
##
## $`1982`
##     4    11
##    NA 161.4
```

Obsérvese que muchos de los resultados anteriores son `NA`, esto es, *no disponibles*, debido a que las funciones `sum()` y `mean()`, y por consiguiente las funciones derivadas: `colMeans()` y `rowSums()`, entregan ese resultado cuando alguno de los elementos del vector dado como argumento es `NA`. Esto se puede resolver, indicándole a cualquiera de esas funciones hacer caso omiso de los valores `NA`, mediante el parámetro `na.rm=TRUE`, como se muestra a continuación.

```
sapply(mls, colMeans, na.rm = T)  # recuerde T es TRUE

##         1978  1979   1980  1981 1982
## Enero   54.0 16.15  65.43 40.25 28.9
## Febrero 38.5 25.65  24.25 21.80 50.5
## Marzo    NaN 84.70 106.97 90.30 60.3

class(sapply(mls, colMeans, na.rm = T))

## [1] "matrix"
```

Una nota final aquí es que en este último ejemplo se ha utilizado la función `sapply()`, en vez de la función `lapply()`. Como se puede ver, en este caso el resultado ha sido simplificado lo más posible, entregando una matriz en vez de una lista con cada uno de los resultados. Obsérvese también que para el mes de marzo y el año 1978, la tabla ha arrojado un resultado `NaN` (*not a number*), aunque se ha instruído hacer caso omiso de los valores `NA` en los cálculos

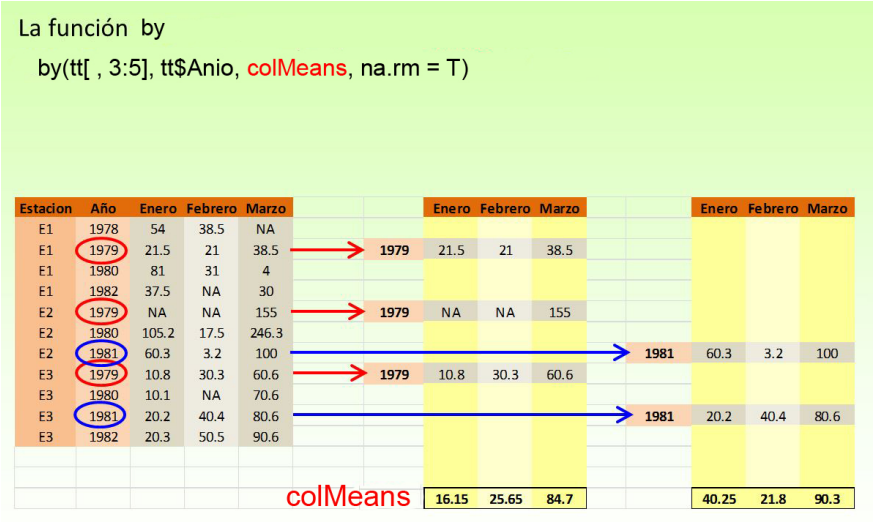

Figura 4.3: Operación de la función `by()`

involucrados. Esto se debe a que para ese año sólo ha reportado valores una estación, 'E1', pero para el mes de marzo no se ha reportado ningún valor, lo que se indica con `NA`, en la celda correspondiente de la tabla; en esa situación, no hay forma de hacer el cálculo de la media, pues se requeriría contar al menos con un valor.

4.3.5. Clasificación y operación: las funciones `by()`, `aggregate()` y `tapply()`

Los últimos ejemplos de la sección anterior, 4.3.4, muestran una secuencia de acciones separadas, a saber: primero una clasificación de los datos, para después aplicar una operación sobre cada una de las partes encontradas mediante la clasificación. De este modo, se aplicó primeramente la función `split()`, para clasificar la información, y después, sobre el resultado de esta clasificación, se aplicó cualquiera de las funciónes `lapply()` o `sapply()`, para distribuir una operación, `colMeans()`, sobre cada una de las partes de la clasificación resultante. Mediante la función `by()`, el lenguaje provee una manera de hacer estas dos acciones simultáneamente. La Fig. 4.3 muestra esquemáticamente la manera en la que opera esta función.

La función `by()`, básicamente tiene tres argumentos principales: el objeto al cual se aplica -típicamente un *data frame*-, el factor o vector usado para clasificar y la función que se aplica a cada uno de los elementos resultantes de la clasificación, y puede además tener argumentos adicionales, tales como `na.rm`, que serán pasados a la función que se aplicará. En seguida se muestra el resultado de aplicarla tal como se ha mostrado en la Fig. 4.3.

```
(rr <- by(tt[, 3:5], tt$Anio, colMeans, na.rm = T))
```

```
## tt$Anio: 1978
##   Enero Febrero   Marzo
##    54.0    38.5     NaN
## ------------------------------------------------
## tt$Anio: 1979
##   Enero Febrero   Marzo
##   16.15   25.65   84.70
## ------------------------------------------------
## tt$Anio: 1980
##   Enero Febrero   Marzo
##   65.43   24.25  106.97
## ------------------------------------------------
## tt$Anio: 1981
##   Enero Febrero   Marzo
##   40.25   21.80   90.30
## ------------------------------------------------
## tt$Anio: 1982
##   Enero Febrero   Marzo
##    28.9    50.5    60.3
```

```
class(rr)
```

```
## [1] "by"
```

Nótese la manera en la que la función ha presentado su salida: se muestra un resultado para cada uno de las clasificaciones encontradas; de hecho, la clase de objeto que arroja está diseñada ex profeso para el resultado de la función. La pregunta aquí es: ¿cómo se utiliza o se tiene acceso a la información de ese resultado? La respuesta es que se utilizan como índices los elementos que dieron lugar a la clasificación, en este caso los años, como si se tratara de una lista. Por ejemplo, para tener acceso a la información correspondiente al año 1981, se puede hacer de las siguientes maneras:

```
# Primera forma:
rr$"1981"
```

```
##   Enero Febrero   Marzo
##   40.25   21.80   90.30
```

```
# Segunda forma:
rr[["1981"]]
```

```
##   Enero Febrero   Marzo
##   40.25   21.80   90.30
```

```
# Tercera forma:
rr["1981"]

## $`1981`
##   Enero Febrero   Marzo
##   40.25   21.80   90.30
```

Para ver el significado de cada una de estas formas de acceso, se recomienda leer el texto correspondiente a listas en la sección 2.5.1 en la página 32 y el uso del operador `[]` en la sección 3.2 en la página 42.

Una forma diferente de realizar la misma tarea del ejemplo anterior, es mediante la función `aggregate()`, como se muestra a continuación.

```
(rr <- aggregate(tt[, 3:5], list(anio = tt$Anio), mean, na.rm = T))

##   anio Enero Febrero Marzo
## 1 1978 54.00   38.50   NaN
## 2 1979 16.15   25.65  84.7
## 3 1980 65.43   24.25 107.0
## 4 1981 40.25   21.80  90.3
## 5 1982 28.90   50.50  60.3

class(rr)

## [1] "data.frame"
```

Esta función, toma en este caso varios argumentos. En la forma mostrada, el primer argumento es un *data frame*, o una porción de éste, sobre cuyas columnas se aplicará la función que se indica en el tercer argumento. En el caso del ejemplo, la entrada es la porción del *data frame* correspondiente a las columnas 3 a la 5, o sea las columnas de datos de precipitación de los meses. El segundo argumento es una lista, cada uno de cuyos elementos es un objeto, que pueda ser interpretado como factor y que tiene tantos elementos como renglones tiene el *data frame* indicado antes; es decir, cada uno de estos objetos es *paralelo* al *data frame* del primer argumento. Los elementos en este argumento sirven para clasificar los renglones del *data frame*. Imaginemos que esta lista fuera de n-arreglos paralelos al frame del primer argumento. Cada índice en los arreglos indica una n-tupla o renglón, y cada una de estas determinaría la categoría o clase de renglón en el *data frame*. En el caso del ejemplo la lista es de un sólo elemento que es arreglo o vector con los años. Aquí, por ejemplo, el año 1979 ocurre tres veces, que son los elementos 2, 5 y 8; lo que indica que los renglones 2, 5 y 8 son, por así decir, la "clase 1979". El tercer argumento es la función que se aplicará, en este caso se trata de la función `mean()`. El cuarto argumento indica que hacer con los valores `NA`; en este caso se indica que sean removidos. La función se aplica sobre los elementos de una misma clase, y respetando las columnas. Así, que para el ejemplo mostrado, el promedio que se calculará pa-

ra el año 1979 en el mes de enero, considera sólo dos elementos, a saber: 21.5 y 10.8, dando como resultado 16.15.

Nótese que, a diferencia de la función `by()`, para la función `aggregate()` se ha pasado como función de operación sobre el primer argumento la función `mean()` en vez de la función `rowMeans()`. Esto se debe a que `by()` actúa como si operara con la salida de una clasificación hecha con `split()`, que es una lista con cada resultado de la clasificación, mientras que `aggregate()` opera directamente sobre las columnas del *data frame*. También, como se puede ver, el resultado de `aggregate()`, convenientemente es una tabla de datos vertida en un *data frame*, que es fácilmente manipulable como se ha mostrado antes.

Finalmente, en este tipo de funciones está la función `tapply()`, que efectua el mismo tipo de operaciones de clasificación y operación, pero actua sobre un vector y no sobre un *data frame*. Para ejemplificar el uso de esta función, se leerá un archivo que contiene la misma información de precipitaciones que se ha manejado en los ejemplos anteriores, pero organizada, o más bien, se debería decir, *desorganizada* de una manera diferente.

```
dd <- read.csv("Precipitaciones2.csv")
dd
```

```
##    Est Anio Mes  Prec
## 1   E3 1980 Mar  70.6
## 2   E1 1978 Ene  54.0
## 3   E1 1979 Feb  21.0
## 4   E3 1979 Mar  60.6
## 5   E1 1980 Mar   4.0
## 6   E3 1981 Mar  80.6
## 7   E1 1978 Feb  38.5
## 8   E3 1982 Ene  20.3
## 9   E1 1982 Feb    NA
## 10  E2 1979 Mar 155.0
## 11  E1 1978 Mar    NA
## 12  E2 1980 Ene 105.2
## 13  E1 1979 Mar  38.5
## 14  E2 1981 Mar 100.0
## 15  E1 1982 Ene  37.5
## 16  E3 1981 Ene  20.2
## 17  E2 1981 Feb   3.2
## 18  E3 1979 Ene  10.8
## 19  E2 1980 Feb  17.5
## 20  E3 1980 Feb    NA
## 21  E2 1979 Ene    NA
## 22  E1 1982 Mar  30.0
## 23  E1 1980 Ene  81.0
## 24  E3 1979 Feb  30.3
## 25  E3 1981 Feb  40.4
```

```
## 26  E2 1980 Mar 246.3
## 27  E3 1982 Feb  50.5
## 28  E2 1981 Ene  60.3
## 29  E3 1982 Mar  90.6
## 30  E1 1979 Ene  21.5
## 31  E1 1980 Feb  31.0
## 32  E2 1979 Feb    NA
## 33  E3 1980 Ene  10.1
```

Cada una de las columnas del *data frame* leído, es o bien un vector numérico, o un factor. Debido a la organización de la tabla anterior, para obtener mismo el resultado que las funciones usadas anteriormente, `by()` y `aggregate()`, la función `tapply()` tiene que sacar provecho de que el segundo argumento, que sirve para clasificar los datos, puede ser una lista de uno o mas objetos, interpretables como factores, y que en el caso del ejemplo estará constituída por las columnas correspondiente a los años y a los meses. Así, el resultado deseado se obtiene como sigue:

```
(rr <- tapply(dd$Prec, list(dd$Anio, dd$Mes), mean, na.rm = T))
```

```
##        Ene   Feb   Mar
## 1978 54.00 38.50   NaN
## 1979 16.15 25.65  84.7
## 1980 65.43 24.25 107.0
## 1981 40.25 21.80  90.3
## 1982 28.90 50.50  60.3
```

La posibilidad de la función `tapply()` de usar como clasificador una lista de objetos interpretables como factores, también está disponible para las otras funciones que utilizan este esquema de clasificación: `split()`, `by()` y `aggregate()`. Para más detalles, consulte las páginas de ayuda, introduciendo en su intérprete, p.ej., ‘‘`?aggregate`’’.

Como se puede ver por lo expuesto en este capítulo, mucho del trabajo que otros lenguajes resuelven con ciclos, decisiones y secuencias de instrucciones, en R, alternativamente, se puede hacer por medio de este elegante conjunto de funciones de clasificación, transformación y agregación de datos, lo que, junto con las operaciones para manipular porciones de los datos estructurados, revisadas en el capítulo anterior, hacen del lenguaje una poderosa herramienta para el procesamiento de información.

En relación con las funciones disponibles en el lenguaje, tales como las trigonométricas, logarítmicas, exponenciales, de distribución de probabilidades, etc., debido a la integración de infinidad de paquetes sobre muy diversos tópicos en el lenguaje, se invita al lector a considerar las fuentes de información contenidas en el intérprete del lenguaje, mediante la instrucción “`??functions`”, que mostrará todo lo que se encuentra bajo ese rubro en el módulo de ayuda del intérprete, y en Internet, la dirección:

`http://cran.r-project.org/web/packages/`, donde se encuentra información de todos los paquetes que se pueden instalar en el intérprete de R.

Capítulo 5

Escritura de Funciones

En los capítulos anteriores se han revisado distintos tópicos de la programación con este lenguaje. En medio de las explicaciones provistas, se ha indicado implícitamente la forma de escribir funciones; en particular, en la sección 2.7 en la página 37, se ha dado una breve introducción al tema de la escritura de funciones. En este capítulo se abordará ese mismo tema, pero de una manera un poco más formal.

5.1. Estructura formal de una función

Como se ha visto con anterioridad, R trata las funciones prácticamente igual que cualquier otra variable. Así, ellas se pueden manipular de manera semejante a como se hace con otros objetos de R: se pueden pasar como argumentos de otras funciones, se pueden regresar como el valor final de una función, se pueden definir en el interior de otra función, etc. Para crear o definir una función, se emplea la directiva ‘‘`function`’’, en general, asociándola con un símbolo mediante una operación de asignación, como se muestra en la Fig. 5.1. Esta directiva, tiene dos partes, la definición de los *argumentos formales* de la función, y el cuerpo de la función. El cuerpo de la función está constituído por una o más expresiones válidas del lenguaje. Al ejecutarse la función, el valor de la última expresión en la secuencia de ejecución, es el valor que la función entrega como resultado; esto simbólicamente se ha indicado en la figura, como la última expresión del cuerpo, aunque, como se verá más adelante no necesariamente es así.

Existen dos *momentos* importantes en la *vida* de una función: la definición de la función, que básicamente ocurre una vez, y la ejecución de la función, que puede ocurrir un sinnúmero de ocasiones y en contextos muy diversos. La nociones de argumentos formales, variables y valor de regreso en el interior de una función, deben ser examinadas desde la perspectiva de esos dos momentos.

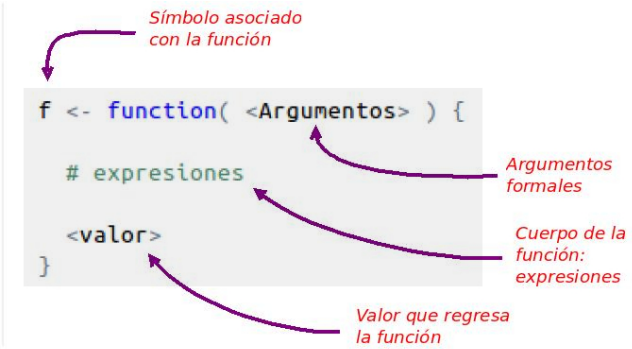

Figura 5.1: Definición de una función

5.1.1. Argumentos y valor de resultado de una función

Los argumentos de una función se enfocan desde las perspectivas de los dos momentos de la vida de una función: su definición y su ejecución. En el momento de la definición de la función se ven principalmente como lo que se denomina *argumentos formales*, mientras que en el momento de la ejecución, se considera cómo esos argumentos formales se asocian con un valor particular a partir de sus respectivos *argumentos verdaderos* o efectivos, que son los que se especifican o se *pasan*, por así decirlo, en el instante de invocar la función.

Los argumentos formales son uno de los principales medios por el cual la función se comunica con el ambiente exterior a ella; esto es, el sitio dónde la función es invocada para ejecutarse, y consiste de una lista de símbolos, que pudieran tener o no asociado algún valor a utilizar en caso de no asignársele alguno en el momento de su invocación. Al momento de ejecutarse la función, los argumentos formales se asociarán con algún valor, típicamente, a partir de los *argumentos verdaderos* que se proveen en la invocación de la función, o de los *valores por omisión* o de *default*, que se hayan indicado en la definición de la función. Veamos un ejemplo.

```
# Definición -- Versión 1
MiFunc.v1 <- function (x, yyy, z=5, t) {
  w <- x + yyy + z
  w
}

# Definición -- Versión 2
MiFunc.v2 <- function (x, yyy, z=5, t) {
```

```
  w <- x + yyy + z
  return (w)
  3.1416 # Este código nunca se ejecuta
}

# Definición -- Versión 3
MiFunc.v3 <- function (x, yyy, z=5, t) {
  x + yyy + z
}

# Ejecuciones:

MiFunc.v1(1,2,3) # Ejecución 1

## [1] 6

MiFunc.v2(1,2) # Ejecución 2

## [1] 8

MiFunc.v3(1) # Ejecución 3

## Error: el argumento "yyy" está ausente, sin valor por omisión

MiFunc.v3(z=3, yyy=2, x=1) # Ejecución 4

## [1] 6

MiFunc.v2(1, y=2) # Ejecución 5

## [1] 8

MiFunc.v1(1, z=3) # Ejecución 6

## Error: el argumento "yyy" está ausente, sin valor por omisión
```

En el código precedente, se han definido tres versiones de una función que tiene la simple tarea de sumar tres números que estarán asociados con los tres argumentos formales x, yyy, z; aunque, intencionalmente se ha provisto de un argumento adicional, t, que no se emplea en el cuerpo de la función. La lista de argumentos formales es semejante en todas las versiones. En ella, al tercer argumento, z, mediante el operador ‘‘=’’, se le asigna el valor 5 por omisión; lo que quiere decir que si no se puede asociar con un argumento verdadero al momento de la ejecución, tomará ese valor. En todas las versiones se hace la suma de los tres primeros argumentos formales; sin embargo, en las dos primeras versiones, el resultado de esta operación se *guarda* en una variable local, w, mientras que en la tercer versión, no. El resultado de la función, esto es el valor que entregará al ejecutarse, se toma de la última expresión ejecuta-

da en el cuerpo de la función. En el caso de la primer versión, el resultado es el valor que tenga la variable `w` al momento de ejecutarse, mientras que en la tercera versión, es directamente el resultado de la suma de los tres argumentos formales. La segunda versión difiere de las otras dos en que se invoca la función `return()`, que tiene dos consecuencias: la ejecución de la función se interrumpe en ese punto y el resultado de la función será el argumento que se pase a `return()`. En el caso del ejemplo, para esta tercera versión, el resultado será el valor de `w`, mientras que la expresión `3.1416`, nunca se ejecutará, y por consiguiente el resultado de la función nunca será ese.

Al momento de ejecutarse la función, los argumentos formales se asocian con o toman sus valores de los argumentos verdaderos. Una forma de hacer esto es mediante el orden o la posición en que aparecen en la definición, y a la que denominaremos *asociación posicional*. Las primeras tres ejecuciones del código anterior, hacen uso de esta característica. En la primera, de acuerdo con el orden, los argumentos `x`, `yyy`, `z`, quedan asociados con los valores 1, 2, 3, respectivamente, que son los argumentos verdaderos, y se ha omitido especificar algún valor para el cuarto argumento, `t`. R no tiene ningún problema con esa omisión, ya que `t` no se usa en ninguna parte del cuerpo de la función, en ninguna de sus versiones[1]. En la segunda ejecución, se ha omitido además especificar el argumento verdadero correspondiente a `z`. Nuevamente, el lenguaje no tiene ningún problema con eso, ya que en la definición se ha provisto el valor 5 en caso de omisión, por lo que en esa ejecución, `z` toma ese valor y por consiguiente el resultado de la suma es 8. En la tercera ejecución, se ha omitido además especificar el argumento verdadero correspondiente a `y`. Como se puede ver, en este caso el intérprete sí se queja, ya que no se ha especificado ningún valor por omisión para dicho argumento.

La otra manera de asociar los argumentos formales con sus correspondientes valores determinados por los argumentos verdaderos al momento de la ejecución, es mediante sus nombres. A esta manera la denominaremos *asociación nominal*. Las ejecuciones 4, 5 y 6, del código anterior y que son paralelas a las primeras tres, hacen uso de esta característica. En este caso, no es necesario respetar el orden en que fueron especificados los argumentos formales, aunque, como se puede ver en la ejecuciones 5 y 6, se pueden combinar ambas maneras; esto es, para algunos argumentos emplear la asociación posicional, y para otros, la asociación nominal. Una nota adicional es que, en este tipo de asociación, como se puede ver en la ejecución 5, el lenguaje puede resolver la asociación con un nombre de manera *parcial*; por ello es que satisfactoriamente resuelve que, en la invocación, la especificación ‘‘`y=2`’’ es lo mismo que ‘‘`yyy=2`’’, y que, de hecho, se podría también haber especificado como ‘‘`yy=2`’’. Esto se puede hacer, siempre y cuando la expresión especificada pueda ser resuelta por este método sin ambigüedad.

[1] Esto tiene que ver con lo que se conoce como la *evaluación perezosa* de los argumentos en R y que consiste en que en el momento de ejecución de una función, el lenguaje no evalúa los argumentos a no ser que la ejecución del código lo requiera. En los casos del ejemplo, como el argumento `t` no se menciona en el cuerpo de la función, nunca se requiere su evaluación y por tanto, no constituye un error el que no se haya especificado en la invocación de la función.

5.1.2. Revisión de los argumentos de una función

Una vez definida una función, hay dos funciones de R que permiten revisar su lista de argumentos formales, a saber: `args()` y `formals()`. En el siguiente código se ve el comportamiento de cada una de ellas.

```
args(MiFunc.v2)

## function (x, yyy, z = 5, t)
## NULL

(ar <- formals("MiFunc.v2")) # puede o no llevar comillas

## $x
##
##
## $yyy
##
##
## $z
## [1] 5
##
## $t

# Si se quiere revisar el argumento z, se hace así:
ar$z

## [1] 5
```

La función `args()`, entrega lo que sería el encabezado de la función, desprovisto de su cuerpo; prácticamente, permite ver la lista de argumentos de la función tal como fue definida. Esta función le resulta útil a un usuario del lenguaje para revisar la lista de argumentos de cualquier función. La función `formals()`, digiere un poco más la información ya que entrega una lista con cada uno de los argumentos formales y los valores asignados por omisión. Esta función le resulta más útil a un programador que desea manipular esta lista[2]. Estas funciones resultan útiles para revisar los argumentos de funciones de biblioteca, que pueden ser muchos, y sus valores por omisión. Por ejemplo, si se quiere revisar esto para la función `lm()`, que se usa para ajustes lineales, se puede hacer con:

```
args(lm)
```

[2]En efecto, esta lista es una clase de objeto de R conocido como "`pairlist`", cada uno de cuyos elementos tiene, por un lado, como nombre el nombre de uno de los argumentos y como valor, el valor correspondiente por omisión, *sin evaluar* aún.

```
## function (formula, data, subset, weights, na.action,
##     method = "qr", model = TRUE, x = FALSE, y = FALSE,
##     qr = TRUE, singular.ok = TRUE, contrasts = NULL,
##     offset, ...)
## NULL
```

5.1.3. El argumento especial ‘‘. . .’’

En la definición de las funciones existe un argumento especial que se puede emplear en distintas circunstancias: se trata del argumento especial ‘‘. . .’’, que sirve para transferir un número variable de argumentos a otra función. La utilidad de este argumento se da en varios casos que se discuten a continuación.

5.1.3.1. El uso del argumento ‘‘. . .’’ para extender una función

Cuando se quiere extender o modificar el comportamiento de una función invocándola desde otra función, una manera de pasar varios argumentos, a los que no se hace referencia en la nueva función, es mediante este argumento, que en este caso podría ser interpretado más o menos como la frase: *“y el resto de los argumentos”*. Para explicar como se comporta este *argumento* aquí, a continuación se propone un ejemplo sencillo.

En la sección anterior, 5.1.1 en la página 81, se codificó la función `MiFunc.v3()`. En la ejecución 3 de esta función, el intérprete se quejó porque, al no existir un valor por omisión para el argumento yyy, su valor quedó indefinido y no se pudo ejecutar la función. Supongáse que se quiere hacer una función que modifique el comportamiento de la otra función, simplemente proveyendo un valor por omisión para dicha función[3]. El código para ese propósito es el siguiente:

```
NuevaMiFunc <- function(x, yyy = -1, ...) {
    MiFunc.v3(x, yyy, ...)
}

NuevaMiFunc(1)

## [1] 5

NuevaMiFunc(x = 1)

## [1] 5

NuevaMiFunc(1, z = 10)

## [1] 10
```

[3]Desde luego que la opción inmediata sería reprogramar la función. Sin embargo, hay muchas funciones de cuyo código no se puede disponer o simplemente, la técnica que se ilustra podría servir para contar con diferentes versiones de la misma función.

Nótese que todos los otros argumentos, "el resto de los argumentos", se pasan intactos de `NuevaMiFunc()` a `MiFunc.v3()`, como se puede ver en la última ejecución del código anterior.

5.1.3.2. El uso del argumento ''. . .'' al principio de una función, cuando no se conoce de antemano el número de argumentos

Otro uso de este argumento especial es al principio de funciones para las que no se conoce de antemano el número de argumentos que les serán enviados. Uno de estos casos es la función `paste()`, que sirve para construir una cadena de caracteres a partir de un número cualquiera de cadenas de caracteres que le entreguen como argumentos. En seguida se muestra el encabezado de argumentos y el uso de dicha función.

```
args(paste)

## function (..., sep = " ", collapse = NULL)
## NULL

paste("uno", "dos", "tres", sep = "++")

## [1] "uno++dos++tres"

paste("uno", "dos", "tres", se = "++")

## [1] "uno dos tres ++"
```

El asunto aquí es que, la asociación con todos los argumentos que siguen al argumento especial "...", es *nominal* y de ninguna manera *posicional*; esto es, deben ser explícitamente nombrados si su valor se requiere al momento de ser ejecutada la función. Además, la asociación parcial de argumentos también está prohibida, como se puede ver en la última ejecución del código anterior, donde el intérprete no entendió que por `se=''++''`, se quería significar `sep=''++''`, y en vez de ello interpretó que se deseaba agregar una cadena más a la salida.

5.2. Visibilidad del código

Cada vez que se introduce un nuevo símbolo en alguna parte del código de un programa escrito en R, al ejecutarse, el lenguaje tiene que resolver de algún modo, de qué manera se asocia ese símbolo con algún objeto computacional creado o por crearse dentro del programa: ¿se refiere el símbolo a algún objeto que ya existía en el programa?, ¿es necesario crear un nuevo objeto asociado con el símbolo que se ha introducido? La visibibilidad del código, y las reglas de alcance del lenguaje R, que son los temas de esta sección, se refieren precisamente a la asociación de los símbolos con los objetos computacionales creados durante la ejecución de algún programa y muy particularmente en el tema de cómo se hace esta asociación cuando se ejecutan las funciones.

5.2.1. Asociación de símbolos con valores

La primer pregunta que surge en el asunto que se trata en este capítulo es: ¿en qué momento se introduce o viene a la existencia en R algún símbolo, típicamente para nombrar alguna variable? La respuesta a esta pregunta es que básicamente hay dos momentos:
Las expresiones de asignación. Observemos por ejemplo qué pasa con la siguiente expresión:

```
x <- y

## Error: objeto 'y' no encontrado
```

Después de la asignación, R se ha quejado de la inexistencia del símbolo ‘‘y’’, pero no se ha quejado de la inexistencia del símbolo ‘‘x’’. Esto se debe a que, el símbolo a la izquierda de la asignación se emplea para la *escritura* de un objeto al que hace referencia y, si no existe, se crea en ese mismo momento; esto equivale a lo que sería la *definición* de una variable en otros lenguajes. En cambio, el símbolo o símbolos que aparecen a la derecha son de *lectura* y por lo tanto se presupone su existencia o definición previa. En este caso, como R no puede determinar una asociación de este símbolo con algún valor, pues ‘‘y’’ no se ha creado previamente, emite un mensaje de error como el mostrado. Así pues, los símbolos en el lado izquierdo de las expresiones de assignación pueden significar la creación de nuevas variables referidas mediante esos símbolos, en el código, a partir de ese momento. Dado que la instrucción anterior ha terminado en error, tampoco se ha asociado el símbolo ‘‘x’’ con valor alguno; esto es, no se ha creado ningún objeto con ese *nombre*.
La declaración de argumentos formales de alguna función. Otra manera de introducir los símbolos correspondientes a *nuevas variables*, es mediante la declaración de los argumentos formales en la definición de alguna función:

```
f <- function(x, y) {
    x + y
}
```

En este ejemplo, el símbolo ‘‘f’’, se ha introducido por la asignación, y los símbolos ‘‘x’’ y ‘‘y’’, se han introducido mediante su declaración como argumentos formales de la función, y cuya validez está delimitada al código de la función.

Pero, ¿dónde y cómo se organizan esas asociaciones entre símbolos y valores? Para introducir esta noción, se asignará al símbolo ‘‘t’’, una función muy simple:

```
t <- function(x) {
    2*x
}
```

```
# Veamos cual es el valor de la
# variable 't' recien declarada:
t

## function(x) {
##     2*x
## }

# Usemos la variable:
t(8)

## [1] 16
```

Como era de esperarse, al introducir en la *consola*[4] solamente la variable ‘‘t’’, regresa su valor, que es una función y que se puede utilizar para calcular el doble de algún valor, 8, en el caso del ejemplo. Resulta, sin embargo, que, como se dijo anteriormente, en la sección 2.3.3 en la página 25, el símbolo ‘‘t’’, ya estaba de entrada asociado con la función de transposición de matrices. ¿Qué ha pasado con la definición anterior de ‘‘t’’?, ¿se ha perdido del todo? De hecho, no. Se puede recuperar la definición anterior, removiendo la que se acaba de introducir. Esto se hace de la siguiente manera:

```
rm(t) # remocion de la definicion anterior de 't'
# Ahora veamos que es 't'
t

## function (x)
## UseMethod("t")
## <bytecode: 0x29eefc8>
## <environment: namespace:base>

# y usemos nuevamente la definicion original:
mx <- cbind(c(4,5),c(6,7))
mx

##      [,1] [,2]
## [1,]    4    6
## [2,]    5    7

t(mx) # otra vez, t() es la traspuesta de una matriz

##      [,1] [,2]
## [1,]    4    5
## [2,]    6    7
```

[4]La *consola* de R, es el ambiente interactivo provisto por el lenguaje, en el cual el usuario de éste introduce las expresiones del lenguaje y recibe, después de su interpretación, las respuestas correspondientes generadas.

Nótese que al solicitar el valor del símbolo ‘‘`t`’’, esta vez, en lugar de desplegar el código de la función asociada, regresa un conjunto de información que, en efecto la identifica como una función, pero con otros datos adicionales. Ello se debe a que se trata de una función de biblioteca. Para el asunto que nos ocupa, de la información desplegada, lo que interesa es el texto: `<environment: namespace:base>`, pues es un indicador, de que el valor asociado ahora al símbolo ‘‘`t`’’, se encontraba guardado justamente *ahí*, en el ambiente (*environment* en inglés) del espacio de nombres (*namespace* en inglés), de un paquete o biblioteca llamada “base”. Pero, ¿cuántos de estos ambientes hay cuando se ejecuta un programa en R?, y ¿de cuál de ellos toma R el valor asociado a un nombre particular? La función `search()`, permitirá responder a ambas preguntas:

```
search()

## [1] ".GlobalEnv"        "package:knitr"
## [3] "package:stats"     "package:graphics"
## [5] "package:grDevices" "package:utils"
## [7] "package:datasets"  "Autoloads"
## [9] "package:base"
```

El resultado de esta función es un vector de cadenas de caracteres, cada una de las cuales es la descripción de un “ambiente”, en el que se *guardan* asociaciones entre símbolos y valores. Para saber los símbolos que contiene alguno de estos ambientes, se usa la función `ls()`, con el nombre del ambiente como argumento. Como generalmente, estos ambientes contienen una gran cantidad de símbolos, filtraremos aquí el resultado con la función `head()`, que limita el resultado a pocos elementos al inicio, seis por *default*. Así, veamos cuales son esos primeros seis símbolos para el ambiente ‘‘`package:stats`’’, y la totalidad de los símbolos en primer ambiente, ‘‘`.GlobalEnv`’’, el argumento default de `ls()`, de la lista anterior, como sigue:

```
# Introduzcamos, en la "consola", primeramente un
# simbolo cualquiera 'MiSimbolo', para la explicacion
# posterior en el texto
MiSimbolo <- 5
head( ls("package:stats") ) # Primeros seis elementos

## [1] "acf"        "acf2AR"     "add1"       "addmargins"
## [5] "add.scope"  "aggregate"

ls() # Equivale a: ls(".GlobalEnv")

## [1] "ar"          "f"           "MiFunc.v1"   "MiFunc.v2"
## [5] "MiFunc.v3"   "MiSimbolo"   "mx"          "NuevaMiFunc"
```

Todos los símbolos que se definen mediante el operador de asignación a nivel de la *consola* del intérprete de R, se introducen en el primer ambiente de

la lista `search()`; esto es, en el ambiente ‘‘`.GlobalEnv`’’. En el ejemplo anterior, se ve que el objeto recien creado asociado al símbolo `MiSimbolo`, aparece justamente en la lista correspondiente a ese ambiente.

Cuando R busca el valor asociado a un símbolo, lo hace justamente en el orden establecido por la lista entregada por la función `search()`. Esa es la razón por la cual al redefinir la función `t()`, el intérprete se encontró primero con la nueva definición provista y no con la que existía previamente en el ambiente ‘‘`package:base`’’, que, como se puede ver, es el último de la lista. De hecho, el primer ambiente siempre será el ambiente gobal ‘‘`.GlobalEnv`’’, y el último el corespondiente a ‘‘`package:base`’’.

Cada vez que se carga un nuevo paquete, mediante la función `library()`, por ejemplo, su ambiente se introduce en la segunda posición de la lista, empujando todos los que ya estaban, una posición *hacia abajo*, pero dejando siempre en primer lugar el ambiente global. Por ejemplo, si se carga el paquete para desarrollo de interfaces gráficas `tcltk`, obtenemos lo siguiente:

```
library(tcltk)
search()
```

```
##  [1] ".GlobalEnv"        "package:tcltk"
##  [3] "package:knitr"     "package:stats"
##  [5] "package:graphics"  "package:grDevices"
##  [7] "package:utils"     "package:datasets"
##  [9] "Autoloads"         "package:base"
```

Otro hecho interesante es que R mantiene por separado los espacios de nombres de las funciones y de todos los otros objetos que no son funciones. De este modo, se pudiera definir un símbolo ‘‘`t`’’, asociado a un vector, por ejemplo, y aún así tener acceso a la función `t()`, veamos:

```
t <- c(1,2,3) # definimos vector
t(mx) # usamos la funcion t
```

```
##      [,1] [,2]
## [1,]    4    5
## [2,]    6    7
```

```
t # pero al consultar t, tenemos ...
```

```
## [1] 1 2 3
```

Aunque en el ambiente global sólo existe el símbolo ‘‘`t`’’ correspondiete al vector, no obstante, R acertadamente invoca la función de transposición `t()`, por la separación de espacios de nombres entre funciones y no funciones.

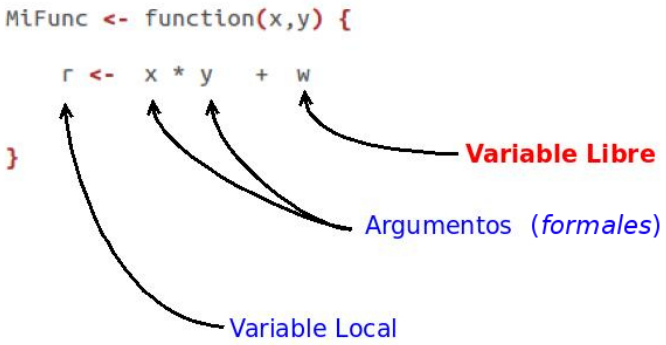

Figura 5.2: Tipos de símbolos en el interior de una función

5.2.2. Reglas de alcance

La discusión precedente lleva ahora a considerar lo que se denomina reglas de alcance, y que se refieren a la forma como el lenguaje resuelve la asociación entre una variable o símbolo libre y su correspondiente valor.

En una función, una variable libre es aquella que no se ha *definido* en el código de la misma, en el sentido explicado al principio de la sección 5.2.1, y cuyo valor, sin embargo, se solicita, al incluir su nombre o símbolo correspondiente, típicamente en el lado derecho de una asignación[5], como se muestra en la Fig. 5.2.

Para asociar las variables libres con un valor, R utiliza lo que se conoce como *alcance léxico*[6]. En este tipo de alcance se establece que *los valores de las variables se buscan en el ambiente en el cuál la función se definió*. Como se puede intuir de lo dicho antes, un ambiente es una colección de pares <*símbolo, valor*>, donde, por ejemplo, un símbolo podría ser `MiSimbolo`, y su valor `5`, o el símbolo `t` y su valor el vector `<1,2,3>`. Ahora bien, en R los ambientes se organizan en una jerarquía; esto es, cada ambiente tiene un ambiente padre y a su vez puede tener cero o más hijos. El único ambiente que no tiene padre es el ambiente vacío, que se encuentra en lo más alto de la jerarquía. Así pues, el lugar dónde R buscará el valor de una variable libre, dependerá del lugar dónde se encuentre escrito el código de la función en el que se hace referencia a ella. Tomemos, por ejemplo, el caso de la función que se ha mostrado en la Fig. 5.2, y veamos

[5]Por ejemplo, otro caso en el que esto puede ocurrir es si dentro de la función se llama a otra función pasando como argumento una variable que no ha sido previamente definida.

[6]El lenguaje S, o más bien su última versión, S-PLUS, usa lo que se conoce como *alcance estático*, que consiste en resolver el problema de las variables libres, asociándolas siempre con los nombres que aparecen en el ambiente global. La diferencia entre este tipo de alcance y el *alcance léxico*, que es el que usa R, se puede apreciar en la explicación que aparece más delante en el texto.

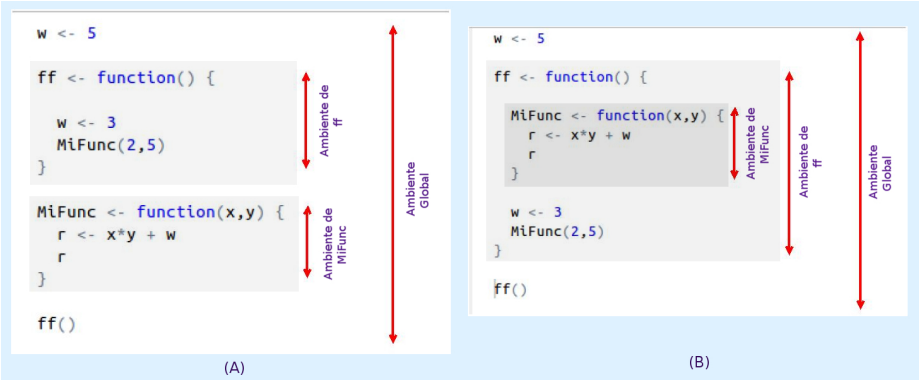

Figura 5.3: Jerarquía en los ambientes de las funciones

la diferencia en el valor que tomará la variable ‘‘w’’, al insertar el código de la función en distintas partes.

La Fig. 5.3 muestra la inserción de la misma función, `MiFunc()`, en dos distintas partes del código global. La jerarquía de los ambientes es diferente en cada caso: en la situación marcada como (A), el ambiente global tiene dos hijos el correspondiente a la función `ff()` y el correspondiente a `MiFunc()`, mientras que en la situación (B), el ambiente global tiene un solo hijo, el ambiente correspondiente a la función `ff()`, que a su vez tiene un hijo, el correspondiente a la función `MiFunc()`. Una nota adicional aquí es que una función puede ser definida en el interior de otra función y eso es precisamente lo que ilustra la situación (B) de la figura.

Para resolver el valor de ‘‘w’’, o sea de una variable libre, en cualquier caso, R busca primero en el ambiente de la misma función, si no encuentra el símbolo ahí, procede a buscar en el ambiente padre, y así lo sigue haciendo con todos los predecesores hasta encontrar una asociación. Por supuesto que, si en este proceso, llega al ambiente vacío sin encontrar una asociación posible, el lenguaje emitirá un mensaje de error.

En el código que sigue se muestra el comportamiento de R las situaciones mostradas en la Fig. 5.3, y se añade una situación más para ejemplificar lo que se ha explicado aquí.

```
w <- 5 # Primera w

ff <- function() {
  w <- 3 # Segunda w
  MiFunc(2,5)
}

MiFunc <- function(x,y) {
  r <- x*y + w
```

```
  r
}

ff()

## [1] 15
```

En este caso, a pesar de que en el interior de la función `ff()` se ha definido un valor para `w` de 3, el lenguaje ha resuelto, de acuerdo con la regla explicada previamente, que el valor de `w` en `MiFunc()` es el del ambiente global, esto es, 5, y por ello el resultado que se despliega es 15.

```
w <- 5 # Primera w

ff <- function() {
  MiFunc <- function(x,y) {
    r <- x*y + w
    r
  }

  w <- 3 # Segunda w
  MiFunc(2,5)
}

ff()

## [1] 13
```

En este caso, la asociación del símbolo `w`, referido en el interior de la función `MiFunc()`, con el valor 3, es inmediata, a partir de la variable `w` definida el ambiente de la función padre `ff()`, por ello el resultado es 13.

```
w <- 5 # Unica w

ff <- function() {
  MiFunc <- function(x,y) {
    r <- x*y + w
    r
  }

  # ELIMINAMOS LA DEFINICION: w <- 3
  MiFunc(2,5)
}

ff()

## [1] 15
```

En este último caso, se ha eliminado la definición de `w` en el ambiente de la función `ff()`, por ello, al no encontrar asociación posible en éste ambiente que es el padre del ambiente de la función `MiFunc()`, procede hacia arriba en la jerarquía, en este caso al ambiente global donde encuentra que `w` tiene un valor de 5. Por eso, aquí el resultado es 15.

5.3. Contenido de los ambientes de las funciones

En el ejemplo que se presentó en la sección anterior, particularmente en el caso ilustrado por la Fig. 5.3(B), el uso de la función `MiFunc()` está confinado al interior de la función `ff()`, que es donde se definió. Al ser por sí mismo `MiFunc`, el símbolo correrspondiente a una variable, cuyo valor es una función en este caso, el lenguaje R, tiene los mecanismos necesarios para revelar ese valor, o sea el código de la función, fuera de la función misma donde se definió. Esto permite usar las funciones definidas en el interior de alguna función, afuera del código correspondiente a la función donde se han definido. En este caso, las reglas de alcance, descritas previamente, juegan un papel muy importante. Para ilustrar esto, se propone aquí la creación de una función constructura de funciones, de la siguiente manera:

```
Construye.multiplicador <- function(n) {
  fff <- function(x) {
    n*x
  }
  fff # Regresa como resultado la funcion creada
}

duplica <- Construye.multiplicador(2)
triplica <- Construye.multiplicador(3)

duplica(5)

## [1] 10

triplica(5)

## [1] 15
```

La función `fff()` tiene una variable libre, a saber, `n`. Para resolver el valor de `n`, en el interior de `fff()`, el lenguaje busca en el ambiente padre, el ambiente de la función `Construye.multiplicador()`, en este caso. Al momento de la ejecución, `n`, que es un argumento formal de `Construye.multiplicador()`, toma consecutivamente los valores de 2 y 3, y esos son, respectivamente, los valores que tomará la variable libre, `n`, al momento de construir las funciones `duplica()` y `triplica()`.

En R, es posible revisar el contenido del ambiente de una función, e incluso encontrar el valor asociado a un símbolo determinado en esos ambientes. Para ello se usan las funciones `environment()`, `ls()` y `get()`, como se muestra a continuación.

```
ls( environment(duplica) )

## [1] "fff" "n"

get( "n", environment(duplica) )

## [1] 2

ls( environment(triplica) )

## [1] "fff" "n"

get( "n", environment(triplica) )

## [1] 3
```

Una nota final de este capítulo, tiene que ver con el proceso de búsqueda que sigue R más allá del ambiente global. En la sección 5.2.1 se vio la función `search()`, y la lista de ambientes que arroja como resultado. A continuación se muestra un fragmento de la línea de búsqueda en ambientes que seguiría R a partir de la función `duplica()`.

```
# El PRIMER ambiente de la busqueda
environment(duplica)

## <environment: 0x2b21310>

# El SEGUNDO ambiente de la busqueda: padre del anterior
parent.env(environment(duplica))

## <environment: R_GlobalEnv>

# .. otra vez el SEGUNDO, solo para comparar
environment(Construye.multiplicador)

## <environment: R_GlobalEnv>

# TERCER ambiente: otro nivel en la jerarquia
parent.env(parent.env(environment(duplica)))

## <environment: package:tcltk>
## attr(,"name")
## [1] "package:tcltk"
## attr(,"path")
## [1] "/usr/lib/R/library/tcltk"
```

```
# CUARTO ambiente: ...y otro nivel mas en la jerarquía:
parent.env(parent.env(parent.env(environment(duplica))))

## <environment: package:knitr>
## attr(,"name")
## [1] "package:knitr"
## attr(,"path")
## [1] "/home/checo/R/x86_64-pc-linux-gnu-library/3.1/knitr"
```

El primer ambiente, corresponde al de la función `duplica()` propiamente; la identificación que se muestra de éste, no nos dice mucho, pero creednos, se trata del ambiente de la función original. El segundo ambiente, al que se ha llegado por dos caminos distintos en el código anterior y que es el ambiente padre del primero, corresponde al de la función `Construye.multiplicador()`; se trata del ambiente global. A partir del ambiente global, los subsecuentes, son en orden los de la lista que se se puede obtener con la función `search()`. Como ejemplo, aquí sólo se han desplegado dos, el tercer y cuarto ambientes en la búsqueda, y cuya identificación se puede constatar en la lista provista por `search()`, en la sección 5.2.1 en la página 90.

5.4. Ejemplo: ajuste de datos a una función de distribución

En las secciones anteriores se ha visto a grandes rasgos los principales temas referentes a la construcción de funciones. Así, se está ya en condiciones de abordar un ejemplo que haga uso de todos estos conceptos. De este modo, en la presente sección se propone un ejemplo que tiene que ver con un interesante tema de la estadística: las funciones de densidad y distribución de probabilidades.

Para darle mayor atractivo a esta sección, se usarán, sin profundizar, algunos de los conceptos que se verán en el siguiente capítulo y que tienen que ver con el tema de la producción de gráficos en R.

5.4.1. Histogramas de frecuencias

Para entender este concepto, se introduce aquí un juego de datos correspondiente a las precipitaciones promedio acumuladas para el mes de octubre del año 1970 al año 2010, en el estado de Guerrero. La tabla correspondiente a estos datos se muestra en la Fig. 5.4.

Un histograma es una representación gráfica de la frecuencia con que ocurren los valores, categorizados como intervalos, de alguna variable aleatoria. Por ejemplo, en la tabla de la Fig. 5.4, la variable aleatoria es la precipitación, y un histograma, correspondiente a sus valores, en R se puede producir como se muestra a continuación.

año	precip	año	precip	año	precip	año	precip
1970	81.2	1981	190	1992	119.5	2003	158.3
1971	104	1982	89	1993	99	2004	144.5
1972	63.5	1983	108.4	1994	130.5	2005	99.4
1973	188.9	1984	43.8	1995	89.4	2006	196.5
1974	48.1	1985	87	1996	176.8	2007	114.3
1975	95.1	1986	64.6	1997	234.8	2008	133.2
1976	261.9	1987	26	1998	227.8	2009	136.7
1977	78.3	1988	66.5	1999	164.4	2010	22.6
1978	145.4	1989	101	2000	86.8		
1979	54.8	1990	131.5	2001	64.7		
1980	76.7	1991	118.1	2002	111.4		

Figura 5.4: Precipitaciones promedio acumuladas en el mes de octubre en el estado de Guerrero en mm

Primeramente, si se supone que la información se encuentra en un archivo de nombre "PrecipOctGro.txt", su lectura para cargar un *data frame* se hace con:

```
pp <- read.table("PrecipOctGro.txt")
head(pp)  # para verificar los primeros 6 renglones

##      Precip
## 1970   81.2
## 1971  104.0
## 1972   63.5
## 1973  188.9
## 1974   48.1
## 1975   95.1
```

Ahora, para la producción del histogama no se requiere del año en el que se tuvo tal o cuál precipitación; esto es, sólo interesan los valores de las precipitaciones.

```
hh <- hist(pp$Precip, breaks = 15)
```

El resultado de la aplicación de la función `hist()`, se puede ver en la Fig. 5.5. El argumento `breaks=15`, le indica a la función *más o menos* cuántas barras producir, o sea, en cuántos intervalos se dividirán los datos. Si este argumento no se proporciona, el lenguaje seleccionará un valor *adecuado*. Como se puede ver en la figura, el número de intervalos resultantes para este caso fue 13, un número más o menos cercano al sugerido. No nos detendremos aquí a revisar la especificación de los títulos principal y de los ejes de la figura, pues esto se verá en un capítulo posterior.

En el código para producir el histograma, se ha asignado el *resultado* de la función `hist()`, a la variable `hh`. De no haberlo hecho, la gráfica correspondiente, mostrada en la Fig. 5.5, se hubiera producido de cualquier manera. La ventaja de guardar el resultado, es que se puede contar con la información empleada para construir el histograma. En seguida se muestra el contenido de tal variable.

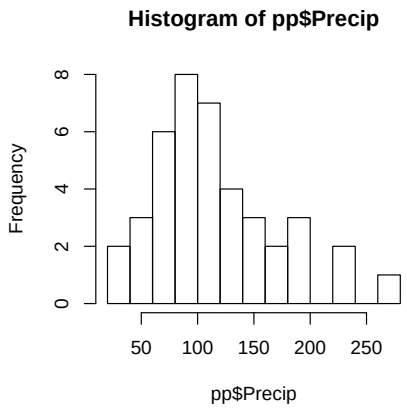

Figura 5.5: Histograma de precipitaciones para el estado de Guerrero en octubre

```
hh

## $breaks
##  [1]  20  40  60  80 100 120 140 160 180 200 220 240 260 280
##
## $counts
##  [1] 2 3 6 8 7 4 3 2 3 0 2 0 1
##
## $density
##  [1] 0.002439 0.003659 0.007317 0.009756 0.008537 0.004878
##  [7] 0.003659 0.002439 0.003659 0.000000 0.002439 0.000000
## [13] 0.001220
##
## $mids
##  [1]  30  50  70  90 110 130 150 170 190 210 230 250 270
##
## $xname
## [1] "pp$Precip"
##
## $equidist
## [1] TRUE
##
## attr(,"class")
## [1] "histogram"
```

De momento interesan los elementos `$breaks` y `$counts` de la variable `hh`. El elemento `$breaks`, indica en qué valores se *rompen* los intervalos de la va-

riable aleatoria. En este caso, por ejemplo, los dos primeros intervalos, serían: $20 \leq precip < 40$ y $40 \leq precip < 60$. El elemento `$counts`, indica la frecuencia de aparición del valor asignado a cada intervalo. Así, por ejemplo, los valores de `$counts` correspondientes a los dos primeros intervalos son 2 y 3, respectivamente, lo que indica que, de los datos originales, hay dos precipitaciones entre 20 y 40 mm, y hay tres entre 40 y 60 mm, y así para todos los otros casos.

Una alternativa en los histogramas, consiste en el uso de la densidad en vez de la frecuencia, definida como, la frecuencia, o número de ocurrencias para un intervalo, dividida entre el número total de ocurrencias y entre el ancho del intervalo, de manera que la suma total de las áreas de las barras del histograma sea la unidad. Por ejemplo, para el primer intervalo, este valor sería: $2/(41 \cdot 20) = 0.002439$. De hecho, en el código anterior, el elemento `$density` de la variable `hh`, muestra cada uno de estos valores. Para producir este diagrama, basta con definir el argumento `freq` en la función `hist()`, como FALSO, como se muestra a continuación.

```
hist(pp$Precip, breaks = 15, freq = FALSE)
```

Ahora, este nuevo resultado de la aplicación de la función `hist()`, se puede ver en la Fig. 5.6. Nótese que este diagrama es igual que el anterior, solamente con un cambio de escala en el eje de las ordenadas. Lo interesante de esta versión es que ahora se puede hablar no en términos absolutos, sino de porcentajes. Por ejemplo, si se toman los intervalos consecutivos del 3 al 5 y se suman las densidades correspondientes y se multiplica ese valor por el ancho del intervalo, lo que corrresponde al área de las tres barras, se puede decir que el 51.22 % de las precipitaciones estuvo entre 60 y 120 mm durante el período de las observaciones.

5.4.2. Densidades y distribuciones de probabilidades

Con referencia al ejemplo anterior, se puede ver que, en general el número de observaciones de las que se puede disponer es limitado: 41, para este caso. Si, hipotéticamente, se pudiera contar con un número infinito de observaciones, el ancho de los intervalos podría reducirse a un valor muy cercano a cero y en vez de la gráfica tipo escalera de la Fig. 5.6, se podría tener una gráfica continua, posiblemente semejante a la que se muestra en la Fig. 5.7, y cuya área total debajo de la curva fuese la unidad. Si, además de eso, se pudiera afirmar que esa curva es representativa no sólo de lo que ha ocurrido históricamente, sino de lo que pudiera ocurrir en el futuro, estaríamos frente al concepto que se conoce como *funciones de densidad de probabilidades*[7] . De igual manera que se hizo en la sección anterior, sólo que hablando en términos de probabilidades, si en la Fig. 5.7, se pudiera evaluar el área bajo la curva, entre a y b y esa fuera

[7]Las funciones de *distribución* de probabilidades son simplemente el acumulado de las funciones de densidad de probabilidades. Si $p(y)$ es la función de densidad de probabilidades de una variable aleatoria y, entonces la función de distribución de probabilidades está dada por $P(y) = \int_{-\infty}^{x=y} p(x)dx$.

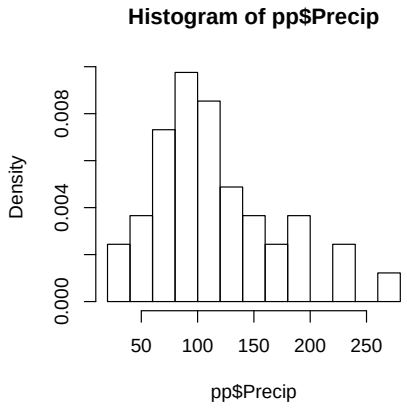

Figura 5.6: Histograma de precipitaciones para el estado de Guerrero en octubre

A, se podría decir que la precipitación tiene una probabilidad A de estar entre los valores a y b. Desde luego que, para evaluar esa área, en este caso sería necesario recurrir a las nociones del cálculo integral.

Desde hace algunos siglos, los matemáticos, como de Moivre, Gauss y Laplace, han desarrollado diversos conceptos al rededor de este tema. De hecho a Gauss se le atribuye la formulación de las funciones de *densidad y distribución normal* de probabilidades. Ambas funciones dependen sólo de dos parámetros estadísticos: la media y la desviación estándar.

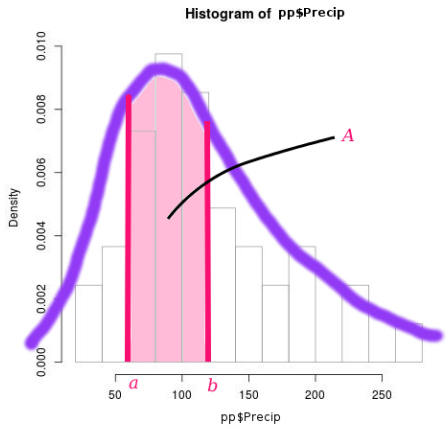

Figura 5.7: Ajuste del histograma a una curva continua