TUTORIAL DE MAPLE

Alfonsa García

Departamento de Matemática Aplicada (EU Informática)

Este tutorial ha sido preparado para la asignatura Codificación de la Información y elaborado, teniendo en cuenta los requisitos de las prácticas de dicha asignatura, a partir de material preparado por **Félix Rincón** y por la propia autora para la antigua asignatura Fundamentos de Criptología.

Presentación

Maple es un **sistema de cálculo científico**, es decir, numérico, simbólico y gráfico. **Algunas de sus capacidades son:**

Operaciones numéricas en aritmética racional exacta o decimal de precisión arbitraria.

Operaciones en aritmética modular

Manipulación algebraica de variables y símbolos.

Operaciones con polinomios, fracciones algebraicas y series.

Cálculo de límites, derivadas e integrales.

Resolución de ecuaciones y sistemas.

Resolución exacta y aproximada de ecuaciones en diferencias y diferenciales.

Operaciones con vectores y matrices.

Cálculo vectorial con funciones de varias variables.

Capacidades gráficas en 2 y 3 dimensiones.

Una amplia biblioteca con más de 3000 funciones matemáticas agrupadas por objetivos.

Lenguaje de programación de alto nivel.

1. Instrucciones básicas y regiones de trabajo

Al poner en marcha el sistema Maple:

Se inicia una sesión de trabajo en un documento análogo al actual, llamado **hoja de trabajo**.

Aparece un **grupo de ejecución**, similar al siguiente, con el **inductor** y el **cursor** parpadeando, que indica que se pueden introducir instrucciones (en formato texto en línea o en formato matemático, para lo que se pueden usar los símbolos de la paleta de expresiones).

1.1 Operaciones básicas

Acción	*Efecto*
Abrir el icono Maple	Iniciar una sesión de trabajo
`;` ó `:`	Terminar instrucción (con o sin mostrar el resultado)
intro	Ejecutar las instrucciones
mayúsculas+**intro**	Saltar de línea en las regiones de entrada
`restart` o icono 	Reiniciar el sistema
nombre `:=` *valor* *nombre* `:=` '*nombre*'	Asignar y desasignar un valor a una variable
`%` `%%` `%%%`	Resultados precedentes
Icono 	Cancelar el cálculo en curso
time()	Medir el tiempo de CPU consumido en la sesión

Manejo de documentos:

<File / New>	Crear una hoja de trabajo nueva, vacía
<File / Open...>	Abrir una hoja de trabajo existente en un fichero
<File / Save>	Salvar la hoja de trabajo actual en un fichero
<File / Print...>	Imprimir la hoja de trabajo actual
<File / Exit>	Salir del sistema

1.2 Regiones de trabajo

Cada hoja de trabajo integra **regiones** de varios tipos:

Regiones de Texto, que se insertan con el icono de la barra de herramientas y contienen anotaciones realizadas por el usuario.

Regiones de Entrada, que se pueden insertar en cualquier momento pulsando el botón de la barra de herramientas. Contienen instrucciones del usuario al sistema. Los comandos se completan con un **terminador**, que puede ser `;` o `:`, y se ejecutan pulsando **intro** con el cursor dentro de la región de entrada.

Regiones de Salida, que contienen resultados generados por Maple.

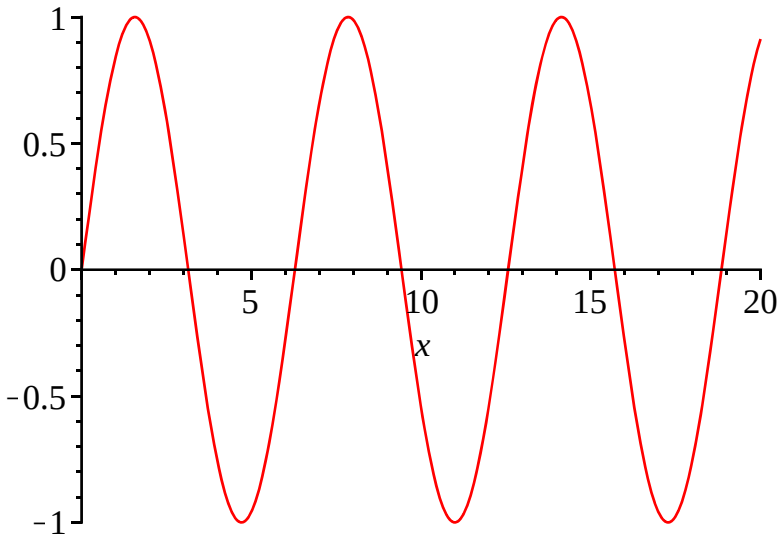
```
> (2*x-x/2)^24;
```

$$\frac{282429536481}{16777216}\, x^{24} \qquad \textbf{(2.2.1)}$$

Gráficos: Contiene el resultado de una instrucción de dibujo, como la siguiente.

Esta instrucción dibuja una función:

```
> plot(sin(x), x = 0..20);
```



1.3 Introducción de comandos

Los **comandos** (**sentencias** y **expresiones**) de las regiones de entrada se construyen con los símbolos matemáticos usuales (números, operadores aritméticos, funciones matemáticas, paréntesis, etc.). Tras escribir la instrucción acabada en punto y coma y pulsar **intro**, Maple devuelve el resultado de cada comando simplificado y evaluado:

```
> (6*x)^3/3^2;
```

$$24\, x^{3} \qquad \textbf{(2.3.1)}$$

Si se escribe la instrucción acabada en dos puntos y se pulsa **intro**, Maple no muestra el resultado pero se guarda en memoria.

```
> 2+1:
```

```
> %^2;
```

$$9 \qquad \textbf{(2.3.2)}$$

Una orden puede ocupar varias líneas. Para saltar de línea, se pulsa la combinación de teclas **mayúsculas**+**intro**:

```
> (3+7)^2
  /
  5^4;
```

$$\frac{4}{25} \qquad \textbf{(2.3.3)}$$

Cada comando puede contener una única expresión o una **secuencia de expresiones** separadas por comas (**,**). En tal caso, Maple devuelve la secuencia de resultados correspondientes a cada una de las intrucciones dadas:

```
> 3!, sin(Pi/3), sqrt(2.0);
```

$$6,\ \frac{1}{2}\sqrt{3},\ 1.414213562 \qquad \textbf{(2.3.4)}$$

1.4 Variables: asignación y evaluación

El **operador de asignación** se usa en la forma *nombre* := *expresión* para asignar el valor *expresión* a la variable *nombre*.

Cuando Maple evalúa una expresión que contiene un nombre, busca si dicho nombre tiene un valor asignado y lo sustituye en la expresión original.

```
> a:=2; 2*a;
```

$$\begin{matrix} 2 \\ 4 \end{matrix} \qquad \textbf{(2.4.1)}$$

Un nombre válido es una cadena de caracteres (letras, números y signos de puntuación) empezando por una letra o un signo de subrayado (_). Los nombres de otro tipo (por ejemplo, con espacios blancos intercalados) deben ponerse entre comillas graves:

```
> `esto es un nombre`:=4;
```

$$4 \qquad \textbf{(2.4.2)}$$

El sistema distingue entre mayúsculas y minúsculas:

```
> B:=5; b; B;
```

$$\begin{matrix} 5 \\ b \\ 5 \end{matrix} \qquad \textbf{(2.4.3)}$$

Si se abre una sesión de trabajo en la que hay variables asignadas, **esta asignación no está operativa si no se ejecuta la instrucción de asignación**. Una vez ejecutada Maple conserva en memoria el valor asignado, aunque se borre la instrucción. Si se quiere "limpiar" la variable es preciso desasignarla.

Para desasignar el valor de una variable se le asigna su propio nombre entrecomillado:

```
> a^2 ; a:= 'a' ; a^2;
```

$$\begin{matrix} 4 \\ a \\ a^2 \end{matrix} \qquad \textbf{(2.4.4)}$$

Recordemos que la instrucción restart reinicia el sistema, desasignando todas las variables:

```
> a:=3; b:=2; c:=4; a^b+c;
```

$$3 \qquad \textbf{(2.4.5)}$$

$$\begin{matrix} 2 \\ 4 \\ 13 \end{matrix}$$

```
> restart; a^b+c;
```

$$a^b + c \qquad \textbf{(2.4.6)}$$

Para **impedir la evaluación** de una expresión, ésta se entrecomilla, en la forma '*expresión*'. Esto impide la evaluación de todas las variables contenidas en *expresión*.

```
> restart;
  a := b;
  b := 3;
```

$$\begin{matrix} b \\ 3 \end{matrix} \qquad \textbf{(2.4.7)}$$

```
> a^2-b/a,'a^2-b/a';
```

$$8, a^2 - \frac{b}{a} \qquad \textbf{(2.4.8)}$$

```
> a^2-'b'/a, 'a^2'-b/a;
```

$$9 - \frac{1}{3}\, b, a^2 - 1 \qquad \textbf{(2.4.9)}$$

2. Errores y sistema de ayuda

Cuando Maple detecta un **error sintáctico**, señala su posición y emite un mensaje explicativo.

```
> 2x+4;
Error, missing operator or `;`
```

El error se corrige editando la región de entrada correspondiente.

```
> 2*x+4;
```

$$2\,x + 4 \qquad \textbf{(3.1)}$$

Si el error es **semántico**, Maple indica su naturaleza.

```
> cos(2, 3);
```

En este caso, la función coseno se invoca con dos argumementos, cuando sólo tiene uno.

```
Error, (in cos) expecting 1 argument, got 2
```

Para entender y corregir los errores es útil invocar el sistema de ayuda de Maple.

Las **páginas de ayuda**, como por ejemplo log , son hojas de trabajo especiales que describen un concepto, incluyendo sintaxis, semántica, ejemplos de uso y enlaces a otras páginas relacionadas. Están estructuradas en forma de árbol, que se puede recorrer mediante los citados enlaces.

El sistema de ayuda se invoca con el menú <Ayuda>, que permite localizar información por contenido, por nombre y por texto:

Para ver los **contenidos** del sistema de ayuda, se selecciona el menú <Ayuda / Ayuda de Maple>.

Para buscar un concepto por su **nombre**, se ilumina el nombre y se selecciona el menú <Ayuda / Ayuda sobre...>, que genera un acceso a la página de ayuda sobre el concepto sombreado.

También se puede pedir ayuda usando ? junto al nombre en una región de trabajo

```
> ?factor;
```

3. Operaciones matemáticas elementales

Maple es capaz de evaluar **expresiones aritméticas** Distingue dos clases de datos numéricos:

Exactos: Se almacenan como símbolos, por lo que carecen de error de redondeo. Incluyen a los números enteros, racionales y algebraicos, a los complejos cuyas partes real e imaginaria son ambas exactas y a las constantes simbólicas como π, *i*, cos(1) o `true`.

Aproximados: Se almacenan como decimales con un número finito de dígitos, por lo que arrastran error de redondeo. Incluyen a los números decimales y a los complejos cuya parte real o imaginaria es decimal.

3.1 Constantes matemáticas numéricas y simbólicas

Constante	*Ejemplos / Nombre Maple*
Enteros	`2`, `-47`
Racionales	`5/2`, `-4/6`, `9/(-6)`, `6/(-2)`
π, *e*, *i*	`Pi`, `exp(1)`, `I`
Decimales finitos	`-4.7`, `.033`
Complejos	`3+4*I`, `1.3-2.*I`
Algebraicos	`3^(1/2)`, `(-2)^(2/3)`, `sqrt(2)`, `RootOf(x^2-2)`
true, false, ∞	`true`, `false`, `infinity`

3.2 Operadores aritméticos

Símbolo	*Operación*	*Ejemplos*
`+`	Suma	`2+3, x+2`
`-`	Opuesto y resta	`-(2+3), 10-3, 10-x`
`*`	Producto	`3*5, 3*y`
`/`	Cociente	`6/4, x/2`
`^`	Potencia	`30^50, x^2`
`!`	Factorial	`5!, n!`
`mod`	Módulo aritmético	`-11 mod 6, 1/5 mod 6`

La precedencia de los operadores es la usual, salvo que se fuerce otra cosa mediante el uso de paréntesis.

3.3 Algunas funciones matemáticas

Funciones que operan con números enteros

Función y argumentos	*Efecto*
`length(x)`	Longitud (nº de dígitos)
`iquo(m, n)`	Cociente entero
`irem(m, n)`	Resto entero
`gcd(x1, x2)`	Máximo común divisor
`lcm(x1, x2, ..., xn)`	Mínimo común múltiplo
`ifactor(r)`	Factorización de enteros
`binomial(m, n)`	Coeficiente binómico
`convert(m, binary)`	Conversión a base 2

Funciones que operan con números reales

Función y argumentos	*Efecto*
`signum(x)`	Signo (1 si x es positivo, -1 si x es negativo)
`abs(x)`	Valor absoluto

`sqrt(x)`	Raíz cuadrada
`exp(x)`	Exponencial
`ln(x), log[b](x)`	Logaritmo (en base *b*)
`sin(x), cos(x), tan(x)`	Trigonométricas
`arc*(x)`	Trigonométricas inversas (`*` = `sin`, `cos`, `tan`)
`floor(x), ceil(x)`	Partes enteras inferior y superior
`round(x), trunc(x)`	Redondeo y truncamiento
`frac(x)`	Parte fraccionaria
`min(x1, x2, ..., xn)`	Mínimo
`max(x1, x2, ..., xn)`	Máximo

Funciones definidas mediante el operador flecha

No es lo mismo introducir la expresión `f:=x^2` que definir la función $f(x)=x^2$. En el primer caso f es sólo el nombre asignado a la expresión x^2. Si se quiere evaluar esta expresión en $x=3$ se puede usar el comando subs.

```
> f:=x^2:
> subs(x=3,f);
```

$$9 \qquad \textbf{(4.3.3.1)}$$

Para definir una función sencilla como $f(x)=x^2$ se puede usar el **operador flecha** mediante la instrucción:

```
> f:=x->x^2;
```

$$x \to x^2 \qquad \textbf{(4.3.3.2)}$$

```
> f(3);
```

$$9 \qquad \textbf{(4.3.3.3)}$$

3.4 Manejo de números decimales

Los números decimales finitos son datos aproximados y se manejan con aritmética aproximada. En concreto, cuando una expresión aritmética contiene algún operando aproximado, Maple **redondea** los demás y opera con **aritmética decimal de precisión arbitraria**. Esto significa que:

Utiliza un número finito de dígitos para el almacenamiento y las operaciones, por lo que se produce **error de redondeo**.

El número de dígitos, que puede ser tan grande como se quiera, está almacenado en la variable Digits, cuyo valor por defecto es 10.

```
> Digits;
```

$$10 \qquad \textbf{(4.4.1)}$$

```
> Digits:=20:
```

Maple evalúa las expresiones formadas por operandos exactos en modo exacto

```
> 2/6 + sqrt(8);
```

$$\frac{1}{3}+2\sqrt{2} \qquad \textbf{(4.4.2)}$$

Si alguno de los operandos es aproximado, las operaciones involucradas se realizan con artimética decimal (de 20 dígitos, en este caso).

```
> 1/3 + 2*sqrt(2.);
```

$$3.1617604580795234309 \qquad \textbf{(4.4.3)}$$

Para forzar la evaluación aproximada de una *expresión*, se aplica la función

`evalf(`*expresión*`,n)`

donde el argumento opcional *n* es el número de dígitos a utilizar, cuyo valor por defecto es el de la variable **Digits**.

```
> evalf(1/3+2*sqrt(2),50);
```

3.1617604580795234309367107817527294904726770840871 **(4.4.4)**

4. Aritmética modular y ecuaciones modulares

Para realizar **arimética módulo un entero**, se utiliza el operador mod, cuya sintaxis es *expresión* **mod** *m* donde:

El operando *expresión* es la expresión a evaluar.

El operando *m* es un entero, mayor que cero, que representa el módulo.

Para **resolver ecuaciones modulares**, se usa la instrucción **msolve**(*ecuación*, *variable*, *m*), donde:

El primer argumento es una ecuación o un conjunto de ecuaciones.

El segundo argumento, opcional, es una variable o un conjunto de variables.

El argumento *m* es el módulo, un entero estrictamente positivo.

4.1 Ejemplo (Operaciones modulares)

(a) Representantes canónicos módulo 6.

```
> [17,-9] mod 6;
```

[5, 3] **(5.1.1)**

(b) Operar módulo 7

```
> 18*25 mod 7, 18^25 mod 7;
```

2, 4 **(5.1.2)**

(c) Suma de los vectores [1, 16, 23, 12] y [17, 0, 16, 18] módulo 25.

```
> [1, 16, 23, 12] +[17, 0, 16, 18] mod 25;
```

[18, 16, 14, 5] **(5.1.3)**

(d) Inversos módulo 10 de 3 y de 4.

```
> 1/3 mod 10;
```

7 **(5.1.4)**

```
> 1/4 mod 10;
Error, the modular inverse does not exist
```

No existe inverso porque mcd(10,4)=2

(e) Raíces cuadradas de la unidad módulo 8.

Son los elementos de Z_8 cuyo cuadrado es 1.

```
> seq([x, x^2 mod 8], x=1..7);
```

[1, 1], [2, 4], [3, 1], [4, 0], [5, 1], [6, 4], [7, 1] **(5.1.5)**

Las raíces cuadradas de la unidad módulo 8 son 1,3,5 y 7.

(f) Raíces cuadradas de la unidad módulo 7.

```
> seq([x, x^2 mod 7], x=1..6);
```

[1, 1], [2, 4], [3, 2], [4, 2], [5, 4], [6, 1] **(5.1.6)**

En este caso las únicas raíces de la unidad son 1 y 6=-1 mod 7. Esto se debe a que 7 es primo.

(e) Resolver la ecuación modular $4x = 2$ módulo 6

```
> msolve(4*x=2,6);
```

$$\{x = 2\}, \{x = 5\} \quad \textbf{(5.1.7)}$$

(f) Resolver el sistema de ecuaciones $\{2x - 3y = 0, 2x + 3y = 6\}$ módulo 7

```
> msolve({2*x-3*y = 0, 2*x+3*y = 6},7);
```

$$\{x = 5, y = 1\} \quad \textbf{(5.1.8)}$$

5. Tipos de datos y operandos de una expresión

En Maple los **datos** son las **expresiones**, cada una de las cuales tiene asociado uno o más **tipos y p**uede constar de otras expresiones más simples, que son sus **operandos**.

Por ejemplo, la expresión $-2xy^2 + \ln(y)$ es de tipo suma y consta de dos operandos: el primero es $-2xy^2$ y el segundo es $\ln(y)$.

Cada expresión se representa mediante un **árbol**, en el cual la **raíz** contiene el tipo y los **subárboles** hijos de la raíz son los operandos.

El tipo de una expresión puede representar:

Una **estructura de datos**, como los tipos `` `+` `` (suma) y `list` (lista).

Una **propiedad matemática**, como los tipos `prime` (primo) o `positive` (positivo).

Maple dispone de más de 80 **tipos predefinidos**. Todos ellos tienen un nombre asignado y la página de ayuda type contiene la lista completa.

Los tipos se clasifican en dos grupos:

Tipos básicos son aquéllos que figuran en la raíz del árbol que representa la expresión, como `integer` (número entero) o `list`.

Tipos estructurados son expresiones que combinan tipos más sencillos. Por ejemplo, `set(integer)` que se refiere a un conjunto de números enteros.

5.1 Ejemplos de algunos tipos básicos frecuentes

La siguiente tabla recoge algunos ejemplos de tipos de datos básicos, indicando el número de operandos de cada uno.

Tipo	nops	Ejemplos
integer	1	1; 5; -2
fraction	2	1/3; 2/(-6)
float	2	4.75; .5
complex	1..2	I; 1/2+I; .5+I
character	1	"0"
string	1	"Una cadena."
`` `+` ``	>= 2	x+1; 1-x*y
`` `*` ``	>=	(1-x)*y; -x

	2	
`^`	2	x^2; 1/x
`=`	2	a = b-3
`<>`	2	a <> b
`<`	2	a < b; a > b
`<=`	2	a <= b; a >= b
`and`	2	a and b
`or`	2	a or b
`not`	1	not a
`..`	2	1..10; a..b
exprseq	>=0	a, b, c
list	>= 0	[a, b, c]
set	>= 0	{a, b, c}

5.2 Acceso al tipo de una expresión

Para obtener el tipo básico de una expresión, se ejecuta **whattype**(*expresión*).

Para saber si una expresión es de un determinado tipo, ya sea básico o estructurado, se usa la función **type**(*expresión*, *tipo*), que devuelve **true** si el argumento *expresión* es del *tipo* especificado por el segundo argumento y **false** en caso contrario.

Por ejemplo, el tipo básico de la expresión $-2\,x\,y^2+\ln(y)$ es suma:

```
> expr := -2*x*y^2+ln(y);
  whattype(expr);
```

$$-2\,x\,y^2+\ln(y)$$
$$\text{`+`} \qquad \textbf{(6.2.1)}$$

La función **whattype** sólo accede al tipo básico, mientras que la función **type** dispone de información sobre todos los tipos.

En particular, la expresión anterior es algebraica, pero no polinómica:

```
> type(expr, algebraic);
  type(expr, polynom);
```

true
false **(6.2.2)**

5.3 Acceso a los operandos de una expresión

Para obtener el número de operandos de una expresión, se usa **nops**(*expresión*)

Por ejemplo, la expresión $expr2 := Z = -2\,x\,y^2 + \ln(y)$ tiene dos operandos, que son los dos miembros de la igualdad, a su vez el segundo operando tiene otros dos, que son los dos sumandos.

```
> expr2 := Z=-2*x*y^2+ln(y);
```

$$expr2 := Z = -2\,x\,y^2 + \ln(y) \qquad \textbf{(6.3.1)}$$

```
> nops(expr2);
```

$$2 \qquad \textbf{(6.3.2)}$$

Para extraer uno o más operandos de una expresión, se usa **op**(*posición*, *expresión*), donde el argumento opcional *posición* puede ser:

Un entero *i*, en cuyo caso se obtiene el operando *i*-ésimo de *expresión*.

Un rango de enteros *i*..*j*, en cuyo caso se obtiene la secuencia de operandos de *expresión*, desde el *i*-ésimo hasta el *j*-ésimo.
Por ejemplo

```
> op(1, expr2),op(2, expr2);
```

$$Z,\ -2\,x\,y^2 + \ln(y) \qquad \textbf{(6.3.3)}$$

Si se invoca, dando como único argumento una expresión *e*, la función **op** obtiene la secuencia de todos sus operandos. Es decir, **op(*e*) = op(1..nops(*e*), *e*)**.

```
> op(expr2);
```

$$Z,\ -2\,x\,y^2 + \ln(y) \qquad \textbf{(6.3.4)}$$

```
> op(op(2,expr2));
```

$$-2\,x\,y^2,\ \ln(y) \qquad \textbf{(6.3.5)}$$

6. Secuencias de expresiones

Una **secuencia** de expresiones (tipo **exprseq**) es similar a una lista, pero no se encierra entre corchetes, simplemente se construye conectando expresiones con el operador binario coma (**,**).

6.1 Operaciones básicas con secuencias

Función y argumentos	*Efecto*
NULL	Secuencia vacía
s1, s2, ..., sn	Formación y concatenación de secuencias
nops([*s*])	Número de elementos de una secuencia
s[*p*]	Extracción del elemento de *s* que ocupa la posición *p* (si *p* es negativo se empieza a contar por el final)
seq(*e*, *i* = *a*..*b*,*p*)	Secuencia generada recorriendo un rango. El tercer arugmento es la longitud de paso
seq(*e*, *i* = *e2*)	Secuencia generada recorriendo una expresión. Equivale a **seq(*e*, *i* in *e2*)**

6.2 Ejemplo (Manipulación de secuencias)

Para generar una secuencia basta escribir sus elementos separados por comas, o bien se puede usar la instrucción **seq**(*expresión*, *i* = *inicio*..*fin*) , que devuelve la secuencia resultante de sustituir en *expresión* la variable *i*, que es local, por los valores en el rango desde *inicio* hasta *fin*, con incrementos de 1.

```
> restart;
```

```
s := 1, 2^2, sin(3);
t := seq(j^2, j=1..10);
```

$$s := 1, 4, \sin(3)$$

$$t := 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 \quad \textbf{(7.2.1)}$$

La función **whattype** es aplicable a cualquier expresión, incluídas las secuencias:

```
> whattype(s);
```

$$exprseq \quad \textbf{(7.2.2)}$$

La función **type** no es aplicable a las secuencias:

```
> type(s, exprseq);
Error, invalid input: type expects 2 arguments, but
received 4
```

La función **nops** no es aplicable a secuencias. Para obtener el número de operandos de una secuencia, se la pone entre corchetes, para transformarla en lista.

```
> nops(s);
  nops([s]), nops([t]);
Error, invalid input: nops expects 1 argument, but
received 3
```

$$3, 10 \quad \textbf{(7.2.3)}$$

Para obtener el segundo operando de la secuencia *s* basta pedir:

```
> s[2];
```

$$4 \quad \textbf{(7.2.4)}$$

Para añadir un elemento a una secuencia basta ponerlo a continuación:

```
> t,s[3];
```

$$1, 4, 9, 16, 25, 36, 49, 64, 81, 100, \sin(3) \quad \textbf{(7.2.5)}$$

Para concatenar dos secuencias basta escribir ambas separadas por una coma

```
> s,t;
```

$$1, 4, \sin(3), 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 \quad \textbf{(7.2.6)}$$

Para construir la secuencia de todas las letras del alfabeto castellano (incluida la ñ) se puede hacer:

```
> seq(j, j="a".."n"),"ñ", seq(j, j="o".."z");
```

"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "ñ", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z" **(7.2.7)**

Nota: La ejecución de la función **seq** no modifica el valor de su variable de control.

7. Listas y conjuntos

Las **listas** (tipo **list**) y los **conjuntos** (tipo **set**) se representan externamente como secuencias encerradas entre corchetes y llaves, respectivamente.
Las listas, como las secuencias, conservan tanto el orden como los elementos repetidos. Por el contrario, en los conjuntos los elementos repetidos se eliminan y el orden no es significativo.

7.1 Operaciones básicas con listas y conjuntos

Función y argumentos	*Efecto*
[] {}	Lista y conjunto vacíos.
member(*x*, *E*, 'p')	Test de pertenencia de *x* a la lista o conjunto *E* y posición *p*.
x* in *E	Operador de pertenencia a una lista o conjunto, $x \in E$.
op(*p*, *E*), *E*[*p*]	Extracción del elemento que ocupa la posición *p* de una lista

o conjunto *E*.

`op(i..j, E), E[i..j]`	Obtención de una lista o conjunto con un rango de elementos de *E*.
`sort(l, f)`	Ordenación de la lista *l* usando *f* como criterio.
`C1 union C2`	Unión de conjuntos, $C_1 \cup C_2$.
`C1 intersect C2`	Intersección de conjuntos, $C_1 \cap C_2$.
`C1 minus C2`	Diferencia de conjuntos, $C_1 \setminus C_2$.
`C1 subset C2`	Inclusión de conjuntos, $C_1 \subseteq C_2$.

7.2 Ejemplo (Manipulación de listas y conjuntos)

Para generar una lista basta poner sus elementos entre corchetes. Por ejemplo, vamos a usar la función rand para generar una lista *L* con 20 números enteros pseudoaleatorios comprendidos entre 0 y 50.

```
> restart:
> L:=[seq(rand(0..50)(), j=1..20)];
```

$L := [28, 46, 44, 31, 5, 33, 43, 35, 37, 4, 26, 31, 16, 39, 36, 25, 33, 17, 42, 18]$ **(8.2.1)**

Para añadir elementos a una lista se genera una nueva con los operandos de la antigua y los nuevos. Por ejemplo, vamos a modificar la lista anterior añadiendo a final los cinco primeros cuadrados perfectos

```
> L:=[op(L), seq(j^2, j=1..5)];
```

$L := [28, 46, 44, 31, 5, 33, 43, 35, 37, 4, 26, 31, 16, 39, 36, 25, 33, 17, 42, 18, 1, 4, 9, 16, 25]$ **(8.2.2)**

Se puede ordenar *L* en orden creciente mediante la función sort.

```
> sort(L);
```

$[1, 4, 4, 5, 9, 16, 16, 17, 18, 25, 25, 26, 28, 31, 31, 33, 33, 35, 36, 37, 39, 42, 43, 44, 46]$ **(8.2.3)**

Para averiguar si un elemento está en una lista se puede usar la función member, incluyendo para almacenar la posición del lelemento en la lista

```
> member(28, L,'pos');
```

true **(8.2.4)**

```
> pos;
```

1 **(8.2.5)**

```
> member(20, L,'pos');
```

false **(8.2.6)**

Para ver si una lista tiene elementos repetidos se la puede convertir en conjunto:

```
> C:=convert(L, set);
```

$C := \{1, 4, 5, 9, 16, 17, 18, 25, 26, 28, 31, 33, 35, 36, 37, 39, 42, 43, 44, 46\}$ **(8.2.7)**

```
> nops(C);
```

20 **(8.2.8)**

```
> nops(L);
```

25 **(8.2.9)**

Hay 5 elementos repetidos en L

Se puede extraer elementos de una lista o conjunto del siguiente modo:

Los tres primeros elementos del conjunto C:

```
> C1:=C[1..3];
```

$$C1 := \{1, 4, 5\} \quad \textbf{(8.2.10)}$$

Los cinco últimos elementos de la lista L:

```
> L2:=L[-5..-1];
```

$$L2 := [1, 4, 9, 16, 25] \quad \textbf{(8.2.11)}$$

8. Tiras o cadenas de caracteres

Una **tira o cadena de caracteres** (tipo `string`) se encierra entre comillas dobles (`""`), que la delimitan sin formar parte de ella.

8.1 Operaciones básicas con tiras de caracteres

Función y argumentos	*Efecto*
`"tira"  ""`	Cadena de caracteres y cadena vacía.
`\"`	Carácter `"` dentro de una tira.
`length(t)`	Longitud (número de caracteres).
`t[p]`	Extraer el caracter de posición `p` de la tira `t`.
`cat(t1, ..., tn)`	Concatenar tiras.
`SearchText(t1, t2)`	Buscar la tira `t1` en la tira `t2`.

8.2 Ejemplo (Manejo de cadenas de caracteres)

```
> restart;
```

Definición de una tira:

```
> tira := "esto es una tira";
  whattype(tira);
```

tira := "esto es una tira"

string **(9.2.1)**

El único operando de una cadena de caracteres es ella misma y la longitud se mide con la función `length`.

```
> nops(tira);
  length(tira);
```

1

16 **(9.2.2)**

Un carácter (tipo `character`) es una cadena de caracteres de longitud 1.

```
> type("ñ",character);
```

true **(9.2.3)**

```
> type("ñ",string);
```

true **(9.2.4)**

```
> type("ññ",character);
```

false **(9.2.5)**

Para extraer una subtira, se indexa la cadena de caracteres con un entero o un rango de enteros. (Si el entero es negativo, empieza a contar por el final)

```
> tira[2],tira[3..6], tira[-1];
```

"s", "to e", "a" **(9.2.6)**

La función `cat`, que concatena sus argumentos, permite unir distintas tiras:

```
> cat(tira[1..12], "más larga");
```

"esto es una más larga" **(9.2.7)**

Se puede concatenar tiras con números enteros y con símbolos:

```
> cat("El resultado es ", -2, `.`);
```

"El resultado es -2." **(9.2.8)**

Podemos usar seq para obtener todos los caracteres de la nueva cadena:

```
> seq(j, j in %);
```

"E", "l", " ", "r", "e", "s", "u", "l", "t", "a", "d", "o", " ", "e", "s", " ", "-", "2", "." **(9.2.9)**

La secuencia vacía **NULL** es el elemento neutro para la concatenación:

```
> cat("esto es una cadena",NULL);
```

"esto es una cadena" **(9.2.10)**

Las funciones **SearchText** y **searchtext** localizan una cadena de caracteres dentro de otra.

La función **SearchText** devuelve la posición de su primer argumento dentro del segundo. Si no se encuentra, devuelve 0:

```
> SearchText("u", tira); SearchText("U", tira);
```

9
0 **(9.2.11)**

La función **searchtext** hace lo mismo, pero no distingue mayúsculas de minúsculas:

```
> searchtext("U", tira);
  searchtext("b", tira);
```

9
0 **(9.2.12)**

8.3 Ejemplo (Alfabeto castellano de letras mayúsculas)

Secuencia de letras del alfabeto castellano:

```
> seq(j, j="A.."N"),"Ñ", seq(j, j="O.."Z");
```

"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "Ñ", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z" **(9.3.1)**

Concatenación de la secuencia anterior, para formar una cadena de caracteres:

```
> Castellano:=cat(%);
```

Castellano := "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ" **(9.3.2)**

Secuencia de números con las posiciones que ocupan en el alfabeto *Castellano* los símbolos del mensaje "ESTE ES UN MENSAJE EN CASTELLANO"

```
> seq(SearchText(j, Castellano), j="ESTE ES UN MENSAJE EN
  CASTELLANO");
```

5, 20, 21, 5, 0, 5, 20, 0, 22, 14, 0, 13, 5, 14, 20, 1, 10, 5, 0, 5, 14, 0, 3, 1, 20, 21, 5, 12, 12, 1, 14, 16 **(9.3.3)**

9. Vectores y Matrices

Los **vectores** y **matrices** se pueden introducir usando la paleta correspondiente del menú <View / Palettes> y son elementos de tipo **Vector** y **Matrix** respectivamente.

```
> restart:
> v:=<1 | 2 | 3>;
```

$$v := \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \quad \textbf{(10.1)}$$

```
> whattype(v);
```

$$Vector_{row} \quad \textbf{(10.2)}$$

```
> M:=<<1 | 4 | 0> , <1 | 2 | 2> , <1 | 5 | 3>>;
```

$$M := \begin{bmatrix} 1 & 4 & 0 \\ 1 & 2 & 2 \\ 1 & 5 & 3 \end{bmatrix} \quad \textbf{(10.3)}$$

```
> whattype(M);
```

$$Matrix \quad \textbf{(10.4)}$$

```
> type(M,'Matrix'(square));
```

$$true \quad \textbf{(10.5)}$$

También se puede definir matrices usando la función Matrix .

9.1 Operaciones con matrices

Se puede operar con matrices usando los operadores habituales (+ para la suma, * para el producto por escalares, el punto para multiplicar vectores por matrices o matrices entre sí y ^ para calcular potencias):

```
> M:= Matrix([[1,0,-1],[2,1,2],[1,2,3]]);
```

$$M := \begin{bmatrix} 1 & 0 & -1 \\ 2 & 1 & 2 \\ 1 & 2 & 3 \end{bmatrix} \quad \textbf{(10.1.1)}$$

```
> v:=Vector[row]([1,0,-1]);
```

$$v := \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \quad \textbf{(10.1.2)}$$

```
> 2*M;
```

$$\begin{bmatrix} 2 & 0 & -2 \\ 4 & 2 & 4 \\ 2 & 4 & 6 \end{bmatrix} \quad \textbf{(10.1.3)}$$

```
> v.M;
```

$$\begin{bmatrix} 0 & -2 & -4 \end{bmatrix} \quad \textbf{(10.1.4)}$$

```
> M^(-1);
```

$$\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & -\frac{1}{4} \\ 1 & -1 & 1 \\ -\frac{3}{4} & \frac{1}{2} & -\frac{1}{4} \end{bmatrix} \quad \textbf{(10.1.5)}$$

```
> M.M^(-1);
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \textbf{(10.1.6)}$$

Maple dispone de aritmética matricial modular y permite hacer automáticamente la inversa modular de una matriz.

```
> v. M mod 2;
```

$$\begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \quad \textbf{(10.1.7)}$$

```
> M^(-1) mod 27;
```

$$\begin{bmatrix} 7 & 14 & 20 \\ 1 & 26 & 1 \\ 6 & 14 & 20 \end{bmatrix} \quad \textbf{(10.1.8)}$$

9.2 Librerías de álgebra lineal

Maple dispone de varias librerías, que contienen funciones útiles de cálculo matricial

La librería LinearAlgebra contiene 118 funciones propias del cálculo matricial.

```
> with(LinearAlgebra);
```

[*&x, Add, Adjoint, BackwardSubstitute, BandMatrix, Basis, BezoutMatrix, BidiagonalForm, BilinearForm, CARE, CharacteristicMatrix, CharacteristicPolynomial, Column, ColumnDimension, ColumnOperation, ColumnSpace, CompanionMatrix, ConditionNumber, ConstantMatrix, ConstantVector, Copy, CreatePermutation, CrossProduct, DARE, DeleteColumn, DeleteRow, Determinant, Diagonal, DiagonalMatrix, Dimension, Dimensions, DotProduct, EigenConditionNumbers, Eigenvalues, Eigenvectors, Equal, ForwardSubstitute, FrobeniusForm, GaussianElimination, GenerateEquations, GenerateMatrix, Generic, GetResultDataType, GetResultShape, GivensRotationMatrix, GramSchmidt, HankelMatrix, HermiteForm, HermitianTranspose, HessenbergForm, HilbertMatrix, HouseholderMatrix, IdentityMatrix, IntersectionBasis, IsDefinite, IsOrthogonal, IsSimilar, IsUnitary, JordanBlockMatrix, JordanForm, KroneckerProduct, LA_Main, LUDecomposition, LeastSquares, LinearSolve, LyapunovSolve, Map, Map2, MatrixAdd, MatrixExponential, MatrixFunction, MatrixInverse, MatrixMatrixMultiply, MatrixNorm, MatrixPower, MatrixScalarMultiply, MatrixVectorMultiply, MinimalPolynomial, Minor, Modular, Multiply, NoUserValue, Norm, Normalize, NullSpace, OuterProductMatrix, Permanent, Pivot, PopovForm, QRDecomposition, RandomMatrix, RandomVector, Rank, RationalCanonicalForm, ReducedRowEchelonForm, Row, RowDimension, RowOperation, RowSpace, ScalarMatrix, ScalarMultiply, ScalarVector, SchurForm, SingularValues, SmithForm, StronglyConnectedBlocks, SubMatrix, SubVector, SumBasis, SylvesterMatrix, SylvesterSolve, ToeplitzMatrix, Trace, Transpose, TridiagonalForm, UnitVector, VandermondeMatrix, VectorAdd, VectorAngle, VectorMatrixMultiply, VectorNorm, VectorScalarMultiply, ZeroMatrix, ZeroVector, Zip*] **(10.2.1)**

```
> Determinant(<<1 | 4 | 0> , <1 | 2 | 2> , <1 | 5 | 3>>);
```

$$-8 \quad \textbf{(10.2.2)}$$

```
> Rank(<<1 | 4 | 0> , <1 | 2 | 2> , <2 | 6 | 2>>);
```

$$2 \quad \textbf{(10.2.3)}$$

Para calcular calcular rangos o hacer la forma escalonada reducida en Z_2, se pueden usar funciones del paquete de Álgebra Lineal Modular.

```
> with(LinearAlgebra:-Modular):
```

Ahora, para determinar el rango en Z_2 de la matriz **A** se puede usar la instrucción **Rank(2, A)**.

```
> A;
```

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \quad \textbf{(10.2.4)}$$

```
> Rank(2,A);
```

$$2 \quad \textbf{(10.2.5)}$$

(En el cuerpo de los números reales esta matriz tendría rango 3).
Para obtener la forma escalonada reducida de una matriz en un cuerpo finito (en nuestro caso Z_2) se puede usar la función RowReduce.

```
> RowReduce(2,A,3,4,3,'det',0,'rango',0,0,true);
> A;
```

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \textbf{(10.2.6)}$$

```
> rango;
```

$$2 \quad \textbf{(10.2.7)}$$

```
> det;
```

$$0 \quad \textbf{(10.2.8)}$$

Notas:
1. La aplicación a la matriz A de la función anterior cambia la matriz asignada a A y la sustituye por su forma escalonada reducida.
2. OJO: No es lo mismo obtener la escalonada reducida de una matriz trabajando en los números reales y después hacer módulo 2, que aplicar el algoritmo de G-J directamente en Z_2.
3. Al copiar una matriz grande, Maple muestra sólo la referencia. Para que se vean las matrices más grandes se puede modificar la variable **rtablesize** del interface.

Construcción de una matriz grande:

```
> m:= (i,j)->  if i<>j  then 1 else 0 fi:
> M:=Matrix([seq([seq(m(i,j), i=1..20)], j=1..10)]);
```

$$M := \begin{bmatrix} \textit{10 x 20 Matrix} \\ \textit{Data Type: anything} \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran\_order} \end{bmatrix} \quad \textbf{(10.2.9)}$$

```
> interface(rtablesize=20);
```

$$10 \quad \textbf{(10.2.10)}$$

```
> M;
```

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \quad \textbf{(10.2.11)}$$

10. Elementos de Programación

10.1 Bucles

Un **bucle** es una secuencia de instrucciones que se repite cierto número de veces.
Maple dispone de bucles **`from`** / **`while`**, cuya sintaxis general es

```
for variable from inicio by paso to fin
while condición do
  cuerpo
od
```

y de bucles **`in`** / **`while`**, cuya sintaxis es

```
for variable in expresión
while condición do
  cuerpo
od
```

En ambos casos:

Las expresiones *`variable`*, *`inicio`*, *`fin`*, *`paso`*, *`condición`* y *`expresión`* son, respectivamente, un nombre, dos números o dos caracteres, un número, una condición booleana y una expresión.

La cláusula *`cuerpo`* es la secuencia de instrucciones a repetir, que se ejecutan una a una, ordenadamente.

Con la sintaxis **`from`** / **`while`**, *`cuerpo`* se repite con *`variable`* tomando valores desde *`inicio`* hasta *`fin`*, con incrementos de *`paso`*, mientras *`condición`* sea cierta.

Con la sintaxis **`in`** / **`while`**, *`cuerpo`* se repite con *`variable`* recorriendo los operandos de *`expresión`*, mientras *`condición`* sea cierta.

El test de terminación para las cláusulas **`to`** y **`while`** se realiza al principio de cada iteración.

Salvo los delimitadores **`do`** y **`od`**, todas las cláusulas son opcionales. Los valores por defecto de *`variable`*, *`inicio`*, *`paso`*, *`fin`* y *`condición`* son, respectivamente, una variable muda, 1, 1, ∞ y **`true`**.

Las sentencias

next y **break**, que sólo pueden aparecer en el interior de un bucle, producen el salto a la siguiente iteración y la salida inmediata del bucle, respectivamente.

El cierre **od** se puede sustituir por **end do**.

Ejemplo (Bucles **to** e **in**)

```
> restart:
```

El siguiente bucle calcula los cuadrados de los números naturales pares menores de 15.

```
> for j from 2 to 15 by 2 do j^2 od;
```

4
16
36
64
100
144
196 **(11.1.1.1)**

Cada cuadrado es el resultado de una instrucción diferente y aparece en su propia región de salida. Por ello, si accedemos a la última ejecución se obtiene 196.

```
> %;
```

196 **(11.1.1.2)**

Tras la ejecución del bucle, su variable de control tiene asignado el valor siguiente al final:

```
> j;
```

16 **(11.1.1.3)**

A continuación, se define un bucle para calcular la suma $1^2 + 2^2 + ... + 10^2$. Para ello, se puede usar un bucle **to**, combinado con un acumulador *suma*, que se inicia a 0.

```
> suma := 0:
  for i to 10 do
    suma := suma + i^2
  od;
```

suma := 1
suma := 5
suma := 14
suma := 30
suma := 55
suma := 91
suma := 140
suma := 204
suma := 285
suma := 385 **(11.1.1.4)**

Comprobación:

```
> add(i^2,i=1..10);
```

385 **(11.1.1.5)**

Ejemplo (Algoritmo de Euclides mediante un bucle **while**)

Se trata de calcular el máximo común divisor de los enteros 26 y 16, aplicando el algoritmo de Euclides de divisiones sucesivas, que consiste en lo siguiente:

Se divide 26 entre 16, generando resto 10.

Se divide 16 entre 10, generando resto 6.

Se divide 10 entre 6, generando resto 4.

Se repiten estas divisiones hasta obtener resto 0. El divisor correspondiente es el máximo común divisor.

En la siguiente instrucción se define el bucle **while**, que omite las demás cláusulas. Se introducen las variables *a*,*b* y *r*, para ir almacenado los sucesivos dividendo, divisor y resto.

```
> restart;
  a:=26;
  b:=16;
  r:=irem(a,b);
  while r <> 0 do
   a:=b;
  b:=r;
  r:=irem(a,b);
  od;
```

$a := 26$
$b := 16$
$r := 10$
$a := 16$
$b := 10$
$r := 6$
$a := 10$
$b := 6$
$r := 4$
$a := 6$
$b := 4$
$r := 2$
$a := 4$
$b := 2$
$r := 0$ **(11.1.1.1.1)**

El resultado se almacena en *b*:

10.2 Selección

La **sentencia de selección** ejecuta acciones dependiendo de que se verifiquen o no ciertas condiciones. Su sintaxis es

```
if condición_1 then acción_1
elif condición_2 then acción_2
...
elif condición_n then acción_n
else alternativa
fi
```

donde:

Si la condición booleana *condición_1* es cierta, se ejecuta *acción_1*.

En caso contrario, si la condición booleana *condición_2* es cierta, se ejecuta *acción_2*.

En caso contrario, se procede sucesivamente con las siguientes parejas de condición booleana *condición_i* y secuencia de instrucciones *acción_i*.

Si todas las condiciones *condición_i*, con $i = 1..n$, son falsas, se ejecuta *alternativa*.

Tanto *acción_i*, con $i = 1..n$, como *alternativa* son una secuencia de instrucciones, que se ejecutan una a una, ordenadamente.

Las cláusulas **elif** son opcionales y su número es arbitrario.

La cláusula **else** es opcional.

El cierre **fi** se puede sustituir por **end if** .

Ejemplo (Primalidad de $2^n - 1$, con n natural)

Se trata de analizar la primalidad de los números de la sucesión $2^n - 1$, con n natural:

0, 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, ...

Los dos primeros no son primos y los dos siguientes sí lo son.

Para observar otros casos, se genera algunos términos con un bucle **from**.

Si el número es menor que dos, entonces no es primo.

Si es primo, se hace constar.

En otro caso, se factoriza con la función **ifactor**.

Se usa **print** para mostrar los resultados intermedios durante la ejecución del bucle:

```
> restart;
  for n from 0 to 10 do
    2^n-1;
    if n < 2 then print(%, ` no es primo.`)
    elif isprime(%) then print(%, ` es primo.`)
    else print(% = ifactor(%))
    fi
  od:
```

0, *no es primo.*
1, *no es primo.*
3, *es primo.*
7, *es primo.*
$15 = (3)\ (5)$
31, *es primo.*
$63 = (3)^2\ (7)$
127, *es primo.*
$255 = (3)\ (5)\ (17)$
$511 = (7)\ (73)$
$1023 = (3)\ (11)\ (31)$ **(11.2.1.1)**

Una curiosidad:

Parece que, cuando n es par, $2^n - 1$ es múltiplo de 3. Esto es cierto, porque

$$2^{2k} - 1 = \left(2^k - 1\right)\ \left(2^k + 1\right),$$

y de entre los tres enteros consecutivos $2^k - 1$, 2^k y $2^k + 1$ uno de ellos ha de ser múltiplo de 3 y no puede ser 2^k.

Cuando n es impar, hay términos de la sucesión que son primos y otros que no.

10.3 Procedimientos

Una **función o procedimiento** es un programa, escrito en el lenguaje de programación de Maple, que automatiza cierta tarea.

La sintaxis de una **definición de un procedimiento** es

```
proc(x_1::t_1, x_2::t_2, ..., x_n::t_n)
local l_1, l_2, ..., l_p;
```

```
        global g_1, g_2, ..., g_q;
        option o_1, o_2, ..., o_r;
          cuerpo   #comentarios

      end
```

donde:

La cláusula *x_1*::*t_1, ..., x_n*::*t_n* es la secuencia de **parámetros formales** de la función. Cada parámetro es un nombre (*x_i*), seguido opcionalmente por dos pares de puntos (::) y un tipo de dato (*t_i*). Si se especifica el tipo, Maple genera un error cuando los argumentos pasados a la función no son del tipo apropiado.

La cláusula **local** *l_1, ..., l_p;* especifica la secuencia de **variables locales** de la función.

La cláusula **global** *g_1, ..., g_q;* especifica la secuencia de **variables globales** de la función.

La cláusula **option** *o_1, ..., o_r;* especifica la secuencia de **opciones** (trace, remember, builtin , operator arrow)

La cláusula *cuerpo* es una secuencia de instrucciones, que se ejecutan una a una, ordenadamente.

La semántica es la siguiente:

Cuando **se define** una función, Maple analiza y simplifica lo que encierran los delimitadores **proc() end**, pero no lo ejecuta.

Una vez definida una función o procedimiento *f*, se invoca en la forma *f*(*v_1*, *v_2*, ..., *v_m*) donde *v_1, v_2, ..., v_m* es la secuencia de **parámetros actuales** y especifica los valores concretos en los que se quiere evaluar la función. Cuando **se invoca** una función, Maple ejecuta el *cuerpo* y devuelve **el resultado de la última instrucción ejecutada**. Esto es la **salida** del procedimiento. Como opción por defecto, el sistema no muestra los resultados intermedios. Es posible observarlos, asignando a la variable **printlevel** un valor entero alto, por ejemplo 100, o bien usando la opción trace.

Se pueden conseguir otros efectos, como sacar por pantalla mensajes, gráficas o avisos con las funciones: print, lprint, WARNING, o bien forzar la salida con las funciones return o error.

Notas:

Todas las cláusulas son opcionales, salvo los delimitadores **proc()** y **end**.

La expresión resultante de una definición de función es de tipo **procedure**.

La porción de código entre un carácter sostenido (#) y un retorno de carro es un comentario.

La cláusula **end** se puede sustituir por **end proc.**

10.3.1 Ejemplo (Contar primos)

Se trata de definir la función **CuentaPrimos** que toma como argumento una lista L de números enteros y calcula cuantos de estos números son primos:

```
> restart;
  CuentaPrimos := proc(L::list(integer))     # Parámetro
  de tipo lista de enteros.
```

```
  local x, contador;                          # Variables
  locales.
   contador:=0;                               #Se inicia le
  contador a 0.
    for x in L do                             # Se recorre
  la lista.
      if isprime(x) then
        contador:=contador+1                  #Si el número
  es primo se incrementa el contador.
       fi                                     #Cierre del
  condicional.
     od;                                      #Cierre del
  bucle.
   contador                                   #Valor
  retornado.
  end;
```

CuentaPrimos := **proc**(*L*::(*list*(*integer*))) **(11.3.1.1)**

local *x*, *contador*;
contador := 0;
for *x* **in** *L* **do**
if *isprime*(*x*) **then** *contador* := *contador* + 1 **end if**
end do;
contador
end proc

La función tiene un parámetro, de tipo lista de enteros, y carece de variables globales y de opciones.

Hay dos variable locales, una que sirve para controlar un bucle y otra para contar

El cuerpo es un bucle con un condicional

Cuando se define un procedimiento, Maple lo almacena en memoria, eliminando los comentarios, y además lo reescribe en pantalla. En general, conviene evitar dicha reescritura, usando el terminador **:** en la definición.

Una vez definida la función, se puede invocar con cualquier argumento del tipo apropiado:

```
> CuentaPrimos([2,-3,5,6,7,24]);
```

3 **(11.3.1.2)**

```
> CuentaPrimos([2,-3]);
```

1 **(11.3.1.3)**

Si el argumento falta o no es del tipo adecuado, se produce un error:

```
> CuentaPrimos();
  CuentaPrimos(4,5,6);
Error, invalid input: CuentaPrimos uses a 1st argument,
L (of type list(integer)), which is missing
Error, invalid input: CuentaPrimos expects its 1st
argument, L, to be of type list(integer), but received
4
```

10.3.2 Ejemplo (Diferencia entre salida y efectos laterales)

Vamos a modificar el procedimiento anterior para la salida siga siendo el número de primos pero almacene en una lista los primos que vaya encontrando y la muestre por

pantalla

```
> CuentaPrimos2 := proc(L::list(integer))
  local x, contador, listaprimos;
  # se añade una variable local para la lista de primos.
   contador:=0;
    listaprimos:=[];                              #Se inicia
  listaprimos como lista vacía.
    for x in L do                               # Se recorre
  la lista.
      if isprime(x) then
        contador:=contador+1;                    #Si el
  número es primo se incrementa el contador
         listaprimos:=[op(listaprimos), x]      #y se añade
  x a la lista de primos
       fi                                        #Cierre del
  condicional.
     od;                                         #Cierre del
  bucle.
   print("Los primos encontrados son:");         #efectos
  laterales
  print(listaprimos);
  print("El número de primos es:");
   contador                                      #Valor
  retornado.
  end:
> CuentaPrimos2([2,-3,5,6,7,24]);
                    "Los primos encontrados son:"
                              [2, 5, 7]
                    "El número de primos es:"
                                3                          (11.3.2.1)
```

Aunque se muestren por pantalla diversos mensajes, la salida del procedimiento es el resultado de la última ejecución.

```
> %;
                                3                          (11.3.2.2)
```

10.3.3 Salida forzada

Hay dos mecanismos que fuerzan la salida de una función:

Retorno explícito. La sentencia

```
return expr_1, expr_2, ..., expr_n
```

produce la salida inmediata de la función que la contiene, retornando al punto de invocación. En tal caso, el valor de la función es la secuencia de expresiones *expr_1, expr_2, ..., expr_n*.

Retorno con resultado de error. La sentencia

```
error texto, expr_1, expr_2, ..., expr_n
```

produce la salida inmediata de la función que la contiene al nivel de sesión de Maple y genera el mensaje de error

```
Error, (in función) texto, expr_1, expr_2, ..., expr_n
```

donde:

función es el nombre del procedimiento.

texto es una cadena de caracteres que contiene el texto del mensaje de error, combinado con posiciones numeradas en la forma **%i**, con $i = 0..9$.

expr_1, expr_2, ..., expr_n es una secuencia de expresiones que se colocan en las posiciones numeradas de *texto*.

En ambos tipos de salida son útiles las variables:

procname: nombre con el que ha sido invocada la función.

nargs: número de parámetros actuales pasados a la función.

args: secuencia de parámetros actuales pasados a la función.

10.3.4 Ejemplo (Retorno explícito)

A continuación, se define la función **Hay_primo**, que recibe una lista de números enteros y devuelve **true** o **false**, según que contenga o no algún primo.
Se puede hacer de modo que el bucle que recorre la lista se interrumpa en el momento que se encuentre el primer primo y que dicho primo encontrado se almacene en un segundo parámetro de la función, que debe ser de tipo evaln.
En el cuerpo se define un bucle **in**, que va recorriendo la lista. Cuando se encuentra el primer primo, se sale de la función con la sentencia **return**.

```
> restart;
  Hay_primo := proc(lista::list(integer), primo::evaln)
  local i;
    for i in lista do
      if isprime(i) then
        primo := i;
        return true
      fi
    od;
    false
  end:
```

Pruebas:

```
> Hay_primo([4,6,9,11,13],p);
```

true **(11.3.4.1)**

```
> p;
```

11 **(11.3.4.2)**

La función genera errores cuando el primer argumento no es una lista de enteros o el segundo argumento no se evalúa a un nombre, o no lo hay:

```
> Hay_primo({4, 7});
  Hay_primo([4, Pi]);
  Hay_primo([4, 7], 7);
  Hay_primo([4,7]);

Error, invalid input: Hay primo expects its 1st
```

```
argument, lista, to be of type list(integer), but
received {4, 7}
Error, invalid input: Hay primo expects its 1st
argument, lista, to be of type list(integer), but
received [4, Pi]
Error, illegal use of an object as a name
Error, invalid input: Hay primo uses a 2nd argument,
primo (of type evaln), which is missing
```

10.3.5 Ejemplo (Salida forzada en un procedimiento definido sobre una secuencia)

No se puede definir una función con un parámetro formal de tipo secuencia de expresiones, ya que, al invocarla, el sistema considera cada operando de la secuencia como un parámetro independiente.
La solución pasa por definir la función sin parámetros formales y usar las variables **nargs** y **args**, que sólo tienen valores asignados dentro del cuerpo de una función:
El siguiente procedimiento calcula el máximo de una secuencia de números que se le pase como parámetro de entrada.

```
> restart;
  Máximo := proc()
  local i, M;
    if nargs = 0 then error "no hay argumentos." fi;
    M:=-infinity;
    for i in args do
      if not type(i, numeric) then return 'procname(args)'
      elif i > M then M:=i
      fi
    od;
    M
  end:
```

Pruebas:

Se puede invocar la función con cualquier número de argumentos:

```
> Máximo(-2, 3, 25/7),  Máximo(5);
```

$$\frac{25}{7}, 5 \qquad \textbf{(11.3.5.1)}$$

Si no hay argumentos se genera una salida con resultado de error:

```
> Máximo();
Error, (in Máximo) no hay argumentos.
```

Si alguno de los argumentos no es numérico, la función devuelve una invocación no evaluada de la función **máximo** con los parámetros actuales, generada por la sentencia **return 'procname(args)'**:

```
> Máximo(3, x);
```

$$Máximo(3, x) \qquad \textbf{(11.3.5.2)}$$

```
>  Máximo(3, 5, Pi);
```

$$Máximo(3, 5, \pi) \qquad \textbf{(11.3.5.3)}$$

```
> whattype(Pi);
```

$$symbol \qquad \textbf{(11.3.5.4)}$$

```
> Máximo(3, 5,evalf(Pi));
```

$$5 \qquad \textbf{(11.3.5.5)}$$

```
> Máximo(sqrt(2),sqrt(3));
```

$$Máximo\left(\sqrt{2}, \sqrt{3}\right) \qquad \textbf{(11.3.5.6)}$$

```
> Máximo(sqrt(2.),sqrt(3.));
```

$$1.732050808 \qquad \textbf{(11.3.5.7)}$$

Si el argumento es una lista de enteros la salida también es la invocación sin evaluar:

```
> máximo([1, 2]);
```

$$máximo([1, 2]) \qquad \textbf{(11.3.5.8)}$$

10.3.6 Ejemplo (Salida con resultado de error, indicando el dato erróneo)

Se define un procedimiento que recibe una secuencia de números enteros y devuelve un conjunto de pares con los números de la secuencia que sean cuadrados perfectos módulo 7, junto con el valor de la raíz cuadrada correspondiente (*Se usa la función* numtheory[msqrt] *para determinar la raíz cuadrada modular*). En el bucle, se incluye una salida con mensaje de error cuando algún elemento de la secuencia no sea número entero.

```
> restart:
> Cuadrados_módulo7:= proc()
  local i, Conjunto, r_c;
    Conjunto:={}:
     for i in args do
      if not type(i, integer) then error "El elemento %1
  no es número entero.",i fi;
      r_c:=numtheory[msqrt](i,7);
      if  type(r_c, integer) then Conjunto:=Conjunto union
  {[i,r_c]} fi
    od;
    Conjunto
  end:
> Cuadrados_módulo7(4, 7, 10, 12, 5);
```

$$\{[4, 2], [7, 0]\} \qquad \textbf{(11.3.6.1)}$$

```
> Cuadrados_módulo7(4, 7, -5, 39);
```

$$\{[-5, 3], [4, 2], [7, 0], [39, 2]\} \qquad \textbf{(11.3.6.2)}$$

```
> Cuadrados_módulo7(4, 7, -5, x);
Error, (in Cuadrados_módulo7) El elemento x  no es
número entero.
>
>
>
```