(a)Programar el método anterior solicitando del usuario la función f, los extremos de integración y un error máximo permitido.

(b)Probar el programa anterior para calcular $\int_1^3 e^x \operatorname{sen}(x)\, dx$ con un error menor que 1/1000. Compara con el resultado exacto

CAPÍTULO 3
PROGRAMACIÓN FUNCIONAL

En muchos lenguajes de programación los bucles e iteraciones presentados en el capítulo anterior son posibles. Sin embargo aunque *Mathematica* ofrece este tipo de programación tradicional, es preferible reducir su uso continuado.

La verdadera potencia de *Mathematica* se hace patente cuando hacemos cálculo simbólico. Por esta razón resulta más eficaz diseñar nuestros programas bajo modelos de programación funcional y reglas de transformación. En este capítulo nos ocuparemos de la programación funcional, lo que nos permitirá tratar todo tipo de expresiones, en particular las funciones, como datos.

Veremos como incluso en problemas númericos (Método de Newton...) la programación funcional es una alternativa a bucles e iteraciones.

La programación funcional se basa :

■ Por un lado, en la noción de expresión ("TODO ES UNA EXPRESION"). Esto hace de *Mathematica* un lenguaje muy flexible, donde las funciones pueden operar sobre todo tipo de datos.

■ Por otro, en operadores funcionales, funciones que a su vez actúan sobre otras funciones.

3.1.LISTAS

Las listas constituyen una de los elementos más comunes y también uno de los más potentes en *Mathematica*. Las listas sirven para agrupar elementos de cualquier tipo. Por ejemplo: a={x,2,f[x]}

Es importante manejar con soltura las listas, ya que su estructura es común a la totalidad de las expresiones. A continuación hacemos un breve recorrido por las instrucciones básicas para manejar listas.

■ 3.1.1. Construyendo listas

Dos de las instrucciones más corrientes para construir listas son Table y Array.

■ 3.1.1.1. Table

Table[f, {i, imin, imax, d}] generamos la lista {f[imin],...f[imax]}
Table[f,{i,imax,d}] generamos la lista {f[1],...f[imax]}
Table[f,{i,imax}] generamos la lista {f[1],...f[imax]} , el paso d se supone1
Table[f,{i,imin,imax,di},{j,jmin,jmax,dj}] generamos listas multidimensionales

Table

■ Ejemplo: generamos la lista formada por los diez primeros números primos

```
Table[Prime[i], {i, 10}]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

3.1.1.2. Array

Se denominan arrays a un tipo especial de listas en el que todas las sublistas tienen la misma longitud. Por ejemplo, serán arrays los vectores y las matrices, pero no una lista de la forma {1,2,{3,4}}.

En *Mathematica* disponemos de la instrucción Array para construir arrays genéricos que luego podemos rellenar. La sintaxis es la siguiente

Array[a,{n,m...}] generamos un array de dimensiones n× m×...

Array

```
A = Array[a, {2, 3}]
```

```
{{a[1, 1], a[1, 2], a[1, 3]}, {a[2, 1], a[2, 2], a[2, 3]}}
```

Podemos escribirla en forma de matriz

```
% // MatrixForm
```

$$\begin{pmatrix} a[1, 1] & a[1, 2] & a[1, 3] \\ a[2, 1] & a[2, 2] & a[2, 3] \end{pmatrix}$$

```
a[i_, j_] := i + j;
A // MatrixForm
```

$$\begin{pmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

3.1.2. Longitud y dimensión de una lista.

Length[lista] obtenemos la longitud de lista
Dimensions[lista] obtenemos la dimensión de lista

Por ejemplo:

```
m = {{1, 2}, {3, 4}, {2, 5}};
Length[m]
Dimensions[m]
```

```
3
```

```
{3, 2}
```

Obtenemos que la longitud de m es tres, que es el número de elementos encerrados entre las primeras llaves y la dimensión es {3,2} donde 3 es la longitud del primer nivel (el número de filas) y dos es la longitud de las listas en el segundo nivel (el número de columnas).

Notar que Dimensions sólo se puede aplicar si la longitud de las sublistas es la misma, si lo aplicamos a listas con distinta longitud en los subniveles da el mismo resultado que Length.

3.1.3. Tomando trozos de listas

3.1.3.1

Algunas de las instrucciones más comunes para extraer elementos de listas son:

lista[[i]] extrae el elemento i-ésimo de lista
Part[lista, *i*] extrae la parte *i* de lista
lista[[i,j,...]] extrae el elemento situado en la posición i,j,...
lista[[i]][[j]]... extrae el elemento situado en la posición i,j,...
Part[lista,i,j,...] extrae el elemento situado en la posición i,j,..

Por ejemplo si escribimos:

```
m = {{1, 2, 3}, {{2, {a, 3}}, 7}}
```

```
{{1, 2, 3}, {{2, {a, 3}}, 7}}
```

El elemento a, lo podemos obtener de las siguientes maneras:

```
m[[2, 1, 2, 1]]
m[[2]][[1]][[2]][[1]]
Part[m, 2, 1, 2, 1]
```

```
a
```

```
a
```

```
a
```

• Si en lista[[i]] especificamos un i negativo se entenderá que nos referimos al elemento que ocupa la posición i contada desde el final de lalista.

First[lista] obtenemos el primer elemento de lista
Last[lista] obtenemos el último elemento de lista

3.1.3.2.

Para referirnos a trozos de listas contamos con las siguientes instrucciones:

Take[lista,n] obtenemos los n primeros elementos de lista
Take[lista,-n] obtenemos los n últimos elementos de lista
Take[lista,{n, m}] obtenemos los elementos desde el n hasta el m ambos incluidos

Take

Rest[lista]obtenemos lista con el primer elemento eliminado.
Drop[lista,n] obtenemos lista con los n primeros elementos eliminados.
Drop[lista,-n] obtenemos lista con los n últimos elementos eliminados
Drop[lista,{n,m}] eliminamos los elementos desde n hasta m.

Drop y Rest

3.1.4 Testeando y buscando elementos en la lista

Position[lista, forma]da la posición en que forma ocurre en la lista.

Por ejemplo:

```
Position[{x^3, 5, x^2, {7, x^4}}, x^_]
{{1}, {3}, {4, 2}}
```

Obtenemos que las posiciones de x elevado a "algo" son {{1},{3},{4,2}}

Count[lista,forma] cuenta el número de veces que forma ocurre en la lista

En el ejemplo de antes contamos el número de veces que x^_ aparece en la lista como elemento de la lista.

```
Count[{x^3, 5, x^2, {7, x^4}}, x^_]
2
```

MemberQ[lista, forma] da True si forma es un elemento de lista y False en caso contrario

En el ejemplo anterior 7 no es un elemento de la lista

```
MemberQ[{x^3, 5, x^2, {7, x^4}}, 7]
False
```

FreeQ[lista,forma] testea si forma ocurre en algún lugar en la lista

En el ejemplo anterior 7 aparece en algún lugar de la lista

```
FreeQ[{x^3, 5, x^2, {7, x^4}}, 7]
False
```

En todas estas instrucciones forma puede ser una expresión como 7, que hemos usado en MemberQ y FreeQ, pero también puede ser un modelo (pattern, en inglés), como x^_, que hemos usado en Position y Count. Nos ocuparemos de los modelos en el capítulo 4.

3.1.5. Modificando listas

Prepend[lista,elemento] añade al principio de la lista elemento.
PrependTo[lista,elemento] además actualiza el valor de la lista.
Append[lista,elemento] añade elemento al final de la lista
AppendTo[lista,elemento] además actualiza el valor de la lista

Prepend y Append

Insert[lista,elemento,i] inserta elemento en el lugar i de lista
Delete[lista, elemento,i] elimina el elemento situado en el lugar i de lista

Insert y Delete

Se puede especificar un i negativo y también posiciones en niveles más interiores que el primero con {i,j,....}.
Es conveniente obtener previamente con Position la posición de los elementos a insertar o eliminar.

■ Si en la lista de ejemplos anteriores queremos eliminar el elemento 7 lo hacemos con

```
Delete[{x^3, 5, x^2, {7, x^4}}, {4, 1}]
```

$$\{x^3, 5, x^2, \{x^4\}\}$$

■ 3.1.6. Combinando listas

Join[lista1,....,listan] obtenemos la concatenación de las listas
Union[lista1,...., listan] obtenemos la unión disjunta de las listas,con el resultado puesto en orden standard.

Join y Union

■ Por ejemplo:

```
m = {x, {2, 3}, x^2, 7, 2};
n = {2, 4, 7, hola};
Join[m, n]
Union[m, n]
```

$$\{x, \{2, 3\}, x^2, 7, 2, 2, 4, 7, \text{hola}\}$$

$$\{2, 4, 7, \text{hola}, x, x^2, \{2, 3\}\}$$

Observar que primero se ordenan los números, después las letras, y después las variables por grado.

■ 3.1.7. Reordenando listas

Sort[lista] pone los elementos de la lista en orden standard.
Reverse[lista] invierte el orden de los elementos en la lista.
RotateRight[lista,n] mueve los elementos de la lista n lugares a la derecha.
RotateLeft[lista,n] mueve los elementos de lista n lugares a la izquierda

■ 3.1.8. Manipulando listas anidadas

Ya hemos mencionado antes que el interés de las listas reside en que, aunque esta estructura es particular, todas las expresiones se adaptan a este patrón, y especialmente a las listas anidadas (listas en las que algunos de sus elementos son listas).

Algunas de las instrucciones para manejar listas anidadas son:

Transpose[lista] transpone lista si lista es una matriz

Flatten[lista] elimina todas las sublistas.
FlattenAt[lista,i] elimina todas las sublistas del lugar i.
FlattenAt[lista,{i,j,...}]. elimina las sublistas que aparecen en la posición indicada.

Flatten

■ Por ejemplo:

```
Flatten[{7, {2, {3, 4}}, {2, 5}}]
```

```
{7, 2, 3, 4, 2, 5}
```

```
Position[{7, {2, {3, 4}}, {2, 5}}, {3, 4}]
```

```
{{2, 2}}
```

```
FlattenAt[{7, {2, {3, 4}}, {2, 5}}, {{2, 2}}]
```

```
{7, {2, 3, 4}, {2, 5}}
```

■ 3.1.9. Listas como conjuntos

Union[list1,lis2,lis3,...] da como resultado una lista unión de list1,lis2,...
Intersection[list1,lis2,lis3,...]da como resultado una lista intersección de list1,lis2..
Complement[list1, list]da como resultado la lista complementaria de list1 en list

■ 3.1.10. Otras instrucciones

Inner[f,m1,m2,g] donde f y g son funciones y m1 y m2 son listas.
Agrupa los elementos de m1 y m2 en paralelo, aplica f a cada una de las parejas y después aplica g.

Inner

Si en Inner no se especifica el argumento g por defecto se entiende que Plus.

■ Veamos un ejemplo:

```
Inner[f, {a, b}, {c, d}, g]
```

```
g[f[a, c], f[b, d]]
```

■ Con esta función es como está construido el producto escalar entre vectores y el producto de matrices:

```
lista1 = {a, b, c};
lista2 = {d, e, f};
Inner[Times, lista1, lista2, Plus]
```

```
a d + b e + c f
```

Outer[f,lis1,lis2,...] es un producto exterior generalizado. Hace todas las combinaciones posibles de lis1,lis2,... a nivel 1 y aplica f.

```
Outer
```

■ Para entender mejor como funciona veamos un ejemplo:

```
Outer[f, {a, b}, {1, 2, 3}]
```

```
{{f[a, 1], f[a, 2], f[a, 3]}, {f[b, 1], f[b, 2], f[b, 3]}}
```

```
Outer[D, {a[x, y, z]}, {x, y, z}] // MatrixForm
```

$$\begin{pmatrix} a^{(1,0,0)}[x, y, z], & a^{(0,1,0)}[x, y, z], & a^{(0,0,1)}[x, y, z] \end{pmatrix}$$

■ 3.1.11. EJERCICIOS CON LISTAS

■ 1. Generar con Table una lista de dos niveles el primero de longitud 10 y el segundo de longitud dos de modo que aparezcan emparejados el i-ésimo número impar y el i-ésimo número par.

■ 2. Sea
t={{{2,1},{3,5}},{{3,6},{5,7}},{{a,a},{x,x},{b,c}},{{7,3},{2,0}},{{3,1},{x,{y,x}}}}

• Obtener la longitud y dimensión de t.

• Añadir z a t, al principio de la sublista {b,c} y asignar a t el resultado.

• Eliminar x de la lista obteniendo primero las posiciones que ocupa, y asignar a t el resultado.

• Testar si z ocurre en algún lugar de la lista. Observar la diferencia entre usar el comando FreeQ o MemberQ.

•Usando la definición de t del principio, eliminar y de la lista y asignar a t el resultado. Allanar, usando Flatten, {x,{x}}

■ 3**.** Utilizando Array generar un array bidimensional de dimensiones 4×5, y rellenarlo con el producto de los índices.

■ 4. Utilizando RotateRight, construir una función que codifique palabras rotando las letras de cada palabra n lugares a la derecha. Llama a esta función Codifica.

■ 5. Implementar una función que tenga como argumentos un número x y una lista{a,b,c,...} de como resultado la lista {{x,{a}},{x,{a},{a,b}},{x,{a},{a,b},{a,b,c}},...}

Sugerencia: Utilizando un acumulador y la instrucción Append, implementarlo en un bucle Do o For.

■ 6. Realizar un programa que pidiendo al usuario un número límite l devuelva un array bidimensional l×3, de modo que en la fila i aparezca i, i^2, i^3.
Después realiza un programa que pida al usuario mediante Input un cubo entre 1 y l^3 , que busque el número en la tabla y devuelva su raíz cúbica.

3.2.EXPRESIONES

En *Mathematica* todo es un expresión: funciones, listas, gráficos..... Esta noción es crucial para unificar principios en *Mathematica*.

Todas las expresiones tienen la misma estructura subyacente: cabecera[argumentos].

■ Por ejemplo cuando escribimos x+y esto es interpretado internamente como Plus[x,y], aunque para este operador, como para otros muchos se admite la escritura más familiar: x+y. *Mathematica* se encarga en estos casos de hacer la traducción. A esta forma interna la conoceremos a partir de ahora con el nombre de **forma completa (**FullForm**)**.

■ Las instrucciones para obtener la forma completa y la cabecera de una expresión son:

FullForm[exp] devuelve la forma completa de exp
Head[exp] devuelve la cabecera de exp

■ Por ejemplo la forma completa de 1+x^2+(y+z)^2 será:

```
FullForm[1 + x^2 + (y + z)^2]
```

```
Plus[1, Power[x, 2], Power[Plus[y, z], 2]]
```

■ La cabecera de {a,b,c} es List

```
Head[{a, b, c}]
```

```
List
```

■ Así mismo podemos obtener de que tipo es un número con Head:

```
Head[2.0]
```

```
Real
```

■ Podemos referirnos a partes de expresiones como si lo hicieramos a partes de listas, teniendo en cuenta la forma completa. Veamos los siguientes ejemplos:

▮ En a={x,y,z}, para referirnos a la parte dos lo hacemos con a[[2]]. Si en vez de a={x,y,z} tenemos a=x+y+z y escribimos a[[2]] obtenemos igualmente "y", que es el segundo argumento en la forma completa Plus[x,y,z]. Con a[[0]] obtenemos la cabecera Plus.
Observar que las expresiones se comportan como listas: la cabecera sustituye a las llaves y los argumentos a los elementos de la lista.

▮ Veamos un ejemplo de una expresión con más de un nivel:

```
a = 1 + x^2 ;
FullForm[a]
```

```
Plus[1, Power[x, 2]]
```

```
a[[2]]
```

x^2

```
a[[2, 2]]
```

```
2
```

Con a[[2,2]] obtenemos el exponente 2, ya que la parte 2 de a es Power[x,2], y la parte 2 de esto último es 2.

■ Así, cada expresión no es más que una "lista", y del mismo modo que las listas contienen sublistas, las expresiones contienen subexpresiones.

Todas las instrucciones que usabamos con listas podemos usarlas con expresiones en general. Algunas de ellas admiten variantes. Las veremos en la sección 4.

- **EJERCICIOS**

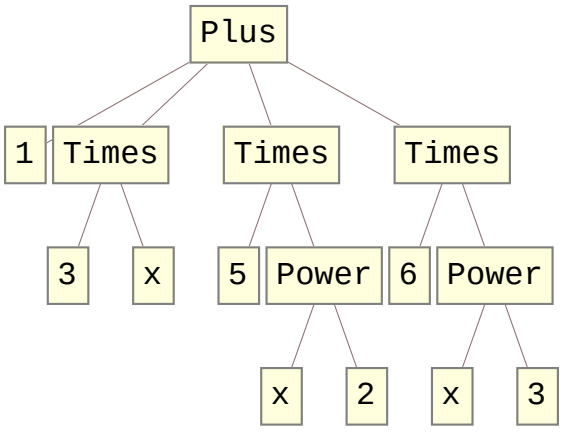
■ 1.- Obtener la forma completa de una regla de transformación {x->a} y de los operadores lógicos ||, &&, ==, !=. Tomando una ecuación polinómica (por ejemplo, x^3+x^2+x+1=0) cualquiera resoverla usando Solve y Roots. Observar como devuelven ambas instrucciones los resultados. Ayudándote de FullForm y de instrucciones para extraer elementos de listas, construye una función que aplicada sobre las soluciones devuelva las soluciones en una lista.

3.3. NIVELES EN EXPRESIONES. EXPRESIONES EN FORMA DE ÁRBOL

Dada una expresión ya hemos visto como actúar sobre ella modificándola y extrayendo partes. Sin embargo cuando las expresiones tienen una cierta uniformidad (por ejemplo son polinomios), es conveniente poderse referir a una colección de partes al mismo tiempo (a los coeficientes, por ejemplo). Surge por esto el concepto de nivel. Para entender mejor lo que es un nivel digamos que cada expresión se puede interpretar en forma de árbol con TreeForm, cada nivel en la expresión se corresponde con un nivel en el árbol, y la cabecera ocupa el nivel cero.

■ Continuando con el ejemplo de los polinomios, escribiendo

```
TreeForm[1 + 3 x + 5 x^2 + 6 x^3]
```



podemos observar que esta expresión tiene tres niveles: en un primer nivel los sumandos, en un segundo nivel los productos coeficiente variable, y en un tercer nivel los exponentes.

Algunas instrucciones para manejar niveles en expresiones son:

Position[expresión, forma, nivel] da la posición en que forma ocurre en el nivel especificado.
Level[expresión, nivel] da la lista de partes de la expresión especificada por nivel.
Depth[expresión] da el número de niveles (la "profundidad") de expresión.

En nivel podemos especificar:

n Los niveles por debajo del n (niveles desde 1 hasta n).
Infinity Todos los niveles.
{n} Sólo el nivel n.
{n,m} Los niveles desde n hasta m, ambos incluidos.

En el ejemplo de antes de los polinomios:

```
p = 1 + 3 x + 5 x^2 + 6 x^3;
Level[p, 1]
Level[p, {2}]
Level[p, {3}]
Depth[p]
```

$\{1, 3\,x, 5\,x^2, 6\,x^3\}$

$\{3, x, 5, x^2, 6, x^3\}$

```
{x, 2, x, 3}
```

```
4
```

Con Level[p,1] obtenemos el nivel 1 en p. Con Level[p,{2}] obtenemos el nivel 2 en p. Con Level[p,{3}] obtenemos el nivel 3 en p . Observamos que el número de niveles en p es 4 .

```
Position[p, x^_]
```

```
{{3, 2}, {4, 2}}
```

Con Position[p,x^_] obtenemos la posición de las subexpresiones de la forma "x elevado a algo" . Observar que no aparece en esta lista la posición de x, esto es porque FullForm de x no es Power[x,_] sino x, es decir Mathematica no hace la sustitución x^1=x.

Ejemplo: Utilizando expresiones referentes a niveles implementar una función que teniendo como argumento un polinomio devuelva sus exponentes.

Primero vamos a probar con un ejemplo

```
P = 1 + 3 x + 7 x^2 + x^3;
```

```
Level[P, {1}]
```

$\{1, 3\,x, 7\,x^2, x^3\}$

```
Level[P, {2}]
```

$\{3, x, 7, x^2, x, 3\}$

```
Level[P, {3}]
```

```
{x, 2}
```

Los exponentes distintos de 0,1 aparecen en general en nivel 3. Si hay un monomio con coeficiente "1" ese exponente se detecta un nivel antes, en nivel 2. Son exponentes los números que aparecen en posición par en nivel 2.

```
P = 1 + 3 x + 7 x^2 + x^3;
Level[P, {2}]
```

```
{3, x, 7, x^2, x, 3}
```

```
P = 1 + x + 7 x^2 + x^3;
Level[P, {1}]
```

```
{1, x, 7 x^2, x^3}
```

```
Level[P, {2}]
```

```
{7, x^2, x, 3}
```

```
p = 1 + 5 x;
Level[p, {1}]
```

```
{1, 5 x}
```

```
Level[p, {2}]
```

```
{5, x}
```

```
p = 5 x;
Level[p, {1}]
```

```
{5, x}
```

```
Level[p, {2}]
```

```
{}
```

```
Depth[p]
```

```
2
```

```
p = 5 + x;
Level[p, {1}]
```

```
{5, x}
```

Entonces los exponentes son los números que aparecen en nivel 3 y los números que aparecen en posición par en nivel 2. Salvo excepcionalmente los exponentes 1 y 0 que se detectan en nivel1. Hay exponente 0 si en nivel 1 hay números en posición impar.

Para detectar si hay exponente 1 o no dependerá de si el coeficiente de x es 1 o no. Si el coeficiente de x es 1, entonces aparecerá x en nivel 1. Si el coeficiente de x no es 1, entonces en nivel 2 aparecerá x en lugar par y si y sólo si.

Ya estamos en condiciones de escribirlo todo en una función

```
Clear[Exponentes]
```

```
Exponentes[P_, var_] :=
    Module[{a, b, c, d, e, f, g, h, j, k},
           a = Level[P, {3}];
           a = Select[a, NumberQ];
           b = Level[P, {2}];
           c = Table[b[[i]], {i, 2, Length[b], 2}];
           d = Select[c, NumberQ];
           e = Join[a, d];
           f = Level[P, {1}];
           If[(f == {}) && (NumberQ[P]), Return[{0}]];
          If[(f == {}), Return[{1}]];
           g = Table[f[[i]], {i, 2, Length[f], 2}];
           h = Select[g, NumberQ];
           If[h ≠ {}, PrependTo[e, h]];
          e = Flatten[e];
          k = Head[P];
          If[k == Plus,
          If[NumberQ[First[f]], PrependTo[e, 0]]];
        If[(k == Plus) || (k == Times),
          If[MemberQ[f, var], PrependTo[e, 1]]];
          If[MemberQ[c, var], PrependTo[e, 1]];
         e = Sort[e]
       ] /; PolynomialQ[P, var]
```

```
P = 1 + x + 7 x^2 + x^3;
Exponentes[P, x]
```

```
{0, 1, 2, 3}
```

```
Exponentes[3 + 5 x^3 + x^2, x]
```

```
{0, 2, 3}
```

```
Exponentes[1 + 5 x, x]
```

```
{0, 1}
```

```
Exponentes[x^5, x]
```

```
{5}
```

```
Exponentes[x, x]
```

```
{1}
```

```
Exponentes[5 x, x]
```

```
{1}
```

```
Exponentes[5 + x, x]
```

```
{0, 1}
```

3.3.1. EJERCICIOS

1.- Sea exp:=f[f[g[a],a],a,h[a],f]. Responde a las siguientes cuestiones:

• ¿cuáles son las posiciones de a desde nivel 2 hasta Infinity?
• Obten los niveles por debajo del 2
• Obten el nivel 2.
• Observa el resultado de Level[exp,-1]
•¿cuántos niveles tiene exp?

2.- Observando el ejemplo anterior de los polinomios, y usando instrucciones referentes a niveles construir una función que aplicada a un polinomio devuelva sus coeficientes.

3.4. MANIPULANDO EXPRESIONES

Todas las instrucciones que usabamos con listas podemos usarlas con expresiones en general. Algunas de ellas admiten variantes relativas a cabeceras y niveles. Estas instrucciones son las siguientes:

En las instrucciones Position, Count, MemberQ, FreeQ podemos opcionalmente especificar el nivel al que queremos que se apliquen.

Join ,Union sólo se pueden usar con expresiones que tengan la misma cabecera.

Cuando usamos Flatten[exp] solo se allanan aquellos subniveles con la misma cabecera que exp. Podemos especificar Flatten[exp,n,h] que allana los subniveles por debajo del n con cabecera h

Select[exp,f] seleciona las partes de expresión para las que f da True.

• Por ejemplo:

```
f[x_] := NumberQ[x];
a = 1 + Log[2] + 7 x^2;
Select[a, f]
```

```
1
```

O bien,

```
Select[a, NumberQ]
```

```
1
```

seleciona aquellas partes de a que son números, pero solo en el primer nivel. Da como resultado 1, ya que Log[2] no es un número, y aunque 7 si lo es, está en el segundo nivel de expresión.

• Con Select[Exp,f,n] seleccionamos los n primeras partes de exp para las que f da True
• Notese que en Select, f es el nombre de una función, es decir no podemos escribir Select[a,NumberQ[x]] sino Select[a,NumberQ]

■ Outer funciona del mismo modo que con listas con la diferencia que todas las expresiones deben tener la misma cabecera. Se sustituye List por la cabecera común de todas las expresiones.

•El resultado de

```
Outer[f, g[a, b], g[1, 2, 3]]
```

```
g[g[f[a, 1], f[a, 2], f[a, 3]], g[f[b, 1], f[b, 2], f[b, 3]]]
```

• Como ejemplo de Outer construimos la matriz jacobiana asociada a una aplicación diferenciable.

```
Jacobianmatrix[f_List, var_List] := Outer[D, f, var];
Jacobianmatrix[{f1[x, y, z], f2[x, y, z], f3[x, y, z]}, {x, y, z}]
```

$$\begin{pmatrix} f1^{(1,0,0)}[x,y,z] & f1^{(0,1,0)}[x,y,z] & f1^{(0,0,1)}[x,y,z] \\ f2^{(1,0,0)}[x,y,z] & f2^{(0,1,0)}[x,y,z] & f2^{(0,0,1)}[x,y,z] \\ f3^{(1,0,0)}[x,y,z] & f3^{(0,1,0)}[x,y,z] & f3^{(0,0,1)}[x,y,z] \end{pmatrix}$$

Si queremos obtenerla en forma de matriz no tenemos más que escribir %//MatrixForm.

```
% // MatrixForm
```

$$\begin{pmatrix} f1^{(1,0,0)}[x,y,z] & f1^{(0,1,0)}[x,y,z] & f1^{(0,0,1)}[x,y,z] \\ f2^{(1,0,0)}[x,y,z] & f2^{(0,1,0)}[x,y,z] & f2^{(0,0,1)}[x,y,z] \\ f3^{(1,0,0)}[x,y,z] & f3^{(0,1,0)}[x,y,z] & f3^{(0,0,1)}[x,y,z] \end{pmatrix}$$

■ 3.4.1 EJERCICIOS

■ 1.- Dado un campo vectorial en R^n obtener su divergencia usando la instrucción Inner.
Nota: Un campo vectorial es un vector de funciones de varias variables con valores reales de la forma (s1,s2,...,sn). Su divergencia se define como la suma $\frac{\partial s1}{\partial x1}+....+\frac{\partial sn}{\partial xn}$.

■ 2.- Utilizando Outer construir una función que calcule el hessiano de f (f es una función real de varias variables). El hessiano de f es la matriz cuadrada que tiene por término ij: $\left(\frac{\partial^2 f}{\partial xi\,\partial xj}\right)$.

■ 3.- Utilizando Outer construir una función que tenga como argumentos una función f de varias variables, la lista de variables de f, var, y que devuelva como resultado la matriz cuadrada cuyo término ij es $\left(\frac{\partial f}{\partial xi}\ \frac{\partial f}{\partial xj}\right)$.

3.5. FUNCIONES PURAS:FUNCIONES COMO ARGUMENTOS.OPERADORES FUNCIONALES.

En *Mathematica* una función es una regla de transformación de expresiones:

```
f[x_]:=cuerpo de la función
```

Para definir una función es necesario asignarle un nombre. Esto es útil si vamos a usar la función muchas veces, ya que nos permite referirnos a ella sólo por el nombre, pero conlleva a la vez, una gran dependencia de ese nombre. Por esta necesidad surge el concepto de función pura. Las funciones puras nos permiten referirnos a funciones sin tener que darlas nombres explícitos. La sintaxis es la siguiente:

Function[variable, cuerpo]
Function[{x1, ...xn}, cuerpo] para una función de varias variables
cuerpo &

En el último caso los argumentos son especificados por:

*n* *n* – ésimo argumento
secuencia de todos los argumentos
*n* secuencia de argumentos empezando por el *n* – ésimo

Es importante en este último caso no olvidar &, ya que si no *Mathematica* no entenderá que se trata de funciones puras.

■ Veamos un ejemplo:
Con #^2& o con Function[x,x^2] representamos en modo puro la función elevar al cuadrado.
Si queremos obtener el cuadrado de un número no tenemos más que aplicar la función de modo habitual:

```
#^2 &[n]
```

n^2

```
Function[x, x^2][n]
```

n^2

■ Se pueden realizar todo tipo de operaciones con funciones puras. Podemos por ejemplo derivarlas :

```
#^2 &'
```

```
2 #1 &
```

La posibilidad de tratar las funciones como expresiones cualesquiera es una consecuencia de la naturaleza simbólica de *Mathematica*. Esto nos permite la utilización de funciones, no sólo sobre datos, sino como argumentos de otras funciones. Esto no es posible en otros lenguajes de programación, donde las funciones sólo pueden actuar sobre datos.

Las funciones que a su vez actúan sobre otras reciben el nombre de operadores funcionales.

A continuación veamos algunos ejemplos :

■ 1.- Definir una función (como función pura) que devuelva True si se aplica sobre expresiones que no tengan subexpresiones y no sean números.
Utilizar esta función como test para extraer de la lista l={x,4,Log[2],{3,2}}los átomos no numericos

```
test[exp_] := (Depth[exp] == 1 && NumberQ[exp] == False)
```

```
test[x]
```

```
True
```

```
test[2]
```

```
False
```

```
test[x^2]
```

```
False
```

```
Select[{x, 4, Log[2], {3, 2}}, test]
```

```
{x}
```

Podemos también escribir el test en forma de función pura

```
Select[{x, 4, Log[2], {3, 2}}, (Depth[#] == 1 && NumberQ[#] == False) &]
```

```
{x}
```

```
Select[{x, 4, Log[2], {3, 2}}, Function[x, Depth[x] == 1 && NumberQ[x] == False]]
```

```
{x}
```

■2.- Construir una función que teniendo como argumento un número x, devuelva la función elevar a ese número.

```
G[x_] := #1^x &
```

La funcion "elevar al cuadrado" es

```
G[2]
```

```
#1^2 &
```

```
G[2][3]
```

```
9
```

■3.-Usando funciones puras construir una función que actuando sobre un valor devuelva la función constantemente ese valor.

```
F[x_?NumberQ] := x &
```

```
F[2]
```

```
2 &
```

```
F[2][3]
```

```
2
```

```
F[2][33]
```

```
2
```

Composition[f,g,...] da la composición de las funciones f, g, ...
InverseFunction[f] da la inversa de la función f
Identity[exp] devuelve la función identidad aplicada a exp

■ Calculamos la función inversa de Sin

```
InverseFunction[Sin]
```

```
ArcSin
```

```
InverseFunction[Sin[#] &]
```

```
ArcSin[#1] &
```

Observar como introducimos las funciones cuando las tratamos como argumentos, es decir, como datos: Nos referimos a ellas por el nombre o bien las escribimos en forma pura. Si introducimos la función seno como Sin[x] no se entiende que nos estamos refiriendo a la función sino al valor particular de Sin en "x".

```
InverseFunction[Sin[x]]
```

```
InverseFunction[Sin[x]]
```

```
Composition[Sin, ArcSin][x]
```

```
x
```

Identity[exp] devuelve la función identidad aplicada a exp.
Through[p[f1,f2][x],q] da p[f1[x],f2[x]] si p es igual a q.
Operate[p,f[x]] da p[f[x]]

■ 3.5.1 EJERCICIOS

■ 1.- Modificar la función NewtonIte a una función auxiliar NewtonAuxIte del capítulo 1 para permitir que la función f de la que vamos a calcular las raíces pueda ser pasada en forma pura. Ello nos permitirá calcular la los valores de f y de su derivada sin necesidad de utilizar reglas de transfomación. Después, para evitar que el usuario tenga que introducir la función f en forma pura, construye otra función NewtonIte que se encargue de hacer la tradución de "la ecuación a la función" y llame a la función NewtonAuxIte.
■ 2.-Construye una función que se llame Hessiano que tenga como argumentos una función f y una lista (lista de variables de f) y que devuelva el hessiano de f.
Pista: Utilizar Outer en la forma Outer[g,var,var].
■ 3.-
a) Utilizando Outer construir una función que tenga como argumentos una función f, una lista de variables var y que devuelva el Array $3\times3\times3$ $\partial_{xi}\ f\ \partial_{xj}\ f\ \partial_{xk}\ f$
b) Utilizando lo anterior construir una función que se llame Operador que tenga como argumentos una función f, una lista de variables var, una matriz cuadrada de la misma longitud que var y que devuelva

$$\sum_{ijk} \partial_{xi}\ f\ \partial_{xj}\ f\ \partial_{xk}\ f\ m_{ij}\ a_k$$

3.6.APLICANDO FUNCIONES A LISTAS. OPERADOR MAP

■ 3.6.1. APLICANDO FUNCIONES A LISTAS

Bastante a menudo necesitamos aplicar una función a una lista de expresiones, de modo que la función f se aplique sobre cada elemento de la lista. Consideremos el ejemplo de elevar al cuadrado los elementos de una lista.

■ Podemos hacerlo usando variables locales e instrucciones de listas con:

```
SquareList[l_List] := Module[{result = {}, i},
 Do[AppendTo[result, l[[i]]^2], {i, Length[l]}]; result]

SquareList[{a, b, c}]
```

$$\{a^2, b^2, c^2\}$$

Esta manera de hacerlo aunque es posible, no resulta la más adecuada. Es similar a la manera tradicional de sumar varios números: contamos con un acumulador donde guardamos los resultados parciales.

■ Como lo que queremos construir es una lista, sería más correcto usar las instrucciones que con tal propósito tiene *Mathematica*: Table y Array.

```
SquareList[l_List] :=
Module[{i}, Table[l[[i]]^2, {i, Length[l]}]]

SquareList[{d, e, f}]
```

$$\{d^2, e^2, f^2\}$$

■ Sin embargo esta operación es tan habitual que la mayoria de las fuciones construidas tienen el atributo Listable, es decir, al aplicarlas sobre una lista se aplican sobre cada elemento de la lista. La función de elevar al cuadrado los elementos de una lista queda entonces bastante simple:

```
SquareList[l_List] := l^2

SquareList[{d, e, f}]
```

$$\{d^2, e^2, f^2\}$$

■ Podemos conseguir que las funciones que construimos sean listables asignándoles el atributo Listable. Esto se hace con :
Attributes[f]={Listable}

■ Sin embargo aunque parece conveniente en un principio que todas las funciones cuenten con este atributo, vamos a ver un ejemplo en que esto constituye todo un problema. El meollo de la cuestión consiste en comprender como se comportan sobre listas, funciones de varios argumentos.

■ Consideramos la función elevar a m , cuya forma interna es: Power[l,m].

• Si el primer argumento de Power es una lista l={a,b,c}y el segundo un número

```
l = {a, b, c};
Power[l, m]
```

$\{a^m, b^m, c^m\}$

• Pero si el segundo argumento de Power es también una lista

```
Power[{a, b, c}, {1, 2, 3}]
```

$\{a, b^2, c^3\}$

los argumentos se toman en paralelo . En estos casos los argumentos deben tener la misma longitud.

■ Tomamos ahora la función derivada: D[f,x]. El segundo argumento puede ser una lista {x,n}, pero en este caso la lista tiene un significado especial: derivamos la función f respecto a x , n veces. Si esta función tuviera el atributo Listable, al aplicarla sobre una lista {f,g}, D[{f,g},{x,n}] obtendríamos {D[f,x],D[g,n]}, ya que los argumentos se toman en paralelo. Este problema lo presentan todas las funciones que permiten como argumentos listas con un significado especial.
Quisieramos que estas funciones se comportasen como listables, pero solo sobre algunos de sus argumentos, sobre los que no son listas especiales.

■ 3.6.2. OPERADOR MAP

■ Para solucionar este tipo de problemas tenemos el comando Map .

Map[*f*, lista] aplica *f a* cada elemento de la lista.

Observar que es un operador funcional, y como tal la función f debe introducirse en modo puro.

• Por ejemplo, si queremos calcular el cuadrado de los elementos de una lista l podemos hacerlo con

```
Map[#^2 &, {a, b, c}]
```

$\{a^2, b^2, c^2\}$

■ Aunque Map en principio se aplica sobre listas podemos aplicarlo sobre expresiones en general. En este caso con

Map[*f*, exp] obtenemos una nueva expresión resultado
de mantener la cabecera *y* aplicar *f a* cada uno de los argumentos de exp.

• Por ejemplo,con

```
Map[#^2 &, a + b + c]
```

$a^2 + b^2 + c^2$

Map[*f*, exp] obtenemos una nueva expresión resultado
de mantener la cabecera *y* aplicar *f a* cada uno de los argumentos de exp.

• Por ejemplo con:

```
Map[(1 + #^2) &, a + b c + c, {2}]
```

$$a + c + \left(1 + b^2\right)\ \left(1 + c^2\right)$$

aplicamos f solo en el segundo nivel de la expresión : bc

MapAll[*f*, expr] aplica *f* en todas las partes de exp

```
MapAll[(1 + #^2) &, a + b c + c]
```

$$1 + \left(3 + a^2 + c^2 + \left(1 + b^2\right)^2\ \left(1 + c^2\right)^2\right)^2$$

■ También es posible aplicar f en partes de expresiones con

MapAt[*f*, exp, {prt1, prt2, ..}]

• Por ejemplo podemos obtener el mismo resultado de antes aplicando a partes en vez de a niveles. b c es la parte {{3}} de expr.

```
Position[a + b * c + c, b * c]
```

```
{{3}}
```

```
MapAt[(1 + #^2) &, a + b * c + c, {{3}}]
```

$$1 + a + c + b^2\ c^2$$

■ Otra sintaxis muy utilizada para Map es:

Map[*f*, exp] es equivalente *a* ***f* /@ exp**
MapAll[*f*, exp] es equivalente *a* ***f* //@ exp**

• Como ejemplo de la utilidad de Map retomamos la construcción de la derivada.
Pretendemos que la función D sea listable sólo sobre el primer argumento.
Para ello usamos Map:

```
dff[f_,arg___]:=D[f,arg];
(*con esto definimos dff como la derivada*)
dff[l_List,arg___]:=Map[D[#,arg]&,l];
(*con esto conseguimos que cuando D se aplique
sobre una lista como primer argumento se aplique
sobre cada elemento de la lista*)
```

De este modo es como está construida **D** internamente. Esta construcci\'on se puede copiar para cualquier función que queramos construir listable sólo en algunos de sus argumentos.

3.6.3. EJERCICIOS

■ 1.- Construir usando Map el operador gradiente en coordenadas cartesianas

■ 2.-Construir usando Map una función que calcule el Hessiano de una función f y que tenga como argumentos una función f y una lista (lista de variables de f) y que devuelva el hesssiano de f

3.7. APLICANDO FUNCIONES CON VARIOS ARGUMENTOS. OPERADOR APPLY

■ El operador **Apply** es una generalización de la nocción usual de aplicar una función a un argumento. Cuando hablamos de aplicar una función a una expresión, entendemos la evaluación de f[exp]. Si queremos aplicar una función a varios argumentos las cosas son más complicadas. Supongamos una lista l={e1,...,en} de argumentos y que queremos calcular f[e1,...,en]. Escribiendo f[l] conseguimos f[{e1,....,e_n], pero lo que queremos es f[e1,...,en]. El operador Apply nace con este propósito. La sintaxis es:

Apply[*f*, lista]

Sin embargo, es posible utilizar Apply con todo tipo de expresiones, no necesariamente listas. La forma de funcionar es la misma:

Apply[f,expr] sustituye la cabecera de exp por f.

■ Por ejemplo: si queremos sumar los elementos de una lista, lo único que tenemos que hacer es sustituir la cabecera List, por Plus

```
Apply[Plus, {a, b, c}]
```

```
a + b + c
```

■ Una sintaxis muy habitual para este operador es:

f **@@** exp

En el ejemplo anterior

```
Plus @@ {a, b, c}
```

```
a + b + c
```

■ Es posible aplicar sólo en los niveles especificados con

Apply[*f*, exp, nivel]

Notar que las funciones deben ser introducidas por su nombre, o bien en forma pura.

3.7.2 EJERCICIOS

1.- Construir utilizando Apply una función que teniendo como argumentos una lista de números calcule su media geométrica.

2.-Utilizando Map y Apply construir una función que se llame Laplaciano , que tenga como argumentos una función f y una lista de variables y que calcule el Laplaciano de f.

Nota: El laplaciano de una función real de varias variables f es: $\Delta f = \frac{\partial^2 f}{\partial x_1^2} + \ldots\ldots + \frac{\partial^2 f}{\partial x_n^2}$

CAPÍTULO 4
PROGRAMACIÓN BASADA EN REGLAS DE TRANSFORMACIÓN

En este capítulo exponemos los fundamentos de la programación por reglas de transformación. Este modo de programar consiste en la definición de objetos y de sus reglas de transformación. Así el objeto deja de ser un tipo de dato, para convertirse en el dato junto con las operaciones que con el se pueden realizar. Esto no nos ha de resultar extraño, es algo habitual en matemáticas.

Esta tendencia es la que ultimamente se sigue en programación y a este estilo de programar se le denomina "programación con orientación al objeto". Resulta especialmente útil cuando se tratan de programar matemáticas simbólicas. Veremos algunos ejemplos donde esta forma de programar resulta particularmente eficaz (definición de producto exterior de tensores, derivada, integral ...).

Este estilo se basa: por un lado en la construcción de modelos para expresiones y por otro en la definición de reglas de transformación para estos modelos (reglas locales y reglas globales)