Precisão Infinita

Ao calcular uma fracção em vírgula flutuante (floating point) como, por exemplo, 0,8333 representando 5/6,
ocorre um erro de precisão. Por este motivo, o Maple permite a representação de fracções de forma exata.

```
> 5/6;
```

$$\frac{5}{6}$$

```
> evalf(5/6);
```

$$0.8333333333$$

```
> evalf(5/6,100);
```

$$0.8333333333333333333333333\backslash$$
$$333333333333333333333\backslash$$
$$333333333333333333333\backslash$$
$$333333333333333333333\backslash$$
$$333333333333$$

```
> a := (34*3+7/11)^2;
```

$$a := \frac{1274641}{121}$$

```
> evalf(a);
```

$$10534.22314$$

```
> evalf(%%,40);
```

$$10534.22314049586776859504 1\backslash 3223140495867 8$$

```
> sqrt(3), sin(Pi/4);
```

$$\sqrt{3}, \frac{\sqrt{2}}{2}$$

Diferença entre o cálculo simbólico e numérico

```
> sqrt(2)*sqrt(8)-4;
```

$$0$$

```
> evalf(sqrt(2))*evalf(sqrt(8))-4;
```

$$-2.\ 10^{-9}$$

Na computação simbólica, não há erros de arredondamento
e o Maple aplica procedimentos matemáticos para a simplificação de expressões.

```
> solve(x^2+x-1,x);
```

$$-\frac{1}{2}+\frac{\sqrt{5}}{2}, -\frac{1}{2}-\frac{\sqrt{5}}{2}$$

```
> (c+d)^3;
```

$$(c+d)^3$$

```
> expand(%);
```

$$c^3+3\,c^2\,d+3\,c\,d^2+d^3$$

```
> Pi;
```

$$\pi$$

```
> evalf(Pi,1000);
```

Definição de Funções

```
> restart;
```

Em primeiro lugar é importante distinguir função de expressão algébrica. Por exemplo:

```
> f := sin(a*x);
```

$$f := \sin(a\,x)$$

```
> f(Pi);
```

$$(\sin(a\,x))\,(\pi)$$

```
> subs(x=Pi, f);
```

$$\sin(a\,\pi)$$

```
> F := unapply(sin(a*x),x);
```

$$F := x \rightarrow \sin(a\,x)$$

```
> F(Pi);
```

$$\sin(a\,\pi)$$

```
> F(y);
```

$$\sin(a\,y)$$

```
> F(x);
```

$$\sin(a\,x)$$

```
> diff(F(x), x);
```

$$\cos(a\,x)\;a$$

Uma função é definida directamente em Maple através do operador seta (->).

```
> g := x -> x^2 + 1;
```

$$g := x \rightarrow x^2 + 1$$

```
> g(2);
```

$$5$$

```
> g(a);
```

$$a^2 + 1$$

```
> g((x-3)^3);
```

$$(x - 3)^6 + 1$$

Como exemplo, vamos definir a função que dá a soma do MDC com o MMC de dois números.

```
> F := (n1,n2) -> gcd(n1,n2) + lcm(n1,n2);
```

$$F := (n1, n2) \rightarrow gcd(n1, n2) + lcm(n1, n2)$$

```
> F(25,15);
```

$$80$$

Podemos ainda definir funções por ramos (piecewise functions)

```
> f:=x -> piecewise(x<0,0,x<1,1/3,2);
```

$$f := x \rightarrow piecewise\left(x < 0, 0, x < 1, \frac{1}{3}, 2\right)$$

```
> f(x);
```

$$\begin{cases} 0 & x < 0 \\ \frac{1}{3} & x < 1 \\ 2 & \textit{otherwise} \end{cases}$$

```
> f(1);
```

$$2$$

```
> f := x -> if x<0 then 0 elif x<1 then 1/3 else 2 end if:
> f(1);
```

$$2$$

```
> f := proc(x)
  local r;
>   if x < 0 then
>     r := 0;
>   else
>     if x < 1 then
>       r := 1/3;
>     else
>       r := 2;
>     end if;
>   end if;
>   return(r);
> end proc:
> f(1);
```

$$2$$

Estruturas de Dados: Conjuntos e Listas

A diferença entre listas e conjuntos está no facto de que as listas preservam o ordenamento
e a repetição de elementos.

```
> restart;
```

```
> { 5, 5, 5, 4, 3, 2, 3, 2, 1};
```

$$\{1, 2, 3, 4, 5\}$$

```
> S1 := {1/2,3/2,5/2};
```

$$S1 := \left\{\frac{1}{2}, \frac{3}{2}, \frac{5}{2}\right\}$$

```
> S2 := {1,2,3};
```

$$S2 := \{1, 2, 3\}$$

```
> S1 union S2;
```

$$\left\{1, 2, 3, \frac{1}{2}, \frac{3}{2}, \frac{5}{2}\right\}$$

```
> % minus {3,5/2};
```

$$\left\{1, 2, \frac{1}{2}, \frac{3}{2}\right\}$$

```
> % intersect S1;
```

$$\left\{\frac{1}{2}, \frac{3}{2}\right\}$$

```
> member(z, {a,b,c});
```

$$false$$

```
> member(b, {a,b,c});
```

$$true$$

A função **member** é uma função de superfície,
pois ela não investiga sub-níveis na estrutura de dados. Por exemplo:

```
> member(1, {{1}, {2}, {3}});
```

$$false$$

O comando **has** testa se uma expressão ou estrutura de dados contem uma sub-expressão ou um elemento.

```
> has({{1}, {2}, {3}},1);
```

$$true$$

```
> expr := sqrt(a^2+b^2+c^2);
```

$$expr := \sqrt{a^2 + b^2 + c^2}$$

```
> has(expr,a);
```

$$true$$

```
> L := [a,b,c,d,e,f,g,h,i];
```

$$L := [a, b, c, d, e, f, g, h, i]$$

```
> L[2];
```

$$b$$

```
> L[-1];
```

$$i$$

```
> L[3..7];
```

$$[c, d, e, f, g]$$

```
> L[1..-1];
```

$$[a, b, c, d, e, f, g, h, i]$$

```
> [ola][1..-1];
```

$$[ola]$$

```
> [ola][1..-2];
```

$$[\ ]$$

O Maple tem comandos comuns a todas as estruturas de dados.

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

$$9$$

```
> nops({a,b,c,d});
```

$$4$$

```
> op(L);
```

$$a, b, c, d, e, f, g, h, i$$

```
> op({a,b});
```

$$a, b$$

```
> op(a+b*c);
```

$$a, b\ c$$

```
> L[2];
```

$$b$$

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

$$b$$

Sucessões/Sequências

Uma sequência é um encadeamento de objectos separados por vírgula.
O comando que gera sequências é o **seq**, cuja sintaxe é **seq(termo(i), i=a..b)**, onde a e b são números.

```
> seq( a^i, i=1..5 );
```

$$a, a^2, a^3, a^4, a^5$$

```
> seq( 11-i, i=1..10 );
```

$$10, 9, 8, 7, 6, 5, 4, 3, 2, 1$$

Uma sequência também pode ser gerada através de uma iteração **for do od**, da seguinte forma:

```
> seq1 := NULL:
> for i from 1 to 5 do
>   seq1 := seq1, termo(i):
> od:
> seq1;
```

$$termo(1), termo(2), termo(3), termo(4), termo(5)$$

```
> seq(termo(i),i=1..5);
```

$$termo(1), termo(2), termo(3), termo(4), termo(5)$$

Uma outra notação para o comando seq é seq(f(i), i=L),
onde L é uma estrutura de dados (uma lista, um conjunto, uma expressão algébrica).

```
> LF := [sin(x), x^3, ln(x), g(x)];
```

$$LF := \left[\sin(x), x^3, \ln(x), g(x)\right]$$

```
> LD := [seq(diff(f,x), f=LF)];
```

$$LD := \left[\cos(x), 3\,x^2, \frac{1}{x}, \frac{d}{dx}\,g(x)\right]$$

```
> map(diff,LF,x);
```

$$\left[\cos(x),\ 3x^2,\ \frac{1}{x},\ \frac{d}{dx}g(x)\right]$$

```
> L := NULL:
> for f in LF do
>   L := L, diff(f,x):
> end do:
> [L];
```

$$\left[\cos(x),\ 3x^2,\ \frac{1}{x},\ \frac{d}{dx}g(x)\right]$$

```
> map(x->x^2, [1,2,3]);
```

$$[1, 4, 9]$$

```
> map(x->x^3, [a,b,c,d]);
```

$$\left[a^3, b^3, c^3, d^3\right]$$

Existem dois outros comandos cuja sintaxe é igual à do **seq**
(e que também podem ser facilmente implementados com o **for)**:
o comado **add** e o comando **mul**.

O comando **add** executa um somatório.
A sua sintaxe é uma das seguintes formas:

```
> seq( a[i]*x^i, i=0..5 );
```

$$a_0, a_1 x, a_2 x^2, a_3 x^3, a_4 x^4, a_5 x^5$$

```
> add( a[i]*x^i, i=0..5 );
```

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5$$

```
> mul( a[i], i=0..3 );
```

$$a_0\ a_1\ a_2\ a_3$$

Programação Recursiva

A **programação recursiva** é uma técnica essencial para a programação no Maple
e na maioria das linguagens de programação. Esta técnica tem um paralelo com
a prova por **indução na Matemática**.

```
> restart;
> 4!;
```

$$24$$

```
> fact := n -> piecewise(n=0,1,n*fact(n-1)):
> fact(4);
```

$$24$$

```
>
> f := n -> if n=0 then 1 else n*f(n-1) fi:
> f(4);
```

$$24$$

```
> fib := n -> piecewise(n=0 or n=1,n,fib(n-1)+fib(n-2)):
> seq(fib(i),i=0..15);
```

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610$$

Exercício: Adição de Polinómios na Forma de Listas
Exemplo:

ap([2,3,0], [7,-5,1,2]);
[2,3,0] = 2*x^2 + 3*x + 0
[7,-5,1,2] = 7*x^3 -5x^2 + x + 2

soma = 7*x^3 -3x^2 + 4*x+2 = [7,-3,4,2]

```
> ap:=(L1,L2)->piecewise(nops(L1)=0, L2,
                         nops(L2)=0, L1,
```

```
                                    [op(ap(L1[1..-2],L2[1..-2])),L1[-
1]+L2[-1]]):
> ap([2,3,0], [7,-5,1,2]);
```

$$[7, -3, 4, 2]$$

Programação Funcional

Existem várias definições para programação funcional.
Considero aqui programação funcional com o seguinte sentido.
É o estilo de programação em Maple onde usamos apenas
funções definidas com o operador seta (NÃO são permitidos procedimentos).
Em particular, não é permitido usar variáveis locais a menos que elas não sofram atribuição.
Comandos que atribuem valores a contadores (como o comando for) não podem ser usados.
Porém, os comandos seq, add e mul podem ser usados.

Pode parecer uma restrição muito drástica, porém a minha experiência tem mostrado
que problemas complexos podem ser resolvidos neste estilo de programação.

select e remove

O comando **select** seleciona elementos de um conjunto ou uma lista
e devolve o mesmo tipo de estrutura de dados da entrada.

```
> S := {seq(i,i=1..20)};
```

$$S := \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20\}$$

```
> select(isprime, S);
```

$$\{2, 3, 5, 7, 11, 13, 17, 19\}$$

```
> remove(isprime, S);
```

$$\{1, 4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20\}$$

zip

O comando **zip** aplica uma função de dois argumentos nas componentes de duas listas ou vectores.

```
> zip((x,y)->x+y, [a,b,c], [alpha,beta,gamma]);
```

$$[a + \alpha, b + \beta, c + \gamma]$$

Exemplos de Programação Funcional com Listas

```
> restart;
> # inversão de uma lista
> inverte := (L::list) -> [seq(L[-i],i=1..nops(L))]:
> inverte([1,2,3,4]);
```

$$[4, 3, 2, 1]$$

```
> # posição de um elemento (primeira ocorrência)
> pos := (x,L) ->
piecewise(not(member(x,L)),false,evalb(x=L[1]),1,1+pos(x,L[2.
.-1])):
> pos(3,[a,b,c,d]);
```

$$false$$

```
> pos(b,[a,b,c,d,c]);
```

$$2$$

```
> # posições de todas as ocorrências de um elemento
> p := (x,y,i) -> if x=y then i else NULL fi:
> posicoes := (x,L) -> seq(p(x,L[i],i),i=1..nops(L)):
> posicoes(b,[a,b,c,b,d]);
```

$$2, 4$$

```
> posicoes2 := (x,L) -> seq(((x,y,i) -> if x=y then i else
NULL fi)(x,L[i],i),i=1..nops(L)):
> posicoes2(b,[a,b,c,b,d]);
```

Comunicação com Outras Linguagens

Comunicação com C

```
> eq:=int(x^2*(x+3)^(1/2),x);
```

$$eq := \frac{1}{35}\left(2\,(x+3)^{(3/2)}\left(24-12\,x+5\,x^2\right)\right)$$

```
> with(codegen):
> C(eq);
      t0 = 2.0/35.0*sqrt(pow(x+3.0,3.0))*(24.0-12.0*x+5.0*x*x);
> C(eq,optimized);

      t1 = x+3.0;
      t2 = sqrt(t1);
      t5 = x*x;
      t9 = 2.0/35.0*t2*t1*(24.0-12.0*x+5.0*t5);
```

Comunicação com Fortran

```
> fortran(eq);
      t0 = 2.D0/35.D0*sqrt(x+3)**3*(24-12*x+5*x**2)
> fortran(eq,optimized);
      t2 = sqrt(x+3)
      t3 = t2**2
      t6 = x**2
      t10 = 2.D0/35.D0*t3*t2*(24-12*x+5*t6)
```

Comunicação com Latex

```
> latex(eq);
{\frac {2}{35}}\, \left( x+3 \right) ^{3/2} \left( 24-12\,x+5\,{x}^{2}
 \right)
```

Também é possível comunicar com Matlab.
Mais detalhes podem ser conseguidos no help online do matlab.

Listagem de Comandos

Cerca de 5% dos comandos do Maple foram programados na linguagem C.
Os restantes 95% dos comandos são baseados neste pequeno grupo
de comandos e utilizam a própria linguagem Maple. É possível ver como
os comandos do Maple foram programados.
Como exemplo, suponhamos que queremos ver a listagem do comando

erf (error function).

```
> ?erf;
> interface(verboseproc=2):
> print(erf);
```

proc(*x*::*algebraic*) ...**end proc**

```
>
```