Recursion

Another way to control program flow in Maple is to use recursion. Recursion refers to a function which occurs in its own definition. This concept is well-known to mathematicians who use a lot of recursive definitions. The most famous one is probably the factorial function $n!$. Which is defined as

$$n! := \begin{cases} 1 & \textbf{if } n = 1 \\ n \cdot (n-1)! & \textbf{if } n \geq 1 \end{cases}.$$

We can convert this definition verbatimly into a Maple procedure:

```
fact := proc(n)
    if n = 1 then
        return 1
    else
        return n·fact(n − 1)
    end if
end proc
```

```
fact := proc(n) if n = 1 then return 1 else return n*fact(n − 1) end if end proc
```
(1.1)

As usual we demonstrate the correctness of the procedure:

```
fact(1) = 1!, fact(2) = 2!, fact(3) = 3!, fact(4) = 4!, fact(5) = 5!
```
$$1 = 1,\ 2 = 2,\ 6 = 6,\ 24 = 24,\ 120 = 120 \qquad \textbf{(1.2)}$$

Another famous recursive function is the Euclidean algorithm for computing the greatest common divisor of two (positive) integers.

```
euclid := proc(a, b) :
    if b = 0 then
        return a
    else
        return euclid(b, irem(a, b))
    end if
end proc:
```

Which computes correctly:

```
euclid(78, 92) = igcd(78, 92), euclid(45, 15) = igcd(45, 15), euclid(101, 67) = gcd(101, 67)
```
$$2 = 2,\ 15 = 15,\ 1 = 1 \qquad \textbf{(1.3)}$$

Recursion is usually a *divide and conquer* approach: In order to solve a problem, first check for simple cases which can be solved immeadiately. If we are not facing a simple case, try to reduce the problem into smaller problems of a similar shape. In the factorial example, the simple case is $n = 1$,

and smaller problem is the computation of $(n-1)!$ instead of the larger problem of computing $n!$. The simple cases are usally called *base cases*. It is important not to forget them. Else the procedure will recurse forever.

Remember our digit sum example. If we want to compute the digit sum of n, then this is easy if $n < 10$ because then n is the only digit. Otherwise, we can reduce the problem to computing the sum of the last digit and the digit sum of the remaining digits. In code, this looks as follows:

```
digitsum := proc(n)
    if n < 10 then
        return n
    else
        return irem(n, 10) + digitsum(iquo(n, 10))
    end if
end proc:
```

And indeed, this computes the same digit sum as the non-recursive version:

$$digitsum(6719)$$

$$23 \qquad \textbf{(1.4)}$$

We will revisit recursive functions later.

To understand recursion, you must understand recursion or ask somebody who understands recursion.

Exercise: Prove that the Euclidean algorithm really returns a greatest common divisor after finitely many steps.

Exercise: Write a recursive procedure to compute the n-th Fibonacci number for $n \geq 0$.

Data types

Maple distinguishes several *types* of data. We can for each piece of data inquire its type with the "whattype" function:

$$whattype\left(\frac{3}{4}\right)$$

$$fraction \qquad \textbf{(2.1)}$$

$$whattype(x^2-1)$$

$$`C` \qquad \textbf{(2.2)}$$

$$whattype(x \cdot y)$$

$$`*` \qquad \textbf{(2.3)}$$

$$whattype(\text{"Hello World!"})$$

$$string \qquad \textbf{(2.4)}$$

```
whattype(1.2)
```

float **(2.5)**

```
whattype(42)
```

integer **(2.6)**

```
whattype(⟨1, 2; 3, 4⟩)
```

Matrix **(2.7)**

A full list of types known to Maple can be found in the help system:

```
?type
```

We can check whether an expression has a certain type using the "type" function:

```
type(1/2, numeric)
```

true **(2.8)**

```
type(1/2, fraction)
```

true **(2.9)**

```
type(1/2, polynom)
```

true **(2.10)**

```
type(1/2, Matrix)
```

false **(2.11)**

As we see, a data value can have more than one possible type.

We can be more specific which type we are looking for:

```
type(x^2+1, polynom(integer, x))
```

true **(2.12)**

```
type(x^2+1, polynom(integer, z))
```

false **(2.13)**

```
type(⟨x+1, 1/2⟩, 'Vector'(Or(polynom(integer, x), fraction)))
```

true **(2.14)**

(Note that we have to put "Vector" into single quotes when it is used as a type. Otherwise, Maple will confuse it with a call to the "Vector" function. The same is true for "Matrix".)

Sequences and Lists

A sequence in Maple is just some expressions separated by commata ",".

$a, b, c, 1, 2, 3$

$$a, b, c, 1, 2, 3 \tag{3.1}$$

$whattype((3.1))$

$$exprseq \tag{3.2}$$

We can assign sequences to a variable:

$S := x^2 - 1, x^2 + x - 2$

$$S := x^2 - 1, x^2 + x - 2 \tag{3.3}$$

S

$$x^2 - 1, x^2 + x - 2 \tag{3.4}$$

We can use sequences in procedure calls. Maple will interpret this like the content of the sequence was written in the same place. Thus, if the sequence is the n-th argument, then Maple will not assign the complete sequence to the n-th formal parameter of the procedure, but instead it will assign the first entry of the sequence to the n-th parameter, the second to the $(n+1)$-st parameter and so on. For example, since S defined above has two entries, the following call corresponds to $gcd(p, q)$ where p is the first entry of S and q is the second one.

$gcd(S)$

$$x - 1 \tag{3.5}$$

We can mix sequences and normal parameters.

```
tmp := proc (a, b, c, d)
  print(cat("a is ", convert(a, string), ".")) :
  print(cat("b is ", convert(b, string), ".")) :
  print(cat("c is ", convert(c, string), ".")) :
  print(cat("d is ", convert(d, string), "."))
end proc:
```

```
proc(a, b, c, d)
    print(cat("a is ", convert(a, string), "."));
    print(cat("b is ", convert(b, string), "."));
    print(cat("c is ", convert(c, string), "."));
    print(cat("d is ", convert(d, string), "."))
end proc
```

(3.6)

$tmp(1, S, 2);$

"a is 1."

"b is x^2-1."

"c is x^2C x-2."

"d is 2."

(3.7)

We can use sequences in parallel assignments:

$p, q := S$

$$p, q := x^2 - 1, x^2 + x - 2 \tag{3.8}$$

p

$$x^2 - 1 \tag{3.9}$$

In order to access the entries of a sequence, the "[ ]" notation can be used. Note, that counting starts (mathematically) at 1.

$S[1]$

$$x^2 - 1 \tag{3.10}$$

$S[2]$

$$x^2 + x - 2 \tag{3.11}$$

Sequences can be generated using the "seq" command. It expects a expression (possibly depending on a parameter) and a range (for that parameter). Optionally a third value can be given which corresponds to the increase of the parameter in each step.

$seq(a_i, i = 2..8)$

$$a_2, a_3, a_4, a_5, a_6, a_7, a_8 \tag{3.12}$$

$seq(a_i, i = 2..8, 2)$

$$a_2, a_4, a_6, a_8 \tag{3.13}$$

$seq(i^2 - i - 1, i = 0..5)$

$$-1, -1, 1, 5, 11, 19 \tag{3.14}$$

Another form of "seq" allows to access the components of a expression.

$seq(t, t \text{ in } x^2 - x - 1)$

$$x^2, -x, -1 \tag{3.15}$$

$seq(f^2, f \text{ in } (x+1)\cdot(x-1))$

$$(x+1)^2, (x-1)^2 \tag{3.16}$$

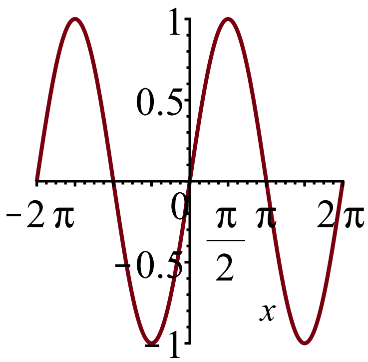
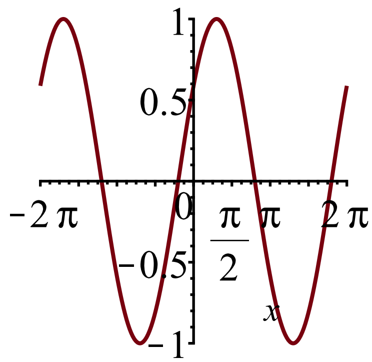
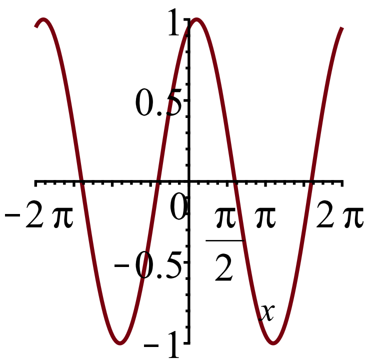
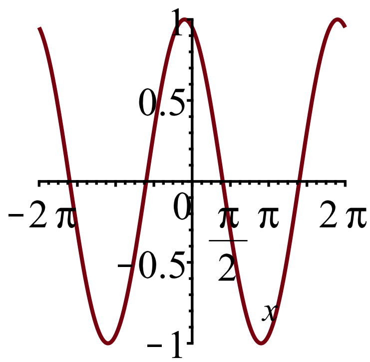
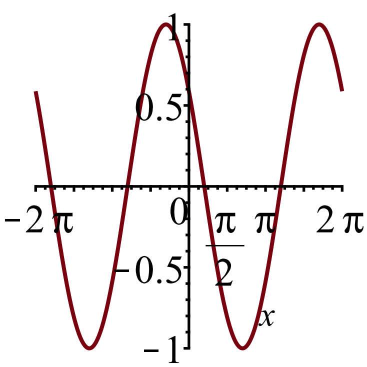
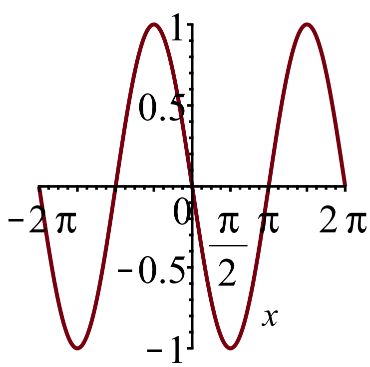
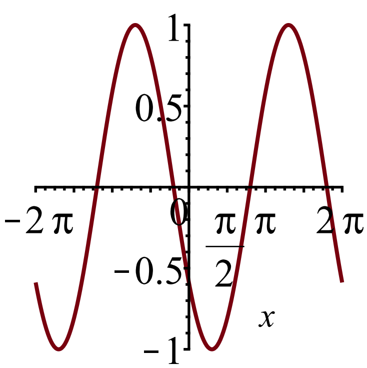
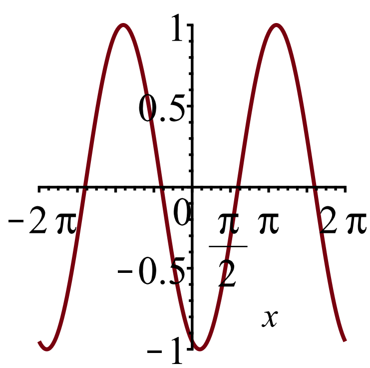
Actually, the "in" syntax work also with "add" and "mul" (= multiply):

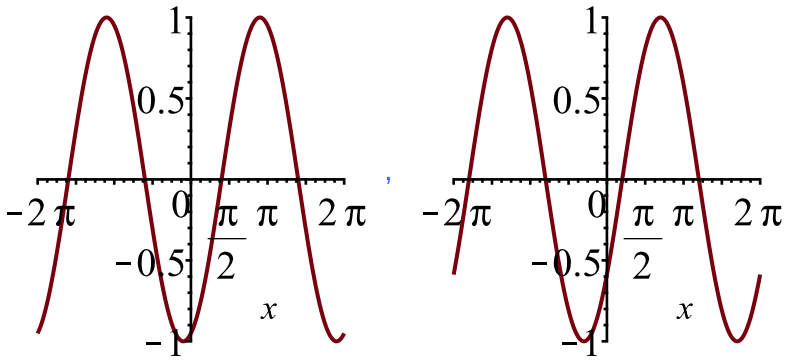
$mul(x^x, x \text{ in } a\cdot b\cdot c\cdot d)$

$$a^a b^b c^c d^d \tag{3.17}$$

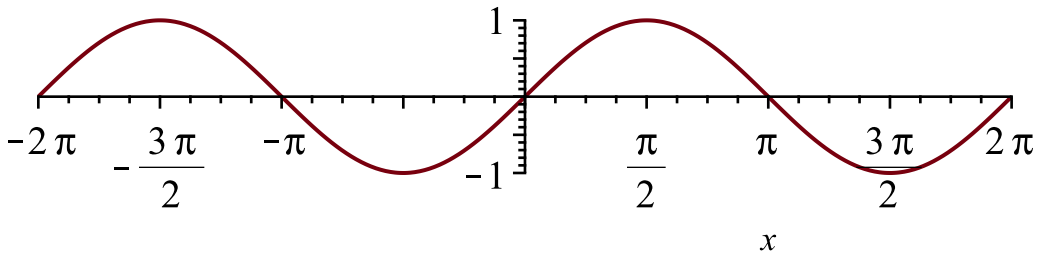
The expression in the first argument of "seq" can be arbitrarily complicated. We have been using this for creating animations. There, the expressions were plots:

$$PS := seq\left(plot\left(\sin\left(x + \frac{i\cdot\pi}{5}\right)\right), i = 0..9\right)$$

$PS :=$ 
1
0.5
-0.5
-1
-2 π
0
π/2
π
2 π
x
, 
1
0.5
-0.5
-1
-2 π
0
π/2
π
2 π
x
, 
1
0.5
-0.5
-1
-2 π
0
π/2
π
2 π
x
, 
1
0.5
-0.5
-1
-2 π
0
π/2
π
2 π
x
, 
1
0.5
-0.5
-1
-2 π
0
π/2
π
2 π
x
, 
1
0.5
-0.5
-1
-2 π
0
π/2
π
2 π
x
, 
1
0.5
-0.5
-1
-2 π
0
π/2
π
2 π
x
, 
1
0.5
-0.5
-1
-2 π
0
π/2
π
2 π
x
,



plots[*display*]([*PS*], *insequence*, *scaling* = *constrained*)



A short form of "seq" is "$".

$i^2\$i = \frac{1}{2} .. \frac{5}{2}$

$$\frac{1}{4}, \frac{9}{4}, \frac{25}{4} \qquad \textbf{(3.18)}$$

In some cases, "$" does not seem to work as nicely as "seq", though.

A list is just a sequence which is enclosed in square brackets:

[S]

$$[x^2-1, x^2+x-2] \quad \textbf{(3.19)}$$

whattype((3.19))

$$list \quad \textbf{(3.20)}$$

We can be more precise in specifying the type of a list

type([S], *list*(*polynom*(*integer*)))

$$true \quad \textbf{(3.21)}$$

type([S], *list*(*fraction*))

$$false \quad \textbf{(3.22)}$$

We can also have an empty list, denoted by

[ ]

$$[\,] \quad \textbf{(3.23)}$$

Lists can contain arbitrary entries including other lists:

$L := [a, b, [c, d], e]$

$$L := [a, b, [c, d], e] \quad \textbf{(3.24)}$$

Also list elements can be accessed using the "[ ]" notation.

$L[1]$

$$a \quad \textbf{(3.25)}$$

$L[3]$

$$[c, d] \quad \textbf{(3.26)}$$

$L[3][1]$

$$c \quad \textbf{(3.27)}$$

The last expression be also be written in the following shorthand form:

$L[3, 1]$

$$c \quad \textbf{(3.28)}$$

Instead of single elements, we can also extract sublists by supplying a range to the "[ ]" operator:

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

$$L := [a, b, c, d, e, f, g] \quad \textbf{(3.29)}$$

$L[1..3]$

$$[a, b, c] \quad \textbf{(3.30)}$$

$L[3..5]$

$$[c, d, e] \quad \textbf{(3.31)}$$

If one end of the range is omitted, Maple extends it to the corresponding end of the list.

$L[3..]$

$$[c, d, e, f, g] \tag{3.32}$$

$L[..3]$

$$[a, b, c] \tag{3.33}$$

Additionally, negative indices are supported. They are interpreted as counting from the last element:

$L[-1]$

$$g \tag{3.34}$$

$L[-2]$

$$f \tag{3.35}$$

Negative indices may also be used with ranges.

$L[1..-2]$

$$[a, b, c, d, e, f] \tag{3.36}$$

$L[-3..-2]$

$$[e, f] \tag{3.37}$$

Actually, ranges and negative values also work for matrices:

$A := \langle 1, 2, 3; 4, 5, 6; 7, 8, 9 \rangle$

$$A := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \tag{3.38}$$

$A[1..2, 2..-1]$

$$\begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix} \tag{3.39}$$

For all, lists, matrices and sequences, in the document mode we can also use index notation to access elements or substructures (type, eg, "S _ 1").

$A_{1..2,\,1}$

$$\begin{bmatrix} 1 \\ 4 \end{bmatrix} \tag{3.40}$$

L_{-2}

$$f \tag{3.41}$$

S_1

$$x^2 - 1 \tag{3.42}$$

We can change the entries of a list just as we did with matrices:

$L[1] := 3$

$$L_1 := 3 \tag{3.43}$$

L

$$[3, b, c, d, e, f, g] \tag{3.44}$$

Another way of accessing list entries uses the "op" command with a numeric index.

$op(2, L)$

$$b \tag{3.45}$$

We can change list entries also with the "subsop" (substitute operant) command. This returns a new list, the original list remains unchanged.

$subsop(2 = y, L)$

$$[3, y, c, d, e, f, g] \tag{3.46}$$

L

$$[3, b, c, d, e, f, g] \tag{3.47}$$

By substituting the special symbol "NULL", we can delete elements from lists. Again, this returns a new list.

$subsop(3 = NULL, L)$

$$[3, b, d, e, f, g] \tag{3.48}$$

L

$$[3, b, c, d, e, f, g] \tag{3.49}$$

The number of elements of a list can be accessed with "numelemens" (just like with sets).

$numelems(L)$

$$7 \tag{3.50}$$

We can ask Maple for the underlying sequence of a list with the "op" command:

$op(L)$

$$3, b, c, d, e, f, g \tag{3.51}$$

$whattype((3.51))$

$$exprseq \tag{3.52}$$

We can use this to add elements on either side of the list:

$[a, op(L)]$

$$[a, 3, b, c, d, e, f, g] \tag{3.53}$$

$[op(L), x]$

$$[3, b, c, d, e, f, g, x] \tag{3.54}$$

We can even use it to concatenate lists.

$[op(L), op(L)]$

$$[3, b, c, d, e, f, g, 3, b, c, d, e, f, g] \quad \textbf{(3.55)}$$

Using the "$" operator explained above, we can model the *list comprehensions* of languages like Python or Haskell in Maple.

$\{x_i \$ i = 1..5\}$

$$\{x_1, x_2, x_3, x_4, x_5\} \quad \textbf{(3.56)}$$

$[\,[x + y \$ x = 0..3] \$ y = 1..4\,]$

$$[[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6], [4, 5, 6, 7]] \quad \textbf{(3.57)}$$

The package "ListTools" contains a lot of functions for lists.

Another detail on loops

The keyword "in" can also used with loops:

```
for x in [1, 2, 4, 8, 16] do
    print(cat("x is now ", x, "."))
end do
```

```
"x is now 1."
"x is now 2."
"x is now 4."
"x is now 8."
"x is now 16."
```
(4.1)

It can pick the operands from an arbitrary expression:

```
for x in expand((a + b)^7) do
    print(cat("x is ", convert(x, string), "."))
 end do
```

```
"x is a^7."
"x is 7*a^6*b."
"x is 21*a^5*b^2."
"x is 35*a^4*b^3."
"x is 35*a^3*b^4."
"x is 21*a^2*b^5."
"x is 7*a*b^6."
```

"x is b^7." **(4.2)**

We could use this in the following way to write a function which flattens a list once, ie, removes one level of sublists.

```
flattenOnce := proc(lst) local L, x:
   L := [ ]:

   for x in lst do
      if type(x, list) then
         L := [op(L), op(x)]
      else
         L := [op(L), x]
      end if
   end do:

   return L
end proc:
```

And indeed, this removes one layer of brackets from a list:

```
flattenOnce([a, [b, c], [d, e, f, g], [h, [i, j]], k])
```

$$[a, b, c, d, e, f, g, h, [i, j], k] \quad \textbf{(4.3)}$$

Exercise: Write a procedure "flatten" which removes all (but the outer) brackets from a list. (Hint: The easiest solution uses recursion.)

More recursion

Definition: Let S be any set. The *power set* of S is the set of all T such that $T \subset S$, that is, the set of all subsets of S.

As a more involved example for recursion, we compute the power set of given set. The idea here is the following: First of all, if $S = \emptyset$, then the powerset contains only S itself. (Since every set is a subet of itself, the power set always contains the set itself.) Otherwise, if S is not empty, then there exists an element $x \in S$. For any subset $T \subset S$ there are two possibilities: Either $x \in T$ or $x \notin T$. In the second case, T is a subset of $S \setminus \{x\}$. In the first case, $T \setminus \{x\}$ is a subset of $S \setminus \{x\}$. This suggest, that we can write all subsets of S as subsets of $S \setminus \{x\}$ for some $x \in S$ with the possible addition of x. This is exactly the approach which is taken by the procedure below. It distinguishes the cases $S = \emptyset$ and $S \neq \emptyset$, and in the second case we compute for a random $x \in S$ recursively the power set of $S \setminus \{x\}$ and for each $T \subset S \setminus \{x\}$, we add T and $T \cup \{x\}$ to the power set of S.

```
powerset := proc(S) local P, x, T:
   if S = ∅ then
      return {∅}
   else
```

```
    P := ∅:
    x := S[1]:
    for T in powerset(S \ {x}) do
      P := P ∪ {T, T ∪ {x}}
    end do:
    return P
  end if
end proc:
```

As usual, we test the procedure:

$$powerset(\{a, b, c\}) \tag{5.1}$$
$$\{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$$

$$powerset(\{1, 2\}) \tag{5.2}$$
$$\{\varnothing, \{1\}, \{2\}, \{1, 2\}\}$$