Assignments again

We already learned that Maple uses the ":=" operator for assigment.

$x := 5$

$$x := 5 \tag{1.1}$$

x

$$5 \tag{1.2}$$

This operator works *destructively*: If we assign another value to x, then the old value will be lost.

$x := 3$

$$x := 3 \tag{1.3}$$

x

$$3 \tag{1.4}$$

We can use x itself on the right hand side of an assignment. Maple will always completely evaluate the right hand side, before the assignment is done. Here, that means, that the old value of x is used to compute the new value.

$x := x - 1$

$$x := 2 \tag{1.5}$$

x

$$2 \tag{1.6}$$

Maple can do *parallel assignments.*

$a, b := 1, 2$

$$a, b := 1, 2 \tag{1.7}$$

a

$$1 \tag{1.8}$$

b

$$2 \tag{1.9}$$

Again, the right hand side is completely evaluated before the assignment is made. Compare

$a, b := b, a$

$$a, b := 2, 1 \tag{1.10}$$

a

$$2 \tag{1.11}$$

b

$$1 \tag{1.12}$$

with

```
a := b; b := a;
```

$$a := 1$$
$$b := 1 \quad \textbf{(1.13)}$$

```
a
```

$$1 \quad \textbf{(1.14)}$$

```
b
```

$$1 \quad \textbf{(1.15)}$$

Introductionary example: Digit sum

As our first taste of programming, we want to compute the digit sum of a (non-negative) integer.

For simplicity, assume first that our number has exactly 4 digits. We could simply assign them to variables and compute their sum. So, in order to compute the digit sum of 6719, say, we could do

```
a, b, c, d := 6, 7, 1, 9
```

$$a, b, c, d := 6, 7, 1, 9 \quad \textbf{(2.1)}$$

```
a + b + c + d
```

$$23 \quad \textbf{(2.2)}$$

If we want to compute the digit sum of another number, we have to go back, change the assignment and reevaluate the two lines.

We can make things easier for us, by wrapping the computation in a function. Maple offers two choices here: While the usual mathematical functions would be sufficient for a simple problem as this, we want to introduce the "procedure" notation which is more suited for the longer programs which we will write below.

```
digitsum := proc(a, b, c, d) :
    return a + b + c + d :
end proc
```

$$digitsum := \textbf{proc}(a, b, c, d)\ \textbf{return}\ a + b + c + d\ \textbf{end proc} \quad \textbf{(2.3)}$$

In order to input multiple lines to Maple use "Shift+Return" and not "Return" alone. Use "Return", when you are done with the input. As we can see in the output, Maple makes no difference between single or more spaces or newlines (or other kinds of "white space").

We can apply our digit sum function just like any Maple function:

```
digitsum(6, 7, 1, 9)
```

$$23 \quad \textbf{(2.4)}$$

Of course, having the user of our function to split the number into digits by hand is not very elegant. There is a better way to do this using the built-in "irem" and "iquo" functions ("integer remainder" and "integer quotient"). Taking the remainder modulo 10, will yield the last digit, while taking the quotient

modulo 10 removes the last digit.

$number := 6719$

$$number := 6719 \tag{2.5}$$

$irem(number, 10)$

$$9 \tag{2.6}$$

$iquo(number, 10)$

$$671 \tag{2.7}$$

We can use this, to extract all digits.

```
rest := number;
d := irem(rest, 10);
rest := iquo(rest, 10);
c := irem(rest, 10);
rest := iquo(rest, 10);
b := irem(rest, 10);
rest := iquo(rest, 10);
a := irem(rest, 10);
rest := iquo(rest, 10);
```

$$rest := 6719$$
$$d := 9$$
$$rest := 671$$
$$c := 1$$
$$rest := 67$$
$$b := 7$$
$$rest := 6$$
$$a := 6$$
$$rest := 0 \tag{2.8}$$

Note the use of ";" to separate statements. (We can also separate statements by ":" with the difference that ":" will suppress the printing of the result.) Use again "Shift-Return" for multiline input.

We can check our computation:

$number = 1000 \cdot a + 100 \cdot b + 10 \cdot c + d$

$$6719 = 6719 \tag{2.9}$$

We include this scheme into our procedure.

```
digitsum := proc(number) # Here a ":" is not allowed!
local a, b, c, d, rest :

   rest := number;
   d := irem(rest, 10);
   rest := iquo(rest, 10);
```

```
    c := irem(rest, 10);
    rest := iquo(rest, 10);
    b := irem(rest, 10);
    rest := iquo(rest, 10);
    a := irem(rest, 10);

    return a + b + c + d :
 end proc;
```

```
digitsum := proc(number)                                                    (2.10)
    local a, b, c, d, rest;
    rest := number;
    d := irem(rest, 10);
    rest := iquo(rest, 10);
    c := irem(rest, 10);
    rest := iquo(rest, 10);
    b := irem(rest, 10);
    rest := iquo(rest, 10);
    a := irem(rest, 10);
    return a + b + c + d
end proc
```

The "local" keyword is used to introduce fresh variables which do not collide with assignments made outside the procedure: Although in the example the variable "a" is known outside the procedure "exmpl", Maple ignores its outside definition due to the "local" keyword. This is not true for "b" since we did not declare "b" to be local.

```
a := 4; b := 5;
exmpl :=proc( ) local a : print( 'a' = a ) : print( 'b'= b ) : end proc;
```

$$a := 4$$

$$b := 5$$

$$exmpl := \mathbf{proc}(\,)\ \mathbf{local}\ a;\ print('a' = a);\ print('b' = b)\ \mathbf{end\ proc} \tag{2.11}$$

```
exmpl( ) :
a
```

$$a = a$$

$$b = 5$$

$$4 \tag{2.12}$$

Coming back to our digit sum, we can input our number just as integer:

```
digitsum(6719)
```

$$23 \tag{2.13}$$

We can reduce the number of variables needed in our procedure, by computing the sum "on the way":

```
digitsum := proc(number)
local rest, total :

    rest := number :
    total := 0 :

    total := total + irem(rest, 10) :
    rest := iquo(rest, 10) :
    total := total + irem(rest, 10) :
    rest := iquo(rest, 10) :
    total := total + irem(rest, 10) :
    rest := iquo(rest, 10) :
    total := total + irem(rest, 10) :

    return total :
end proc;
```

```
digitsum := proc(number)                                    (2.14)
    local rest, total;
    rest := number;
    total := 0;
    total := total + irem(rest, 10);
    rest := iquo(rest, 10);
    total := total + irem(rest, 10);
    rest := iquo(rest, 10);
    total := total + irem(rest, 10);
    rest := iquo(rest, 10);
    total := total + irem(rest, 10);
    return total
end proc
```

(Note that results of computations *inside* a procedure are not printed. Thus, it does not matter whether we use ":" or ";" here.)The new version computes the same result as the old definition:

```
digitsum(6719)
```

23 **(2.15)**

The last procedure had the same pair of repeating four times. We can do better than that by using a *loop*. Loops are very important in programming, and we will discuss them in more detail later.

```
digitsum := proc(number)
 local rest, total :

    rest := number :
    total := 0 :

    from 1 to 4 do
       total := total + irem(rest, 10) :
```

```
    rest := iquo(rest, 10) :
  end do:

  return total :
end proc;
```

```
digitsum := proc(number)                                                    (2.16)
  local rest, total;
  rest := number;
  total := 0;
  to 4 do total := total + irem(rest, 10); rest := iquo(rest, 10) end do;
  return total
end proc
```

(Note that in the output Maple simplifies the statement for the loop a little bit. The version is equivalent to ours.)We can test the procedure to see that it yields the same result again:

```
digitsum(6719)
```

$$23 \qquad (2.17)$$

Since the upper bound for the loop does not to be fixed but can be computed by Maple on the fly, we may modify the last incarnation of our procedure to accept numbers with an arbitrary number of digits:

```
digitsum :=proc(number)
 local rest, total :

  rest := number :
  total := 0 :

  from 1 to length(number) do
    total := total + irem(rest, 10) :
    rest := iquo(rest, 10) :
  end do:

  return total :
end proc;
```

```
digitsum := proc(number)                                                    (2.18)
  local rest, total;
  rest := number;
  total := 0;
  to length(number) do total := total + irem(rest, 10); rest := iquo(rest, 10) end do;
  return total
end proc
```

We test it with several numbers of different length:

digitsum(111111) # *Ie, 111,111*

$$6 \tag{2.19}$$

digitsum(11111111) # *Ie, 11,111,111*

$$8 \tag{2.20}$$

Exercise: Prove that a number is divisible by 3 (resp. 9) if and only if its digit sum is divisible by 3 (resp. 9). (Hint: Note that 9, 99, 999, etc are all divisible by 3 and by 9).

Exercise: The *alternating digit sum* is defined as

$$altdigitsum\left(\sum_{j=0}^{n} a_j \cdot 10^j\right) = \sum_{j=0}^{n} a_j \cdot (-1)^j$$

where $n \geq 0$ and $0 \leq a_0, \ldots, a_n \leq 9$. For our running example 6719 the alternating digit sum is $-6 + 7 - 1 + 9 = 9$. Write a procedure to compute the alternating digit sum for an arbitrary positive integer.

Exercise: Prove that a number is divisble by 11 if and only if its alternating digit sum is.

Introductionary example: Matrix multiplication

As we have seen before, Maple does have matrix multiplication built-in via the "." (dot) operator. However, it make a good example for the usage of loops.

We have seen that we can create Matrices either via the "Matrix" command or via the "< >" notation:

Matrix(2, 3, [1, 2, 3, 4, 5, 6]) = ⟨⟨1|2|3⟩, ⟨4|5|6⟩⟩

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{3.1}$$

If we do not give Maple a list of entries, it will create a zero matrix:

Matrix(2, 3)

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{3.2}$$

Yet another notation uses again "< >".

$A := ⟨1, 2, 3; 4, 5, 6⟩$

$$A := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{3.3}$$

We can access individual entries of the matrix by using the "[ ]" notation:

$A[1, 2]$
$A[2, 3]$

$$6 \tag{3.4}$$

Unlike in the mathematical world, Maple also allows to change entries within a matrix. For this, we use the "[ ] :=" notation:

$A[1, 2] := 9$

$$A_{1,2} := 9 \tag{3.5}$$

A

$$\begin{bmatrix} 1 & 9 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{3.6}$$

$A[2, 3] := -1$

$$A_{2,3} := -1 \tag{3.7}$$

A

$$\begin{bmatrix} 1 & 9 & 3 \\ 4 & 5 & -1 \end{bmatrix} \tag{3.8}$$

We can ask Maple for the size of a matrix using the "Dimension" function of the "LinearAlgebra" package:

$LinearAlgebra:\text{-}Dimension(A)$

$$2, 3 \tag{3.9}$$

This is already almost everything, which we need in order to implement our own matrix multiplication routine. Recall that the product of A and B where A is a $m \times n$ matrix and B is a $n \times p$ matrix is the $m \times p$ matrix C with entries $C[i, j] = \sum_{k=1}^{n} A[i, k] \cdot B[k, j]$. For our procedure we will create a zero matrix C of appropriate size and then loop over all its positions in order to fill them up with the above sum. For this we an extension to the "from x to y" loops we have already seen where Maple tells us in which iteration we actually are:

```
for i from 1 to 10 do
    print('i'= i)
 end do
```

$i = 1$
$i = 2$
$i = 3$
$i = 4$
$i = 5$
$i = 6$

$$i = 7$$
$$i = 8$$
$$i = 9$$
$$i = 10 \quad \textbf{(3.10)}$$

Now, we can implement matrix multiplication (note the parallel assignment and the use of ":" at the very end to prevent Maple from printing the procedure):

```
matrixmultiplication := proc(A, B)
local m, n1, n2, p, i, j, k, C, total :

    m, n1 := LinearAlgebra:-Dimension(A) :
    n2, p := LinearAlgebra:-Dimension(B) :

    # We assume that n1 = n2

    C := Matrix(m, p) :

    for i from 1 to m do
       for j from 1 to p do

          total := 0 :
          for k from 1 to n1 do
             total := total + A[i, k]·B[k, j]
          end do:
          C[i, j] := total

       end do: # j-loop
    end do: # i-loop

    return C :
 end proc:
```

We can test our procedure against the builtin "." operator:

$B := \langle 1, 1; 2, 1; 2, 3 \rangle$

$$B := \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 2 & 3 \end{bmatrix} \quad \textbf{(3.11)}$$

$matrixmultiplication(A, B) = A.B$

$$\begin{bmatrix} 25 & 19 \\ 12 & 6 \end{bmatrix} = \begin{bmatrix} 25 & 19 \\ 12 & 6 \end{bmatrix} \quad \textbf{(3.12)}$$

We may write our procedure more elegantly by computing the sums already in $C[i, j]$, ie, by not using an additional variable:

```
matrixmultiplication := proc(A, B)
local m, n1, n2, p, i, j, k, C:

    m, n1 := LinearAlgebra:-Dimension(A):
    n2, p := LinearAlgebra:-Dimension(B):

    # We assume that n1 = n2

    C := Matrix(m, p):

    for i from 1 to m do
       for j from 1 to p do
          for k from 1 to n1 do
             C[i, j] := C[i, j] + A[i, k]·B[k, j]
          end do:
       end do: # j-loop
    end do: # i-loop

    return C:
 end proc:
```

(The "#" introduces a *comment* in Maple. Maple will ignore everything from "#" to the end of the same line. Comments can be used to provide additional information about a program to human readers.) Again, we test our procedure:

```
matrixmultiplication(A, B) = A.B
```

$$\begin{bmatrix} 25 & 19 \\ 12 & 6 \end{bmatrix} = \begin{bmatrix} 25 & 19 \\ 12 & 6 \end{bmatrix} \qquad \textbf{(3.13)}$$

Actually, we could even be more elegant by not using loop to compute the sum but by using instead the built-in "add" command:

```
matrixmultiplication := proc(A, B)
local m, n1, n2, p, i, j, k, C:

    m, n1 := LinearAlgebra:-Dimension(A):
    n2, p := LinearAlgebra:-Dimension(B):

    # We assume that n1 = n2

    C := Matrix(m, p):

    for i from 1 to m do
       for j from 1 to p do
          C[i, j] := add(A[i, k]·B[k, j], k = 1 ..n1)
```

```
    end do: # j-loop
  end do: # i-loop

  return C:
end proc:
```

Once more, we test the procedure

$$matrixmultiplication(A, B) = A.B$$

$$\begin{bmatrix} 25 & 19 \\ 12 & 6 \end{bmatrix} = \begin{bmatrix} 25 & 19 \\ 12 & 6 \end{bmatrix} \quad \textbf{(3.14)}$$

There is an even more elegant way to write this procedure, which we will see later in the lecture.

Exercise: Write a procedure for the addition matrices in the same fashion as in the example.

Exercise: Write a similar procedure for scalar multiplication.

Exercise: For $a_1,..., a_n$ in a suitable domain, the *Vandermonde matrix* is defined as

$$\begin{bmatrix} 1 & a_1 & \cdots & a_1^{n-1} \\ 1 & a_2 & \cdots & a_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & a_n & \cdots & a_n^{n-1} \end{bmatrix}.$$

Write a procedure which has as input a vector (of arbitrary dimension) and produces the Vandermonde matrix of its entries. (Of course, you should not use the built-in "VandermondeMatrix" procedure in the "LinearAlgebra" package.)

Exercise: Prove that the determinant of a Vandermonde matrix is non-zero if and only if $a_i \neq a_j$ for all $i \neq j$. Use this to prove that for every list of datapoints $(x_1, y_1),..., (x_n, y_n)$ there exists exactly one polynomial p of degree $n-1$ such that $p(x_i) = y_i$ for all $i = 1,..n$.

Short excursus on strings

Maple allows to handle arbitrary chunks of text. For this, the text has to be enclosed in double quotes ("). Such a piece of text is called a "string". Strings evaluate to themselves when we enter them to Maple:

```
"This is a string."
```

"This is a string." **(4.1)**

Used together with the "print" command, strings are useful to display messages to the user from

within a longer computation.

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

"Hello World!" **(4.2)**

Maple allows to convert arbitrary expressions into strings by using the "convert" command.

$$convert\left(x^2+\frac{1}{2}, string\right)$$

"9/2" **(4.3)**

Another operation which Maple offers for string is concatenation, ie, the combination of several smaller strings into a single larger one.

$$cat(\text{"Hello"}, \text{" "}, \text{"World!"})$$

"Hello World!" **(4.4)**

The "cat" command accepts an arbitrary number of arguments. Only the first has to be a string or a name (= unevaluated variable), the others can be arbitrary.

$i := 5 : j := 6 :$

$$cat(\text{"The value of i is "}, i, \text{", and the value of j is "}, j, \text{"."})$$

"The value of i is 5, and the value of j is 6." **(4.5)**

More complicated expressions should be converted into strings before used with "cat".

Maple also has a "sprintf" function which works similar to the function of the same name in the C programming language. It expects a format string as a first argument and a number of values as the remaining arguments. The format string can contain slots which are filled in with the other arguments in the same order in which they are given, and the result is returned as a string. There are different types of slots. The most useful one is "%a" (where the "a" stands for anything) which converts any Maple object into a string (just like "convert" does).

$$sprintf(\text{"An integer: \%a, and another integer: \%a."}, 2, 3)$$

"An integer: 2, and another integer: 3." **(4.6)**

$$sprintf(\text{"A matrix: \%a, and a polynomial: \%a."}, \langle 1, 2; 3, 4\rangle, x^2-x-1)$$

"A matrix: Matrix(2, 2, [[1,2],[3,4]]), and a polynomial: 1." **(4.7)**

Maple also has the "printf" function which works exaclty the same except that the strings are printed directly and not returned.

There exist more types of slots like, for example, "%d" for integers and "%f" for floating point numbers. It is also possible to specify the formats further. For instance, one can specify how many digits of a float should get printed. The syntax is the same as in C. See the Help page for more information and examples.

$?sprintf$

More on loops

Above, we have seen how we can use "for" loops to iterate over a range of numbers:

```
for i from 0 to 5 do
    print(cat("The value of i is now ", i, "."))
end do
```

```
"The value of i is now 0."
"The value of i is now 1."
"The value of i is now 2."
"The value of i is now 3."
"The value of i is now 4."
"The value of i is now 5."
```

(5.1)

By default, Maple increases the loop variable *i* by one in each step. We can change this, by adding a "by" clause:

```
for i from 0 to 5 by 2 do
    print(cat("The value of i is now ", i, "."))
end do
```

```
"The value of i is now 0."
"The value of i is now 2."
"The value of i is now 4."
```

(5.2)

We can use negative increments to count down:

```
for i from 10 to 0 by -1 do
    print(cat("The value of i is now ", i, "."))
end do
```

```
"The value of i is now 10."
"The value of i is now 9."
"The value of i is now 8."
"The value of i is now 7."
"The value of i is now 6."
"The value of i is now 5."
"The value of i is now 4."
"The value of i is now 3."
"The value of i is now 2."
"The value of i is now 1."
"The value of i is now 0."
```

(5.3)

We are even allowed to use non-integer increments (note that we use "convert" to print the fractions more nicely):

for *i* **from** 0 **to** 5 **by** $\frac{2}{3}$ **do**
  print(*cat*("The value of i is now ", *convert*(*i*, *string*), "."))
end do

"The value of i is now 0."
"The value of i is now 2/3."
"The value of i is now 4/3."
"The value of i is now 2."
"The value of i is now 8/3."
"The value of i is now 10/3."
"The value of i is now 4."
"The value of i is now 14/3." **(5.4)**

We have seen that we are allowed to leave out the "for" clause if we are not interested in the loop variable. Also the "by" clause is obviously optional. The same is actually true for the "from" clause as well: If we leave it out, Maple will always start counting from 1.

for *i* **to** 5 **do**
  print(*cat*("The value of i is now ", *i*, "."))
end do

"The value of i is now 1."
"The value of i is now 2."
"The value of i is now 3."
"The value of i is now 4."
"The value of i is now 5." **(5.5)**

In fact, we are also allowed to leave out the "to" clause. In that case, however, we must be careful to provide some other criterion to terminate the loop. Otherwise, Maple would just loop on forever!

One way of providing such a criterion is to add a "while" clause. A loop of the form "**while** *cond* **do**..." will run as long as *cond* evaluates to *true*. It will stop, when *cond* becomes *false*. This form of a loop is very useful for more complicated termination conditions.

for *i* **from** 1 **while** $i^3 \le 100$ **do**
  print(*cat*("The value of i is now ", *i*, " and i^3 is ", i^3, "."))
end do

"The value of i is now 1 and i^3 is 1."
"The value of i is now 2 and i^3 is 8."
"The value of i is now 3 and i^3 is 27."

"The value of i is now 4 and i^3 is 64." **(5.6)**

Often "while" loops are used in the following form which allows for non-linear increments of the loop variable:

```
i := 1 :
while i ≤ 1000 do
    print(cat("i = ", i, "."));
    i := 2·i :
 end do:
```

```
"i = 1."
"i = 2."
"i = 4."
"i = 8."
"i = 16."
"i = 32."
"i = 64."
"i = 128."
"i = 256."
"i = 512."
```
(5.7)

Be careful! A very common mistake is to forget to change the loop variable within the loop. This will make the computation run forever.

There are other statements connected to loops ("break" and "next") which we will discuss later. Also later, we will discuss a special form for looping over lists.

As another example for a loop, we compute the n-th Fibonacci number. Recall that the Fibonacci series is defined by

$$F_0 = 0, \quad F_1 := 1, \quad \text{and} \quad F_n := F_{n-1} + F_{n-2} \text{ for all } n \geq 2.$$

We use a loop which remembers the last two Fibonacci numbers F_n and F_{n+1} computed starting with F_0 and F_1. In each iteration of the loop, we Compute F_{n+2} and overwrite the last values.

```
fibonacci := proc(n)
local last, secondlast :
    last, secondlast := 1, 0 :

    from 1 to n do
       last, secondlast := last + secondlast, last
    end do:

    return secondlast :
 end proc:
```

As usual, we should test our procedure:

$fibonacci(0), fibonacci(1), fibonacci(2), fibonacci(3), fibonacci(4), fibonacci(5), fibonacci(6)$

0, 1, 1, 2, 3, 5, 8 **(5.8)**

Exercise: Print all terms of the for $2^i \cdot 3^j$ for $0 \le i + j \le 10$.

Exercise: *Chebyshev polynomials of the first kind* are defined via the relations:

$$T_0(x) = 1, \quad T_1(x) = x, \quad \text{and} \quad T_n(x) = 2\,x \cdot T_{n-1}(x) - T_{n-2}(x) \quad \text{for all } n \ge 2.$$

Write a procedure to compute the *n*-th Chebychev polynomial.

Branching

Very often, we will need to vary our computations depending on a condition. As an easy example: The linear equation "$a \cdot x = 1$" has a solution in x if and only if $a \neq 0$.

Maple uses the "if-then" statement to decide whether to do a certain computation or not:

```
i := 10 :
if i ≥ 5 then
    print("i is greater or equal to 5.")
 end if:
 if i < 5 then
    print("i is strictly smaller than 5.")
 end if
```

"i is greater or equal to 5." **(6.1)**

```
i := 1 :
if i ≥ 5 then
    print("i is greater or equal to 5.")
 end if:
 if i < 5 then
    print("i is strictly smaller than 5.")
 end if
```

"i is strictly smaller than 5." **(6.2)**

The above case where we want to perform one computation when a condition is true and another computation when the same condition is false is so common that Maple has a special syntax for it: the "if-then-else" statement:

```
i := 8 :
if i ≥ 5 then
   print("i is greater or equal to 5.")
else
    print("i is NOT greater or equal to 5.")
 end if
```

"i is greater or equal to 5." **(6.3)**

```
i := 2 :
if i ≥ 5 then
   print("i is greater or equal to 5.")
else
    print("i is NOT greater or equal to 5.")
 end if
```

"i is NOT greater or equal to 5." **(6.4)**

Sometimes, we will have to distuinguish between more than two cases. For this, "elif" clauses (= "else if") are used. We can use arbitrarily many "elif" clauses in the same "if" statement.

```
i := 6 :
if i > 7 then
    print("i is larger than 7.")
 elif i = 7 then
    print("i equals 7.")
 elif i = 6 then
    print("i equals 6.")
 else
    print("i is smaller than 6.")
 end if
```

"i equals 6." **(6.5)**

As an example for the usefulness of this, we compute roots of the quadratic polynomial $a \cdot x^2 + b \cdot x + c$:

```
quadraticRoots := proc(a, b, c)
local discr :
    discr := b^2 - 4·a·c :

   if discr = 0 then
      return -b/(2·a)
   elif discr > 0 then
      return (-b + sqrt(discr))/2, (-b - sqrt(discr))/2
   else
```

```
    return (-b + I·sqrt(-discr))/(2·a), (-b - I*sqrt(-discr))/(2·a)
  end if
end proc:
```

We can test our procedure:

```
quadraticRoots(1, 0, -1)
```

$$1,\ -1 \tag{6.6}$$

Passing the polynomial as series of coefficients is not very elegant. We can use Maple's "coeff" command to extract the coefficients of any polynomial:

```
unassign('x'):
p := 4·x^2 + 5·x + 1
```

$$p := 4\,x^2 + 5\,x + 1 \tag{6.7}$$

```
coeff(p, x, 2)
```

$$4 \tag{6.8}$$

```
coeff(p, x, 10)
```

$$0 \tag{6.9}$$

With this, we can rewrite our function as:

```
quadraticRoots := proc(p)
local discr, a, b, c:
  a := coeff(p, x, 2):
  b := coeff(p, x, 1):
  c := coeff(p, x, 0):
  discr := b^2 - 4·a·c:

  if discr = 0 then
    return -b/(2·a)
  elif discr > 0 then
    return (-b + sqrt(discr))/2, (-b - sqrt(discr))/2
  else
    return (-b + I·sqrt(-discr))/(2·a), (-b - I*sqrt(-discr))/(2·a)
  end if
end proc:
```

We test it with

```
quadraticRoots(x^2 + 2)
```

$$I\sqrt{2}, -I\sqrt{2} \qquad \textbf{(6.10)}$$

The function does not return correct results for polynomials of degree other than 2. Using Maple's "degree" function and an "if-then" statement, we can give a friendly warning to the user of our procedure if the degree of the input is too large using the "error" statement.

```
quadraticRoots := proc(p)
local discr, a, b, c :
   if degree(p, x) ≠ 2 then
      error "Sorry, I can only handle quadratic polynomials."
   end if:

   a := coeff(p, x, 2) :
   b := coeff(p, x, 1) :
   c := coeff(p, x, 0) :
   discr := b^2 - 4·a·c :

   if discr = 0 then
      return -b/(2·a)
   elif discr > 0 then
      return (-b + sqrt(discr))/2, (-b - sqrt(discr))/2
   else
      return (-b + I·sqrt(-discr))/(2·a), (-b - I*sqrt(-discr))/(2·a)
   end if
end proc:
```

Indeed, we now get our error message if the input is not of the correct degree:

```
quadraticRoots(x^3 - 5)
```

Error, (in quadraticRoots) Sorry, I can only handle quadratic polynomials.

But for input of degree 2, everything works as before:

```
quadraticRoots(x^2 - x - 1)
```

$$\frac{1}{2} + \frac{\sqrt{5}}{2}, \frac{1}{2} - \frac{\sqrt{5}}{2} \qquad \textbf{(6.11)}$$

Exercise: The song "99 bottles of beer" has the lyrics:

Ninety-nine bottles of beer on the wall, Ninety-nine bottles of beer.

Take one down, pass it around, Ninety-eight bottles of beer on the wall.

Ninety-eight bottles of beer on the wall, Ninety-nine bottles of beer.
Take one down, pass it around, Ninety-eight bottles of beer on the wall.

⋮

One bottle of beer on the wall, One bottle of beer.
Take one down, pass it around, Ninety-eight bottles of beer on the wall.

No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, Ninety-nine bottles of beer on the wall.

Write a procedure which prints the complete lyrics starting with n bottles where $n \geq 0$. You do not need to print the reading of the numbers as in the example. Also lines like "99 bottles..." are OK. See also the next exercise.

Exercise: Write a procedure which converts any number n with $0 \leq n \leq 999$ to its English reading. That is, for 256, say, your procedure should return the string "two hundred fifty six."

Exercise: Write a procedure which converts any positive number into Roman numerals. For example, it should give the following results:

input	4	8	256	1199	2013
output	*IV*	*VIII*	*CCLVI*	*MCXCIX*	*MMXIII*

Exercise: Make the matrix multiplication example notify the user of wrong input.

Even more on loops

Sometimes we would like to end a loop prematurely. For this, Maple has the "break" statement.

```
for i from 2 to 10 do
   if issqr(i) then      # "issqr" checks if a number is a square
      break
   end if:
   print(cat("i is ", i, "."))
end do
```

"i is 2."

"i is 3." **(7.1)**

With "break" only the inner-most loop is terminated.

```
for i from 1 to 4 do
   for j from i to 4 do
      if j = 3 then
         break
```

```
    end if:
    print(cat("i is ", i, " and j is ", j, "."))
  end do
end do
```

```
"i is 1 and j is 1."
"i is 1 and j is 2."
"i is 2 and j is 2."
"i is 4 and j is 4."
```

(7.2)

There is also the "next" statement which does not end a loop completely but just skips one iteration. The example only prints the non-prime numbers between 1 and 10.

```
for i from 1 to 10 do
  if isprime(i) then
    next
  end if:
  print(cat("i is ", i, "."))
end do
```

```
"i is 1."
"i is 4."
"i is 6."
"i is 8."
"i is 9."
"i is 10."
```

(7.3)

We could write the above computation equivalently with an "if-then-else" statement. We use "NULL", the empy sequence, to mimick an "empty operation".

```
for i from 1 to 10 do
  if isprime(i) then
    NULL
  else
    print(cat("i is ", i, "."))
  end if
end do
```

```
"i is 1."
"i is 4."
"i is 6."
"i is 8."
"i is 9."
"i is 10."
```

(7.4)

Of course, this particular example would better be implemented as

```
for i from 1 to 10 do
    if not isprime(i) then
        print(cat("i is ", i, "."))
    end if
end do
```

```
"i is 1."
"i is 4."
"i is 6."
"i is 8."
"i is 9."
"i is 10."
```

(7.5)

Just as "break", "next" always refers to the innermost loop:

```
for i from 1 to 3 do
    for j from 1 to 3 do
        if j = 2 then
            next
        end if:
        print(cat("i is ", i, " and j is ", j, "."))
    end do
end do
```

```
"i is 1 and j is 1."
"i is 1 and j is 3."
"i is 2 and j is 1."
"i is 2 and j is 3."
"i is 3 and j is 1."
"i is 3 and j is 3."
```

(7.6)

Of course, inside a procedure, also the "return" statement terminates a loop.

As an example of "break", we print all the divisors of a number.

```
printdivisors := proc(number)
local rest, i :
   rest := number :

   while rest > 1 do
      for i from 2 to rest do
         if irem(rest, i) = 0 then
            rest := iquo(rest, i) :
            print(cat("Next divisor is ", i, ", rest is ", rest, ".")) :
            break
         end if
      end do
   end do
```

```
end proc:
```

We demonstrate that this works with

```
printdivisors(720):
```

```
"Next divisor is 2, rest is 360."
"Next divisor is 2, rest is 180."
"Next divisor is 2, rest is 90."
"Next divisor is 2, rest is 45."
"Next divisor is 3, rest is 15."
"Next divisor is 3, rest is 5."
"Next divisor is 5, rest is 1."                                    (7.7)
```

And indeed, we have

```
720 = 2·2·2·2·3·3·5
```

```
720 = 720                                                          (7.8)
```

As an example for "next", we implement the famous "Sieve of Eratosthenes". This method from ancient Greek allows to efficiently find all prime numbers smaller than a given upper bound. The idea is to write out all numbers up to that bound starting with 2. As long as there are still numbers left, we print the left-most number and erase all its multiples. Each number printed in that way must be a prime number because it has not been a multiple of any number smaller than itself. We could implement this sieve with lists which would allow us to really remove the corresponding multiples. For now, however, we will be using a table which for each number tells us whether the number has already been recognised as a multiple or not. In this simple example, we (ab-)use a vector to represent that table. A better suited data structure would be an array.

```
eratosthenes := proc(upperBound)
local isprime, i, k:
  isprime := Vector(upperBound, fill = true):

  for i from 2 to upperBound do
    # Check whether i is a prime
    if not isprime[i] then
      next
    end if:

    print(cat("The number ", i, " is prime.")):

    # Remove multiples
    for k from 2*i by i to upperBound do
      isprime[k] := false
    end do:
  end do:
```

```
    return NULL :
end proc:
```

As usual, we test the procedure:

eratosthenes(100):

"The number 2 is prime."
"The number 3 is prime."
"The number 5 is prime."
"The number 7 is prime."
"The number 11 is prime."
"The number 13 is prime."
"The number 17 is prime."
"The number 19 is prime."
"The number 23 is prime."
"The number 29 is prime."
"The number 31 is prime."
"The number 37 is prime."
"The number 41 is prime."
"The number 43 is prime."
"The number 47 is prime."
"The number 53 is prime."
"The number 59 is prime."
"The number 61 is prime."
"The number 67 is prime."
"The number 71 is prime."
"The number 73 is prime."
"The number 79 is prime."
"The number 83 is prime."
"The number 89 is prime."
"The number 97 is prime." **(7.9)**

Exercise: Print all numbers less than 100 which are not squares. (Hint: Use the "issqr" function.)

Exercise: A *palindrome* is a number (or a word) which is the same read from left to right or from right to left. Examples are the numbers 191, 12321 or the word "level". Write a procedure which tests whether a given (positive) integer is a palindrome and prints a corresponding message to the

user. Try to not compare more digits than strictly necessary. (Hint: You should decompose the number into its digits first, you can use a vector to store them.)

Exercise: Print all numbers in a given range which are divisible by 4 but not by 100 or which are divisible by 400. Do you know what kind of numbers you are printing?