Programming in Maple: The Basics.

Michael Monagan
Institut für Wissenschaftliches Rechnen
ETH-Zentrum, CH-8092 Zürich, Switzerland
monagan@inf.ethz.ch

Abstract

This is a tutorial on programming in Maple. The aim is to show how you can write simple programs in Maple for doing numerical calculations, linear algebra, and programs for simplifying or transforming symbolic expressions or mathematical formulae. It is assumed that the reader is familiar with using Maple interactively as a calculator.

Contents

1 Introduction

How does one learn to program in a computer language? Most people learn by studying examples and experimentation. That's why this document contains a large number of examples and exercises for you to try out at the end. When you are not sure of something, try it out, and experiment with your own examples.

A few words to those who are familiar with other programming languages.

- Maple is a procedural programming language. It also includes a number of functional programming constructs. If you have written programs in Basic, Pascal, Algol, C, Lisp, or Fortran, you should be able to write numerical programs in Maple very quickly.
- Maple is not strongly typed like C and Pascal. No declarations are required. Maple is more like Basic and Lisp in this aspect. However types exist. Type checking is done at run time and must be explicitly programmed.
- Maple is interactive and the programming language is interpreted. Maple is not suitable for running numerically intensive programs because of the interpreter overhead. Though it is suitable for high-precision numerical calculations and as a tool for generating numerical codes.

This document is based on Maple version V. Maple is actively being further developed. New versions come out every one or two years which contain not only changes to the Mathematical capabilities of Maple, but also changes to the programming language and user interface. For this reason I have tried to steer you away from constructs in the language which are being, or are likely to be removed, and have mentioned some things that will be in the language in future versions of Maple.

The *Maple V Language Reference Manual* is the main reference for programming in Maple. It is published by Springer-Verlag. Other useful sources of information are the *First Leaves: Tutorial Introduction to Maple*, also published by Springer-Verlag, and the extensive on-line documentation which is accessible with the `?` command in Maple. Part of the reason for writing this document is some of the important things I have stated here are not mentioned elsewhere, or they are buried in the documentation thus essentially inaccessible.

1.1 Evaluation

Before I begin, I want to point out the most important difference between Maple and traditional programming languages. If an identifier has not been assigned a value, then it stands for itself. I.e. it is a symbol. Symbols are used to represent unknowns in equations, variables in polynomials, summation indices, etc. Consider the Maple assignment statement

```
> p := x^2+4*x+4;

                                  2
                            p := x  + 4 x + 4
```

Here the identifier p has been assigned the formula $x^2 + 4x + 4$. The identifier x has not been assigned a value, it is just a symbol, an unknown. The identifier p has been assigned a value. It is now like a programming variable, and its value can be used in subsequent calculations just like a normal programming variable. What is the value of p ?

```
> p;
                                2
                               x  + 4 x + 4
```

It is the formula $x^2 + 4x + 4$. What is the value of x?

```
> x;
                                    x
```

It is the symbol x. Because a variable can be assigned a value which contains symbols, the issue of *evaluation* arises. Consider

```
> x := 3;
                                  x := 3

> p;
                                    25
```

Here we assigned x the value 3 and asked Maple to print out the value of p. What should Maple print out? Should it print out the polynomial $x^2 + 4x + 4$ or should it evaluate the polynomial, i.e. compute $3^2 + 4 \times 3 + 4$ and return the result 25? We see that Maple does the latter. This issue of evaluation will come up from time to time as it may affect the efficiency and semantics of a program. This difference between Maple and traditional programming languages, where identifiers can be used for both programming variables and mathematical unknowns is nice. But be careful not to mix the two. Many problems that users encounter have to do with using identifiers for both symbols and programming variables. For example, what happens if we now try to compute $\int p dx$?

```
> int(p,x);
Error, (in int) invalid arguments
```

An error has occurred in the integration function `int`. Here we are thinking of x as a symbol, the integration variable. But x has previously been assigned to the integer 3. Maple evaluates the arguments to the `int` function and tries to integrate 25 with respect to 3. It doesn't make sense to integrate wrt to 3! How does one make x back into a symbol? In Maple one *unassigns* the variable x by doing

```
> x := 'x';
                                    x := x

> int(p,x);
                                  3      2
                             1/3 x  + 2 x  + 4 x
```

1.2 Expression: Sums, Products, Powers, Functions

In Maple, mathematical formulae, e.g. things like $\sin(x+\pi/2)$, and $x^3y^2-2/3$ are called *expressions*. All formulae are made up from symbols, numbers, arithmetic operators and functions. Symbols are things like `sin, x, y, Pi` etc. Numbers include $12, \frac{2}{3}, 2.1$ etc. The arithmetic operators are `+` (addition), `-` (subtraction) `*` (multiplication) `/` (division) and `^` (exponentiation). Functions are like `sin(x)`, `f(x,y)`, `min(x1,x2,x3,x4)`. For example, the formula $p = x^2y + 3x^3z + 2$, which is a polynomial, is input as follows in Maple

```
> p := x^2*y + 3*x^3*z + 2;
                                  2        3
                            p := x  y + 3 x  z + 2
```

and the formula $\sin(x + \pi/2)e^{-x}$ is input as

```
> sin(x+Pi/2)*exp(-x)
                                cos(x) exp(- x)
```

Notice that Maple simplified $sin(x + \pi/2)$ to $cos(x)$ for us. Formulae in Maple are represented as *expression trees* or *DAGs* (Directed Acyclic Graphs) in computer jargon. When we program Maple functions to manipulate formulae, we are basically manipulating expression trees. Maple provides three basic routines for examining these expression trees in `type, op` and `nops` . The `type` function

`type(` f, t `)`

returns the value `true` if the expression f is of type t. Basic types include `string, integer, float` `` `+` ``, `` `*` ``, `` `^` ``, `function` . The `whattype` function is also useful for printing out the type of an expression. For example, our polynomial p is a sum of 3 terms. Thus

```
> type( p, integer );
                                     false

> whattype(p);
                                       +

> type( p, `+` );
                                     true
```

Note the use of back quotes here, `. Back quotes are used for strings in Maple which contain funny characters like `/,.` etc. Back quotes are not the same as forward quotes (the apostrophe) ' or double quotes ".
Other numerical types are `rational`, `float` and `numeric`. The type `rational` means the rational numbers, i.e. integers and fractions of integers. The type `float` means floating point numbers, i.e. numbers with a decimal point in them. The type `numeric` means any of these kinds of numbers, i.e. numbers of type `integer`, `rational`, or `float`. Maple users will have noticed that Maple distinguishes between exact numbers and approximate numbers. The presence of a decimal point is significant! Consider

```
> 2/3;
                                    2/3

# That is a rational number, the following is a floating point number
> 2 / 3.0;
                                .6666666667
```

> ATTENTION Engineers: many of you are used to using and writing Fortran or C software. In Fortran and C you always use decimal numbers. I.e. you have *learned* to always write 0.5 for 1/2. In Maple, if you are typing in a formula you probably mean 1/2. If you are typing in some data you probably mean 0.5 . 1/2 is not the same as 0.5 in Maple.

Our example p is a sum of terms. How many terms does it have? The `nops` function returns the **n**umber of **op**erands of an expression. For a sum, this means the number of terms in the sum. For a product, it means the number of terms in the product. Hence

```
> nops(p);
                                     3
```

The `op` function is used to extract one of the **op**erands of an expression. It has the syntax

$$\texttt{op(}\ i,\ f\ \texttt{)}$$

meaning extract the i'th operand of the expression f. In our example, this means extract the i'th term of the sum f.

```
> op(1,p);
                                     2
                                    x  y

> op(2,p);
                                      3
                                   3 x  z

> op(3,p);
                                     2
```

The op function can also be used in the following way

$$\texttt{op(}\ i..j,\ f\ \texttt{)}$$

This returns a sequence of operands of f from i to j. For example

```
> op(1..3,p);
                              2       3
                             x  y, 3 x  z, 2
```

A useful abbreviation is `op(f)` which is equivalent to `op(1..nops(f),f)` which means create a sequence of all the operands of f. What about the second term of the sum p. It is a product of 3 factors

```
> type(op(2,p),`*`);          # It is a product

                                     true

> nops(op(2,p));              # It has 3 factors

                                      3

> op(1,op(2,p));              # Here is the first factor

                                      3

> op(op(2,p));                # Here is a sequence of all 3 factors

                                       3
                                   3, x , z
```

Indexed names and Functions

Maple has two kinds of variables or *names* as Maple calls them. There are strings like `x`, `sin`, `Pi`, which are of type `string` and indexed names or subscripted variables like $A_1, A_{i,j}, A_{i_j}$ which are of type `indexed`. These examples are input `A[1]`, `A[i,j]`, `A[i][j]` in Maple. Most functions in Maple accept both kinds of variables. The Maple type `name` means either a string or a subscript. Example

```
> type(a,string);
                                     true

> type(a,name);
                                     true

> whattype(A[1]);
                                    indexed

> type(A[1],indexed);
```

```
true
```

```
> type(A[1],name);
```

```
true
```

If f in indexed name, then the nops function returns the number of indices, and the op function returns the i'th index. Also $op(0, f)$ returns the name of the index. Example

```
> nops(A[i,j]);
```

```
2
```

```
> op(1,A[i,j]);
```

```
i
```

```
> op(0,A[i,j]);
```

```
A
```

```
> nops(A[i][j]);
```

```
1
```

```
> op(1,A[i][j]);
```

```
j
```

```
> op(0,A[i][j]);
```

```
A[i]
```

Functions work very similarly to indexed names. The syntax for a function call is

$$f(x_1, x_2, \ldots,)$$

where f is the name of the function, and $x_1, x_2, \ldots$ are the arguments. The nops function returns the number of arguments, and the op function returns the i'th argument. Also $\mathrm{op}(0, f)$ returns the name of the function. Example

```
> nops(f(x,y,z));
```

```
3
```

```
> op(1..3,f(x,y,z));
```

```
x, y, z
```

```
> op(0,f(x,y,z));
```

```
f
```

We can now create and pick apart any Maple formula. Exercise 6 asks you to write a Maple program that picks apart a Maple formula. We conclude here with some examples showing how this can be done interactively.

```
> f := sin(x[1])^2*(1-cos(Pi*x[2]));

                                       2
                             sin(x[1])  (1 - cos(Pi x[2]))

> whattype(f);
                                            *

> op(1,f);
                                                    2
                                          sin(x[1])

> op(1,op(1,f));
                                          sin(x[1])

> op(1,op(1,op(1,f)));
                                             x[1]

> type(op(1,op(1,op(1,f))),indexed);
                                             true
```

1.3 Statements: Assignment, Conditional, Loops

The Maple syntax for the assignment, if, for and while statements is taken from Algol 60. The assignment statement looks like

$$name := expr$$

where *expr* is any expression. We want to mention here an evaluation problem that you will sooner or later run into. It arises because Maple can have unknowns as well as numbers. Consider the assignment to p earlier.

```
> p := x^2+4*x+4;

                                          2
                                    p := x  + 4 x + 4
```

Here we assigned the name p to a formula in the symbol x. What happens now if we use the same name p on both sides of the assignment statement?

```
> p := p+x;
                                          2
                                    p := x  + 5 x + 4
```

Nothing spectacular. But let us see how the evaluation rules operate here. The name p is assigned the result of evaluating and also simplifying the right hand side $p+x$. In evaluating $p+x$, the value of p is the polynomial x^4+4x+4 and the value of x is just x. The result of evaluation is therefore the expression $(x^4+4x+4)+x$. Next this is simplified to the final result x^2+5x+4. Now comes the big question. What if p here did not already have a value? I.e. consider the following

```
> q := q+x;

                    Warning: Recursive definition of name

                                 q := q + x
```

Well, you may say, obviously the user forgot to assign q a value first. Indeed that is probably the case, but let us continue and see what happens to Maple. Maple certainly allows q to not have a value just like x in the previous example.

The above statement resulted in a warning from Maple. The name q is now assigned a formula which involves q. What happens now if we try to evaluate q? We would have to evaluate $q+x$. But to evaluate $q+x$ we have to evaluate q again. Therein lies an infinite evaluation loop. On my system, if I try to evaluate q Maple dies with a rather spectacular crash

```
> q := q+x^2;
Segmentation fault
```

> ATTENTION: a recursive definition of a name like q in the above example will when evaluated typically result in crash as Maple runs out of Stack space. On some systems Maple will die. On other systems, Maple is able to stop just before it dies and report that there is a problem. If Maple ever crashes on you, there is a good chance that this is the reason. Maple will also die of course if there is a true infinite loop in an algorithm.

To recover from such a warning, simply unassign the variable q, i.e. do `q := 'q';`. Note that Maple does not detect all recursive assignments because this would be too expensive to do in general.

In a conventional programming language, this problem of a recursive definition cannot arise because all variables must have values. If they haven't been assigned a value, this is really an error. Depending on the language, they may get a default value, or they get whatever junk happens to be present in memory at the time, or the language may detect this and issue an error.

The conditional statement in Maple has the following syntax.

if *expr* **then** *statseq*
    [**elif** *expr* **then** *statseq*]*
    [**else** *statseq*]
fi

where *statseq* is a sequence of statements separated by semi-colons, [] denotes an optional part, and * denotes a part which can be repeated zero or more times. A typical if statement would be

```
if x < 0 then -1 elif x = 0 then 0 else 1 fi
```

The for statement has two variations. The first one is

[**for** *name*] [**from** *expr*] [**by** *expr*] [**to** *expr*] [**while** *expr*]
do *statseq* **od**

Thus each of the `for, from, by, to, while` clauses can be omitted. If omitted, the default values are a dummy variable, 1, 1, infinity and true respectively. A typical example of a for loop is

```
for i to 10 do print(i^2) od;
```

If one omits the **for**, **from**, **by** and **to** clauses we have a so called while loop.

```
i := 10^10+1;
while not isprime(i) do i := i + 2 od;
print(i);
```

Combining the for and while loops is sometimes nice, e.g. this example of searching for the first prime $> 10^{10}$ could be done as

```
for i from 10^10+1 by 2 while not isprime(i) do od;
print(i);
```

The second variation on the for loop is the so called for-in loop. It is really just a short hand for this loop which occurs very often

```
for i to nops(s) do f(op(i,s)) od;
```

where s may be a sum, but in general, any Maple expression or data structure such as a list or set – see next section. One can code this as a for-in loop as follows

```
for i in s do f(i) od;
```

The syntax for the for-in loop is

[**for** *name*] [**in** *expr*] [**while** *expr*]
do *statseq* **od**

2 Data Structures

More complicated programs involve manipulating and storing data. How we represent out data affects the algorithms that we write, and how fast our programs will run. Maple does have a good set of data structures. The ones we will look at here are sequences, lists (or vectors), sets, tables (or hash tables), and arrays. Maple does not directly have records or linked lists. We'll say how these can be implemented in Maple at the end of this section.

2.1 Sequences

A sequence is a just that, a sequence of expressions separated by commas. You can input a sequence directly

```
> s := 1,4,9,16,25;
                    s := 1, 4, 9, 16, 25

> t := sin,cos,tan;
                    t := sin, cos, tan
```

Sequences of sequences simplify into one big sequence, that is, sequences are *associative*. For example

```
> s := 1,(4,9,16),25;
                    s := 1, 4, 9, 16, 25

> s,s;
              1, 4, 9, 16, 25, 1, 4, 9, 16, 25
```

Note, the special symbol NULL is the empty sequence. Sequences are used for many purposes. The next section shows how lists and sets are constructed from sequences. Here we note that function calls are really constructed from sequences. For example the `min` and `max` functions in Maple take an arbitrary number of values as arguments, i.e. a sequence of arguments. For example

```
> max(s);

                          25

> min(s,0,s);

                          0
```

> ATTENTION: the `op` and `nops` functions cannot be used for sequences. This is because the sequence itself becomes the arguments to a function. Thus the calls `op(s);` and `nops(s);` are equivalent to `op(1,4,9,16,25);` and `nops(i,1,4,9,16,25)` respectively, and this results in an error. Put a sequence in a list first if you want to use `op` or `nops`.

The `seq` function is extremely useful for creating sequences. It comes in two flavours, corresponding to the two kinds of for loops. The first one is

$$\texttt{seq}(\ f(i),\ i = m..n\ )$$

The way seq works is just as if you had programmed the following loop.

```
s := NULL;
for i from m by 1 to n do s := s, f(i) od;
```

For example

```
> seq( i^2, i=1..5 );
                                 1, 4, 9, 16, 25

> s := NULL; for i from 1 to 5 do s := s, i^2 od;

                                      s :=

                                      s := 1

                                    s := 1, 4

                                   s := 1, 4, 9

                                 s := 1, 4, 9, 16

                               s := 1, 4, 9, 16, 25
```

Notice that `seq` is more efficient than the for loop because it does not create the intermediate sequences. The other flavour of `seq` is

$$\texttt{seq(}\ f(i),\ i = a\ \texttt{);}$$

This is equivalent to

```
seq( f(op(i,a)), i=1..nops(a) )
```

Here are a couple of interesting examples. The `coeff` function computes the coefficient of the term of degree i of a polynomial in x. The `D` function in Maple is the derivative operator.

```
> a := 3*x^3+y*x-11;
                                         3
                               a := 3 x  + y x - 11

> seq( coeff(a,x,i), i=0..degree(a,x) );

                                  -11, y, 0, 3

> seq( D(f), f=[sin,cos,tan,exp,ln] );

                                             2
                        cos, - sin, 1 + tan , exp, a -> 1/a
```

2.2 Lists and Sets

Lists, sets and functions are built up from sequences. A list is a data structure for collecting objects together. Square brackets are used to create lists.

```
> l := [x,1,1-z,x];
                        l := [x, 1, 1 - z, x]

> whattype(l);

                               list
```

The empty list is denoted by []. Sets can also be used to collect objects together. The difference is that duplicates are removed. Squiggely brackets are used for sets.

```
> s := {x,1,1-z,x};
                        s := {1, 1 - z, x}

> whattype(s);
                               set
```

The empty set is denoted by {}. The `nops` function returns the number of elements of a list or set and the `op` function extracts the i'th element. You can also use the subscript notation to access the i'th element of a sequence, list or set. Example

```
> op(1,s);
                                1

> s[1];
                                1

> op(1..3,s);
                           1, 1 - z, x

> s[1..3];

                           1, 1 - z, x
```

Here is a loop that prints true if a list or set contains the element x, false otherwise.

```
for i to nops(s) while s[i] <> x do od;
if i > nops(s) then print(false) else print(true) fi;
```

The built in `member` function does this for you. `member`(x, s) returns true if the element x is in the list or set s. You can append a new element to a list l by doing

```
l := [op(l),x];
```

You can remove the i'th element of a list by doing

```
l := [ l[1..i-1], l[i+1..nops(l)] ];
```

Or, better, use the `subsop` function

```
l := subsop(i=NULL,l);
```

where note the `subsop` function works for any type of expression. For sets, the set operators are `union`, `intersect` and `minus`, e.g.

```
> t := {u,x,z};
                              t := {x, z, u}

> s union t;
                              {x, z, y, u}
```

The Maple user will probably have noticed that Maple orders the elements of sets in a strange order which seems to be unpredictable. The algorithms that Maple uses to remove duplicates, and to do set union, intersection and difference all work by first sorting the entries of the input sets. Maple sorts by *machine address*, i.e. in the order that the entries occur in computer memory. Since this depends on where they are put in memory, this is why the order is odd and unpredictable. Maple does this simply because it is very fast.

2.3 Tables

Tables or hash tables are extremely useful for writing efficient programs. A table is a one-to-many relation between two discrete sets of data. For example, here is a table of colour translations for English to French and German.

```
> COLOUR[red] := rouge,rot;

                         COLOURS[red] := rouge, rot

> COLOUR[blue] := bleu,blau;

                        COLOURS[blue] := bleu, blau

> COLOUR[yellow] := jaune,gelb;

                       COLOURS[yellow] := jaune, gelb
```

The domain of the `COLOUR` table is the name of colour in English. The range of the table is a sequence of the name of the colour in French and German. In general, the domain and range values can be sequences of zero or more values. The domain values in the table are called the *keys* or *indices*. The Maple `indices` function returns a sequence of the indices. The range values in the table are called the *values* or *entries*. The Maple `entries` function returns a sequence of the entries. For example

```
> indices(COLOUR);

                         [red], [yellow], [blue]

> entries(COLOUR);
```

```
[rouge, rot], [jaune, gelb], [bleu, blau]
```

Note, the order that the `indices` and `entries` functions return the table indices and entries is not necessarily the same order as the order in which they were entered into the table. This is because Maple makes use of *hashing* to make table searching very fast and the initial order in which the entries were made is lost. However, there is a one to one correspondence between the output of the `indices` and the `entries` functions.

What can one do with a table? The basic capability is given a table *key* or *index* we can look up the corresponding entry very quickly. I.e. the operation

```
> COLOUR[red];

                              rouge, rot
```

returns the French and German names for the colour red quickly. How quickly? Roughly in constant time, no matter how large the table gets. Even if our table contains 1000 different colours, the time to search the table will be very fast. That is the basic use of a table. Information is entered into a table by assignment, and table look up is done by table subscript. What else can one do with a table? We can test if an entry is in the table using the assigned function. Example

```
> assigned(COLOUR[green]);
                              false

> assigned(COLOUR[blue]);
                              true
```

And we can remove entries from a table by unassignment.

```
> COLOUR[blue] := 'COLOUR[blue]';

                   COLOUR[blue] := COLOUR[blue]

> print(COLOUR);

                   table([
                       red = (rouge, rot)
                       yellow = (jaune, gelb)
                   ])
```

2.4 Arrays

A one-dimensional array is created using the `array` command

$$\mathrm{array}(\ m..n\ );$$

This creates an array with indices $m, m+1, \ldots, n$. Often m is 1. Entries can be inserted into the array by assignment as for tables, e.g.

```
v := array(1..n);
v[1] := a;
for i from 2 to n do v[i] := a*v[i-1] mod n od;
```

One-dimensional arrays are like tables with one index restricted to be in a fixed range. They are more efficient to access than tables and range checking on the indices is done. Here is an example of using a one-dimensional array to sort a sequence of numbers. E.g. suppose we are given an array $v =$ `array(1..n)` as above then this code sorts v into ascending order using the *bubblesort* algorithm.

```
for i to n-1 do
    for j from i+1 to n do
        if v[i] > v[j] then temp := v[i]; v[i] := v[j]; v[j] := temp fi
    od
od;
```

Another application of one-dimensional arrays is as an intermediate data structure. For example, suppose we represent a polynomial $a(x) = \sum_{i=0}^{m}$ as a list $[a_0, \ldots, a_m]$ of coefficients. Suppose we are given also a polynomial b of degree n. We can use an array c to compute the product $a \times b$ as follows

```
m := nops(a)-1; # degree of a
n := nops(b)-1; # degree of b
c := array(0..m+n); # allocate storage for the product
for i from 0 to m+n do c[i] := 0 od;
for i from 0 to m do
    for j from 0 to n do
        c[i+j] := c[i+j] + a[i+1]*b[j+1]
    od
od:
[seq(c[i],i=0..n+m)]; # put the product in a list
```

Two-dimensional arrays, and higher dimensional arrays work similarly. E.g. a two-dimensional array is created by the command

$$\text{array}(\ c..d,\ m..n\ );$$

For example, suppose we are given the vector v below of symmetric polynomials in 3 variables x_1, x_2, x_3.

```
v := array(1..4):
v[1] := 1:
v[2] := x[1] + x[2] + x[3]:
v[3] := x[1]*x[2] + x[1]*x[3] + x[2]*x[3]:
v[4] := x[1]*x[2]*x[3]:
```

Let us construct a two-dimensional array M where $M_{i,j}$ is the derivative of v_i wrt x_j.

```
> M := array(1..4,1..3):
> for i to 4 do for j to 3 do M[i,j] := diff(v[i],x[j]) od od:
> M;
                                   M

> eval(M);
                    [      0            0            0      ]
                    [                                       ]
                    [      1            1            1      ]
                    [                                       ]
                    [ x[2] + x[3]  x[1] + x[3]  x[1] + x[2] ]
                    [                                       ]
                    [  x[2] x[3]    x[1] x[3]    x[1] x[2]  ]
```

ATTENTION: Notice that the value of `M` is just the name of the array `M`. Evaluation rules for arrays and tables are special. The reason is so that you can have arrays with unassigned entries but we will not get into details here about that. For now, take note that whenever you want to print an array or table or return an array or table from a procedure use the `eval` function.

For further information on arrays see `?array` . Note also that one-dimensional arrays indexed from 1 are used to represent vectors and two-dimensional arrays indexed from 1 are used to represent matrices. See the section on Matrices and Vectors.

2.5 Records

Maple doesn't explicitly have a record data structure like Pascal's `record` or C's `struct`. By a record data structure we mean a data structure for keeping together a heterogeneous collection of objects, i.e. a set of objects not necessarily of the same type.

An example of a place where you would like a record data structure would be in choosing a data structure to represent a quaternion. A quaterion is a number of the form $a + bi + cj + dk$ where a, b, c, d are real numbers. To represent a quaterion, we need to store only the four quantities a, b, c and d. Another example would be a data structure to represent the factorization of a polynomial in $Q[x]$. The factorization of $a(x)$ looks like

$$a(x) \;=\; u \times f_1^{e_1} \times \ldots \times f_n^{e_n}$$

where each of the factors $f_i \in Q[x]$ is monic and irreducible. We need to store the factors f_i and the exponents e_i and the unit $u \in Q$.

There are several possibilities for representing records in Maple. The simplest, and most obvious one, is to use a list. I.e. we would represent the quaternion $a + bi + cj + dk$ as the list $[a, b, c, d]$, and the factorization of $a(x)$ as the list $[u, f]$ where f is a list of lists of the form $[f_i, e_i]$. We use subscripts to refer to a component of the data structure, e.g. the unit part of a factorization would be given by a[1]. One can use the *macro* facility to define an identifier to be equal to a constant if you prefer to use a symbol to reference a component as shown in the following example

```
> a := [-1/2,[[x+1,2],[x-1,1]]];

                     a := [-1/2, [[x + 1, 2], [x - 1, 1]]]

> macro(unit=1,factors=2,base=1,exponent=2);
> a[unit];
                                     -1/2

> a[factors][1][base];
                                     x + 1

> a[unit]*a[factors][1][base]^a[factors][1][exponent]*a[2][2][1]^a[2][2][2];

                                          2
                              - 1/2 (x + 1)  (x - 1)
```

A second possibility for representing records in Maple is to use a function call. I.e. we could represent $a + bi + cj + dk$ as QUATERNION(i,j,k,l). An advantage of this representation is that you we can tell Maple how to do various operations on functions. We will go into details later. Here we mention that you can define how to *pretty print* a function. The example below will show what we mean

```
> QUATERNION(2,3,0,1);

                              QUATERNION(2, 3, 0, 1)

> `print/QUATERNION` := proc(a,b,c,d)  a + b*'i' + c*'j' + d*'k' end:
> QUATERNION(2,3,0,1);

                                  2 + 3 i + k
```

Here we have defined a printing procedure or subroutine. This routine is called once for each different QUATERNION in a result from Maple prior to displaying the result. Note the use of quotes in the procedure, i.e. we want to see the identifiers i, j, and k in the output, not the value of the variables i, j, k which we might be using for something else.

A third possibility for representing a record is to think of it as a multivariate polynomial in the field names, and to store the values in the coefficients. This is quite useful when the fields are numerical and you wish to be able to do arithmetic on the fields. For example, we could represent a quaternion as a polynomial in the variables i, j, k as in the output representation above. I.e.

```
> z1 := 2 + 3*i + k;

                               z1 := 2 + 3 i + k

> z2 := 2 - 3*i + 2*j + 2*k;

                            z2 := 2 - 3 i + 2 j + 2 k
```

```
> coeff(z1,i); # the coefficient in i

                                       3

> z1 + z2;

                                  4 + 3 k + 2 j
```

Although this looks nice we don't recommend using the names i, j, or k because you will for sure use them for loop variables!

2.6 Linked Lists

Maple lists are not linked lists. Maple lists are really arrays of pointers to their entries. Maple lists differ from Maple arrays in that they are read only, i.e. you can't assign to a component of a Maple list. Linked lists are recursive data structures. The difference can be seen by studying a simple example. Consider how we might represent a polynomial $a(x) = \sum_{i=0}^{n} a_i x^i$ in Maple. One possibility would be to store the coefficients $a_0, \ldots a_n$ in a Maple list. E.g. we could represent the polynomial $p = x^4 + 3x^2 + 2x + 11$ as follows

```
[11, 2, 3, 0, 1]
```

The degree of the polynomial is the number of entries in the list minus one. Alternatively we could use a linked list

```
[ 1, [ 0, [ 3, [ 2, [ 11, NIL ]]]]]
```

We see that the linked list is a recursive data structure. It is either record of two values, traditionally called the *CAR* and *CDR* – terminology from the Lisp programming language, or the special value NIL signifying the empty linked list. The first field in the record contains a data value, in our case, a coefficient. The second field is a pointer to another linked list, which contains the rest of the data values. Note that in order to compute the degree of a polynomial represented by a linked list we have to compute the depth of the linked list. You could do this using the loop

```
for n from 0 while p <> NIL do p := p[2] od;
```

Now, why would we represent a polynomial using a linked list instead of a Maple list? The principle reason is that we can prepend a new entry onto a linked list in constant time instead of linear time. E.g., suppose we wanted to add the term $5x^5$ to our polynomial p. In the Maple list representation we must create a new list with 6 entries by doing

```
[op(p),5];
```

This requires at least 6 words of storage for the new Maple list. Don't be fooled. The `op` call here remember is short for `op(1..nops(p),p)` which creates a sequence of all the entries in the Maple list p. This takes constant time and uses no storage but now, we have sequence `(11, 2, 3, 0, 1), 5` which remember gets flattened into one big sequence `11, 2, 3, 0, 1, 5`. That is where 6 words of storage get allocated to store the new sequence. In general, to add $a_{n+1}x^{n+1}$ to a polynomial of degree n this takes $O(n)$ time and storage. But what about the linked list? Adding the term $5x^5$ is

```
[5,p];
```

This only needs a Maple list of length two, hence constant storage. The time will also be linear time if p is a global variable, but constant time if p is a local variable or parameter inside a procedure. This evaluation detail will be explained in the section on procedures. But for now assume that the running time and space will be $O(1)$ for linked lists.

> ATTENTION Lisp Programmers: You can't change the components of a Maple list. I.e. there can be no equivalent of `REPLACA` and `REPLACDR`. Maple lists, sets, sequences, functions are read only data structures. Only arrays and tables can be changed.

We have shown how one can implemented a linked list by essentially using a Maple list as a record of the `CAR` and `CDF` fields. A second possibility would be to use a function call instead of a Maple list. E.g.

```
LINKLIST( 1, LINKLIST( 0, LINKLIST( 3, LINKLIST( 2, LINKLIST( 11, NIL )))))
```

3 Maple Procedures

3.1 Parameters, Local Variables, RETURN, ERROR

A Maple procedure has the following syntax

proc (*nameseq*)
  [**local** *nameseq* ;]
  [**options** *nameseq* ;]
  statseq
end

where *nameseq* is a sequence of symbols separated by commas. Here is a simple procedure which given x, y computes $x^2 + y^2$.

```
proc(x,y) x^2 + y^2 end
```

This procedure has 2 parameters x and y. The value returned is $x^2 + y^2$. In general the value returned by a procedure is the last value computed unless there is an explicit return. An example of a procedure with an explicit return is the following `MEMBER` procedure. `MEMBER(x,L)` returns `true` if x is in the list L, false otherwise.

```
MEMBER := proc(x,a) local v;
    for v in L do if v = x then RETURN(true) fi;
    false
end;
```

The MEMBER procedure has a local variable v so that it does not interfere with the global user variable v. Variables that appear in a procedure which are neither parameters nor local variables are the global variables.

The ERROR function can be used to generate an error message from within a procedure. For example, the MEMBER routine should check that the argument really is a list and issue an appropriate error message otherwise.

```
MEMBER := proc(x,L) local v;
    if not type(L,list) then ERROR(`2nd argument must be a list`) fi;
    for v in L do if v = x then RETURN(true) fi;
    false
end;
```

3.2 Tracing Procedure Execution: `printlevel`

Here is a Maple procedure which computes the greatest common divisor of two non-negative integers using the Euclidean algorithm. For example, the greatest common divisor of the integers 21 and 15 is 3 because 3 is the largest integer that divides both 21 and 15. By the way, the Euclidean algorithm is one of the oldest known algorithms in Mathematics. It dates back to around 300 BC.

```
GCD := proc(a,b) local c,d,r;
    c := a;
    d := b;
    while d <> 0 do r := irem(c,d); c := d; d := r od;
    c
end;
```

The `irem` function note computes the **i**nteger **rem**ainder of two integers. How does the GCD routine really work? Can the results of each of the steps of executing the procedure GCD? The simplest tool for looking at the execution of a procedure is the `printlevel` facility. The `printlevel` variable is a global variable that is initially assigned 1. If you set it to a higher value, a trace of all assignments, and procedure entry and exits is printed. Lets see what happens when we compute `GCD(21,15);`

```
> printlevel := 100;
> GCD(21,15);
--> enter GCD, args = 21, 15
                                   c := 21

                                   d := 15
```

```
                              r := 6

                              c := 15

                              d := 6

                              r := 3

                              c := 6

                              d := 3

                              r := 0

                              c := 3

                              d := 0

                                3

<-- exit GCD = 3
                                3
```

We see that the input arguments to the GCD procedure are displayed along with the value returned. The execution of each assignment statement is also displayed.

An interesting point to note about our Maple GCD procedure is that because Maple uses arbitrary precision integer arithmetic this routine works for integers of any size, e.g. here is the greatest common divisor between 100! and 2^100

```
> GCD(100!,2^100);
                  158456325028528675187087900672
```

Our `GCD` procedure could also have been been written recursively in this way

```
GCD := proc(a,b)
    if b = 0 then a else GCD(b,irem(a,b)) fi
end;
```

The recursive version is simpler and easier to understand. Let us see a trace of the recursive version

```
> GCD(15,21);
--> enter GCD, args = 15, 21
--> enter GCD, args = 21, 15
--> enter GCD, args = 15, 6
--> enter GCD, args = 6, 3
--> enter GCD, args = 3, 0
                                3

<-- exit GCD = 3
                                3
```

```
<-- exit GCD = 3
                                       3

<-- exit GCD = 3
                                       3

<-- exit GCD = 3
                                       3

<-- exit GCD = 3
                                       3
```

3.3 Arrow Operators

For simple procedures, which compute only a formula, there is an alternative syntax called the arrow syntax. The mimics the syntax for functions often used in algebra.

symbol `->` [**local** *nameseq*;] *expr*

is used for functions of 1 parameter. In general, for 0 or more parameters, the parameters are put in parentheses i.e.

(*nameseq*) `->` [**local** *nameseq*;] *expr*

The example which computes $x^2 + y^2$ can be written more succinctly as

```
(x,y) -> x^2+y^2;
```

3.4 Scope Rules: Parameters, Locals, Globals

Maple does support nested procedures. For example, you can write

```
f1 := proc(x) local g; g := x -> x+1; x*g(x) end;
```

Procedure f_1 has a local variable g which is a procedure. This computes $x * (x + 1)$. However, nested parameters and local variables do not use nested scoping rules. E.g. the above procedure is not equivalent to this one

```
f2 := proc(x) local g; g := () -> x+1; x*g() end;
```

because the reference to x in the g procedure here does not refer to the parameter x in the f procedure. It refers to the global variable x. Consider these examples

```
> f1(a);

                               a (a + 1)

> f2(a);
```

```
                          a (x + 1)

> x := 7;

                           x := 7

> f2(a);

                            8 a
```

One similarly cannot refer to local variables in outer scopes.

> ATTENTION Computer Scientists: Although Maple supports nested procedures, which can be returned as function values, it does not support nested lexical scopes, so you cannot directly return closures. This is being looked at for a future version of Maple. E.g. the following does NOT define a shift operator `f -> x -> f(x+1)`

3.5 Evaluation Rules: Actual and Formal Parameters, Locals, Globals

Consider the function call `f(x1,x2,...,xn);` The execution of this function call proceeds as follows. The function name *f* is evaluated. Next the arguments *x1, x2, ..., xn* are evaluated from left to right. Then if *f* evaluated to a procedure, the procedure is executed on the evaluated arguments. There are a few exceptions to this for a 6 of the builtin functions in particular `eval` and `seq`.

Now, what about the evaluation of variables inside procedures? Consider the `procedure`

```
f := proc() local p; p := x^2+4*x+4; x := 5; p end
```

Here p is a local variable and there are no parameters. But what is the variable x? Because it is neither a parameter nor a local variable Maple defines it to be a *global* variable. Or if you prefer, the *user* variable. What are the evaluation rules for parameters, locals, and global variables? In the introduction we considered the problem

```
> p := x^2+4*x+4;
                              2
                        p := x  + 4 x + 4

> x := 3;
                              x := 3

> p;
                                25
```

Here p and x are global variables. Global variables are evaluated fully, i.e. recursively, hence the result is 25. What if p is a local variable as in our procedure `f` above? I.e. what is the result of executing

```
f();
```

And what if p is a parameter as in the following procedure

```
> x := 'x'; # Make x a symbol first

                              x := x

> g := proc(p) x := 5; p end:
> g(x^2+4*x+4);

                              2
                             x  + 4 x + 4
```

For reasons of efficiency and desirability, the Maple designers have decided that local variables and parameters evaluate one level, i.e. the value of p in the above two examples is the polynomial $x^2 + 4x + 4$ not the value 49. Full evaluation only occurs for global variables. The `eval` function can be used to get full evaluation for local variables and parameters, and one level evaluation of global variables should you ever need it, e.g.

```
> x := 'x'; # Make x a symbol first

                              x := x

> g := proc(p) x := 5; eval(p) end:
> g(x^2+4*x+4);

                                49
```

3.6 Recurrence Equations and Option Remember

The Fibonacci numbers F_n are defined by the linear recurrence $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$. This can be coded directly by

```
F := proc(n)
    if n = 0 then 0 elif n = 1 then 1 else F(n-1)+F(n-2) fi
end;
```

Here are the first few Fibonacci numbers

```
> seq( F(i), i=0..10 );

                0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

However this is a not an efficient way to compute the Fibonacci numbers. In fact, you will never be able to compute `F(100)` using this procedure even on the fastest computer. If you count the number of calls to the F procedure, you will see that it is called repeatedly on the same arguments. It is clear that one should remember the previous two values when computing the next value. This could be done in a loop in the following way

```
F := proc(n) local fnm1,fnm2,f;
    if n = 0 then RETURN(0) fi;
    fnm2 := 0;
    fnm1 := 1;
    for i to n do f := fnm1 + fnm2; fnm2 := fnm1; fnm1 := f od;
    fnm1
end;
```

Another way to code this is to use *option remember*. This option is a used to store values as they are computed so that they can be used when they are needed. Consider

```
F := proc(n) option remember;
    if n = 0 then 0 elif n = 1 then 1 else F(n-1)+F(n-2) fi
end;
```

This program computes $F(100)$ quite quickly. What happens when F is executed is the following. Each Maple procedure has an associated *remember* table. The table index is the arguments and the table entry is the function value. When F is called with n, Maple first looks up F's *remember* table to see if $F(n)$ has already been computed. If it has, it returns the result from F's remember table. Otherwise, it executes the code for the procedure F, and automatically stores the pair $n, F(n)$ in F's remember table.

We also illustrate the possibility of explicitly saving values in a remember table by using the so called *functional assignment*. This is more flexible than the remember option because it allows one to save only selected values in the remember table.

```
F := proc(n) F(n) := F(n-1)+F(n-2) end;
F(0) := 0;
F(1) := 1;
```

3.7 Types and Map

The `type` function can be used to code a routine that does different things depending on the type of the input. For example, the `DIFF` routine below differentiates expressions which are polynomials in the given variable x.

```
DIFF := proc(a,x) local u,v;
    if not type(a,algebraic) then ERROR(`1st argument must a formula`) fi;
    if not type(x,name) then ERROR(`2nd argument must be a name`) fi;
    if type(a,numeric) then 0
```

```
    elif type(a,name) then if a = x then 1 else 0 fi
    elif type(a,`+`) then map( DIFF, a, x )
    elif type(a,`*`) then u := op(1,a); v := a/u; DIFF(u,x)*v + DIFF(v,x)*u
    elif type(a,anything^integer) then
        u := op(1,a); v := op(2,a); v*DIFF(u,x)*u^(v-1)
    else ERROR(`don't know how to differentiate`,a)
    fi
end;
```

The DIFF example shows the use of the map function, a very useful function. It has the following syntax

$$\text{map}(\ f, a, x_1, ..., x_n\ )$$

The meaning is to apply the function f to the operands of the expression a passing any additional arguments to f. The most common is where there are no additional arguments. Formally, this is equivalent to computing the sequence

```
seq( f( op(1,a), x1, ..., xn ), i=1..nops(a) );
```

and then creating from this sequence a value of the same type of f. Here are some examples to illustrate

```
> p := x^3+2*x+1;
                                       3
                                 p := x  + 2 x + 1

> map( F, p );
                                    3
                                 F(x ) + F(2 x) + F(1)

> map( x -> x^2, p );
                                     6      2
                                    x  + 4 x  + 1

> map( degree, p, x );
                                          4
```

The DIFF function also shows the use of a *structured* type. The types anything, name, `+` and `*` are simple types. The type anything^integer is a *structured* type. It means that the value must be a power, and the base can be anything but the exponent must be an integer. It is equivalent to writing

```
if type(a,`^`) and type(op(2,a),integer) then
```

Structured types allow you to replace long type tests with concise tests. Let us illustrate two other common cases. The type name=algebraic..algebraic is the type of the second argument for definite integration, and definite summation, as in

```
> int( exp(-2*t)*ln(t), t=0..infinity );

                             - 1/2 ln(2) - 1/2 gamma

> sum( 1/(i^2-1), i=2..infinity );

                                        3/4
```

Of course, the integration and summation commands also allow indefinite integration and summation, i.e.

```
> int( exp(-2*x)*ln(x), x );

                        - 1/2 exp(- 2 x) ln(x) + 1/2 Ei(- 2 x)

> sum( 1/(n^2-1), n );

                                              1
                                - 1/n - -----------
                                        2 n (n - 1)
```

So the type of the second argument can be either just a variable name or, the above type. We do this as follows `{ name, name=algebraic..algebraic }`. The use of set braces is for denoting alternatives.

Another common case is when you want to test for a set or list of names or equations. E.g. the `solve` command allows one to solve a set of equations for a set of unknowns. A set of zero or more equations can be tested with the type: `set(equation)`, e.g. as in

```
> type( {x+y=2, x-y=3}, set(equation) );

                                        true
```

But actually, the `solve` command allows also a set of algebraic formulae which are implicitly equated to zero, i.e.

```
> solve( {x+y-2, x-y-3}, {x,y} );

                                {y = -1/2, x = 5/2}
```

Hence the type should be `{set(algebraic),set(equation)}`. Notice that this is not the same as `set({algebraic,equation})`. Why?

Further information about the various different types can be obtained from the on-line help system under ?type .

3.8 Variable Number of Arguments: `args` and `nargs`

It is possible in Maple for a function to take a variable number of parameters. An example of such a function is the `max` function in Maple. Here is an initial attempt to code up this function

```
MAX := proc(x1) local maximum,i;
    maximum := x1;
    for i from 2 to nargs do
```

```
            if args[i] > maximum then maximum := args[i] fi
        od;
        maximum
    end;
```

The special variable *nargs* is the number of arguments, and the variable *args* is a sequence of the arguments, hence `args[i]` is the i'th argument.

3.9 Returning Unevaluated

The `MAX` procedure that we have just written works for numerical arguments only. If you try the Maple function `max` you will see that it also works for symbolic arguments. Consider

```
> MAX(1,2,x);
Error, (in MAX) cannot evaluate boolean
> max(1,2,x);

                                    max(2, x)
```

Maple cannot execute the procedure `MAX` because it cannot compute whether `args[i] < maximum` for a non-numeric value. We want `MAX` of some numbers to compute the answer, but otherwise, we want `MAX(x,y)` to stay as `MAX(x,y)` so we can compute with `MAX(x,y)` symbolically just like `sin(x)` stays as `sin(x)`.

Also, our `MAX` procedure that we wrote only works for numbers of type `numeric`. It would be nice if `MAX` knew that $\pi > \sqrt{2}$ for instance. A more powerful comparison of two real values is made by the `signum` function where `signum(x)` returns -1 if it can show that the $x < 0$, $+1$ if $x \geq 0$, otherwise it returns *unevaluated*, i.e. it returns `signum(x)`. Example

```
> signum(sqrt(2)-1);
                                       1

> signum(sqrt(2)-Pi);
                                      -1

> signum(a-b);
                                 signum(a - b)
```

Let us employ the signum function to make our `MAX` function smarter and also let our `MAX` function handle symbolic arguments.

```
    MAX := proc() local a,i,j,n,s;
        n := nargs;
        a := array(1..n);
        for i to n do a[i] := args[i] od;
        # Compare a[i] with a[j] for 1 <= i < j <= n
        i := 1;
        while i < n do
```

```
        j := i+1;
        while j <= n do
            s := signum(a[i]-a[j]);
            if s = 1 then # a[i] >= a[j]
                a[j] := a[n]; n := n-1;
            elif s = -1 then # a[i] < a[j]
                a[i] := a[j]; a[j] := a[n]; j := n; n := n-1; i := i-1;
            else # cannot determine the sign
                j := j+1
            fi
        od;
        i := i+1
    od;
    if n = 1 then RETURN(a[1]) fi;
    'MAX'( seq(a[i], i=1..n) );
end;
```

What is interesting about the above code is the last line. The quotes ' here mean that the `MAX` function is not reexecuted (otherwise it would go into an infinite loop). Rather the unevaluated function call MAX(...) is returned, indicating that the maximum could not be computed. However, some simplifications may have taken place. For example

```
> MAX( x, 1, sqrt(2), x+1 );
                                              1/2
                                 MAX(x + 1, 2   )
```

3.10 Simplifications and Transformation Rules

Often one wants to introduce simplifications which can be described algebraically or by transformation rules. For instance, given a function f we may know that f is both commutative and associative. max is in fact such a function, i.e. it is true that $\max(a,b) = \max(b,a)$ and $\max(a, \max(b,c)) = \max(\max(a,b),c) = \max(a,b,c)$. How can we implement these properties in Maple? What we want is a *canonical* way for writing expressions involving the max function. We can implement commutativity by sorting the arguments. For associativity we can denest any nested max calls. I.e. we would transform both $\max(\max(a,b),c)$ and $\max(a, \max(b,c)$ into $\max(a,b,c)$. Actually this also implements $\max(\max(a)) = \max(a)$, i.e. max is idempotent. Here is a `MAX` function to do this

```
MAX := proc() local a;
    a := [args];
    a := map( flatten, a, MAX ); # denest nested MAX calls
    'MAX'( op(sort(a)) );
end;
flatten := proc(x,f)
    if type(x,function) and op(0,x) = f then op(x) else x fi
end;
```

For example

```
> MAX(a,MAX(c,b),a);
```

```
                              MAX(a, a, b, c)
```

We see that we should also recognize the property that $\max(a,a) = a$. To do this, instead of putting the arguments in a list, we will put them in a set so that duplicates are removed. Also, since sets are sorted automatically, we can remove the call to sort. Hence we have

```
MAX := proc() local a;
    a := {args};
    a := map( flatten, a, MAX );
    'MAX'( op(a) );
end;
```

```
> MAX(a,MAX(c,b),a);
```

```
                              MAX(a, b, c)
```

The reader may be a little puzzled as to just what out `MAX` procedure is doing. We have seen earlier that if we assign a positive integer to the `printlevel` variable, we get a trace of all statements executed. However, often the output from this simple tracing facility is too much. We will also get the output from the flatten procedure. The `trace` function can be used to selectively trace given routines. Lets see then just how the MAX function works

```
> trace(MAX);
```

```
                                  MAX

> MAX(a,MAX(b,a),c);
--> enter MAX, args = b, a
                              a := {a, b}

                              a := {a, b}

                               MAX(a, b)

<-- exit MAX = MAX(a,b)
--> enter MAX, args = a, MAX(a,b), c
                          a := {a, c, MAX(a, b)}

                             a := {a, c, b}

                              MAX(a, c, b)

<-- exit MAX = MAX(a,c,b)
                              MAX(a, c, b)
```

3.11 Optional Arguments and Default Values

Many Maple routines accept optional arguments. And often this is to allow the user to avoid specifying parameters by having the program use default values. Examples are the functions `plot`, `factor`, `collect`, `series`. Let us consider the `degree` function. The `degree` function computes the degree of a univariate polynomial in a variable x, the total degree of a multivariate polynomial. For example

```
> p := x^3+2*x+1;

                                   3
                             p := x  + 2 x + 1

> degree(p);

                                     3

> q := 3*x^2*y+2*y^2-x*z+7;

                                2        2
                        q := 3 x  y + 2 y  - x z + 7

> degree(q);

                                     3
```

Sometimes you will want to compute the degree in one variable, say x. This can be done by specifying an optional second argument to the degree function, the variable. For example

```
> degree(p,x);

                                     3

> degree(q,x);

                                     2
```

How would we code the degree function. Lets assume that the input is a formula and if an optional second argument is given that it is a name or set of names for the variables. We would write

```
DEGREE := proc(a,x) local s,t;
    if nargs = 1 then # determine the variable(s) for the user
        s := indets(a);
        if not type(s,set(name)) then ERROR(`input not a polynomial`) fi;
        DEGREE(a,s)
    elif type(a,constant) then 0
    elif type(a,name) then
        if type(x,name) then if a = x then 1 else 0 fi
        elif type(x,set(name)) then if member(a,x) then 1 else 0 fi
        else ERROR(`2nd argument must be a name or set of names`)
```

```
            fi
        elif type(a,`+`) then max( seq( DEGREE(t,x), t=a ) )
        elif type(a,`*`) then
            s := 0;
            for t in a do s := s + DEGREE(t,x) od;
            s
        elif type(a,algebraic^integer) then DEGREE(op(1,a),x) * op(2,a)
        else ERROR(`cannot compute degree`)
        fi
    end;
```

The `indets` function used here returns a set of all the *indeterminates* (or variables) that appear in the input. We leave it to the reader to study each rule that is being used here, and the order in which the cases are done.

3.12 Returning Results Through Parameters

Many functions in Maple return more than one value. Of course, it is always possible to return more than one value in a sequence or list. However, it is also possible to return values through parameters like in other programming languages, and often, this is more convenient. For example, consider the `divide` function in Maple which does polynomial long division. The call `divide(a,b)` returns true if and only if the polynomial b divides the polynomial a with no remainder, e.g.

```
> divide(x^3-1,x-1);

                                        true
```

But usually, if b divides a, one wants to do something with the quotient q. In Maple, this can be done by giving the `divide` function a third parameter which is a name which will be assigned the quotient if b divides a, e.g.

```
> if divide(x^3-1,x-1,'q') then print(q) fi;

                                      2
                                     x  + x + 1
```

Notice the use of quotes here to pass the divide function the name q and not the value of q. Let us consider another example and study how to write a program which assigns to an optional parameter. Consider our `MEMBER` function which tested to see if a value x appears in a list L. Let us modify our function such that `MEMBER(x,L,'p')` still returns whether x appears in the list L, but also, assigns the name p the position of the first appearance of x in L.

```
    MEMBER := proc(x,L,p) local i;
        for i to nops(L) do
            if x = L[i] then
                if nargs = 3 then p := i fi;
                RETURN(true)
            fi
```

```
      od;
      false
   end;
```

Here is an example

```
> MEMBER(4,[1,3,5],'position');
```

false

```
> position;
```

position

```
> MEMBER(3,[1,3,5],'position');
```

true

```
> position;
```

2

We see that the effect of the assignment to p inside the `MEMBER` procedure is that the actual parameter `position` is assigned.

> ATTENTION Pascal and C programmers: This means that one can't use parameters like local variables inside a procedure.

4 Programming in Maple

4.1 Matrix and Vector Computation in Maple

A vector in Maple is represented by a one-dimensional array indexed from 1, and a matrix is represented by a two-dimensional array, with row and column indices starting from 1. Thus here is one way to create a 5 by 5 Hilbert matrix. Recall that a Hilbert matrix is a symmetric matrix where the (i,j)'th entry is $1/(i+j-1)$.

```
> H := array(1..5,1..5):
> for i to 5 do for j to 5 do H[i,j] := 1/(i+j-1) od od;
> H;
```

H

```
> eval(H);
                          [  1   1/2  1/3  1/4  1/5 ]
                          [                         ]
                          [ 1/2  1/3  1/4  1/5  1/6 ]
                          [                         ]
                          [ 1/3  1/4  1/5  1/6  1/7 ]
                          [                         ]
                          [ 1/4  1/5  1/6  1/7  1/8 ]
                          [                         ]
                          [ 1/5  1/6  1/7  1/8  1/9 ]
```

> ATTENTION: Notice that the value of `H` is just the name of the matrix `H`. Evaluation rules for arrays, hence matrices and vectors, are special. The reason is so that you can have matrices with unassigned entries. For now, take note that whenever you want to print a matrix or return a matrix from a procedure use the `eval` function.

The `linalg` package contains many functions for computing with vectors and matrices in Maple. This matrix could also have been created in the following way using the `matrix` command in the linear algebra package.

```
linalg[matrix](5,5,(i,j) -> 1/(i+j-1));
```

You can compute the determinant and inverse of the matrix H and many other matrix operations. See `?linalg` for a list of the operations.

Here we show a program for doing a so called row reduction on a matrix. using Gaussian elimination. The call `GaussianElimination(A,'r')` computes the reduced matrix, an upper triangular matrix. It also has an optional 2nd argument which if given is assigned the rank of the matrix.

```
GaussianElimination := proc(A,rank)
local m,n,i,j,B,r,c;

    if not type(A,'matrix(rational)') then
        ERROR(`expecting a matrix of rational numbers`,A) fi;

    m := linalg[rowdim](A);
    n := linalg[coldim](A);

    B := array(1..m,1..n);
    for i to m do for j to n do B[i,j] := A[i,j] od od;

    r := 1; # r and c are row and column indices
    for c to m while r <= n do

        for i from r to n while B[i,c] = 0 do od;  # search for a pivot
```

```
        if i <= n then
            if i <> r then # interchange row i with row r
                for j from c to m do
                    t := B[i,j]; B[i,j] := B[r,j]; B[r,j] := t
                od
            fi;
            for i from r+1 to n do
                if B[i,c] <> 0 then
                    t := B[i,c]/B[r,c];
                    for j from c+1 to m do B[i,j] := B[i,j]-t*B[r,j] od;
                    B[i,c] := 0
                fi
            od;
            r := r + 1                # go to next row
        fi
    od;                        # go to next column

    if nargs>1 then rank := r-1 fi;
    eval(B)

end:
```

4.2 Numerical Computation in Maple

Floating point numbers in Maple can be input either as decimal numbers e.g. 3.1 or directly using the `Float` function

$$\texttt{Float(m,e)} = m * 10^e$$

where the mantissa m is a Maple integer of any precision, and the exponent e is restricted to be a machine size integer, typically 31 bits. The `op` function can be used to extract the mantissa and exponent of a floating point number. Note: the floating point constant 0.0 is treated specially and is simplified to 0, i.e. the integer 0 automatically. Here is a uniform random number generator that generates uniform random numbers with exactly 6 decimal digits precision in the range $[0, 1)$.

```
> UniformInteger := rand(0..10^6-1):
> UniformFloat := proc() Float(UniformInteger(),-6) end:
> seq( UniformFloat(), i=1..6 );

            .669081, .693270, .073697, .143563, .718976, .830538
```

We have used the built-in function `rand` that returns a random number generator for the specified range.

Floating point arithmetic is done in decimal with rounding. The precision used is controlled by the global variable `Digits` which has 10 as its default value. It can be set to any value however. Arithmetic with floating point numbers is automatic. The `evalf` function is used to evaluate an exact symbolic constant to a floating point approximation. For example

```
> Digits := 25:
> sin(1.0);
                    .8414709848078965066525023

> sin(1);
                              sin(1)

> evalf(");
                    .8414709848078965066525023
```

Maple knows about the elementary functions and many special functions such as $J_v(x)$, $\Gamma(x)$, and $\zeta(x)$. In Maple these are `BesselJ(v,x)` `GAMMA(x)` and `Zeta(x)` respectively. They are all computed to high precision by summing various series.

The model of floating point arithmetic used is that the relative error of the result is less than $10^{1-Digits}$. This is a stronger model than that used by hardware implementations of floating point arithmetic and it requires that intermediate calculations to be done at higher precision to obtain such accurate results. Here is an example of summing a Taylor series. Suppose we want to compute the error function

$$erf(x) \;=\; \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

for small x. If you are unfamiliar with the error function, take a moment to plot it in Maple. We can make use of the Taylor series for $erf(x)$ about $x = 0$

$$\frac{\sqrt{\pi}}{2} erf(x) \;=\; \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{n!(2n+1)} \;=\; x - \frac{x^3}{3} + \frac{x^5}{10} - \frac{x^7}{42} + \frac{x^9}{216} + \ldots$$

for computing $erf(x)$ for small x, $-1 < x < 1$ (with no error checking) as follows

```
ErfSmall := proc(a) local x,x2,result,term,sumold,sumnew;
    x := evalf(a); # evaluate the input at Digits precision
    Digits := Digits + 2; # add some guard digits
    sumold := 0;
    term := x;
    sumnew := x;
    x2 := x^2;
    for n from 1 while sumold <> sumnew do
        sumold := sumnew;
        term := - term * x2 / n;
        sumnew := sumold + term / (2*n + 1);
```

```
    od;
    result := evalf( 2/sqrt(Pi)*sumnew );
    Digits := Digits-2;
    evalf(result) # round the result to Digits precision
end;
```

For large x, this routine will be inaccurate and inefficient. See the exercises for how to compute $erf(x)$ for large x .

For a second example, let us consider writing a Newton iteration to compute the $\sqrt{2}$ to 50 Digits. Recall that the Newton iteration for solving an equation $f(x) = 0$ is

$$x_{k+1} \;=\; x_k - \frac{f(x_k)}{f'(x_k)}$$

In our case, to compute $\sqrt{a}$ we want to solve $x^2 - a = 0$. So the iteration is

$$x_{k+1} \;=\; (x_k + \frac{a}{x_k})/2$$

where $a = 2$. The following routine will do the job.

```
SqrtNewton := proc(a) local xk, xkm1;
    Digits := 55; # add some guard digits
    xkm1 := 0;
    xk := evalf(a/2); # initial floating point approximation
    while abs(xk-xkm1) > abs(xk)*10^(-50) do
        xkm1 := xk;
        print(xk);
        xk := (xk + a/xk)/2;
    od;
    Digits := 50;
    evalf(xk); # round the result to 50 Digits
end;
```

We have put in a print statement to show the result of each iteration. Here is what happens

```
> SqrtNewton(2);

                              1.

     1.500000000000000000000000000000000000000000000000000

     1.416666666666666666666666666666666666666666666666667

     1.414215686274509803921568627450980392156862745098039216

     1.414213562374689910626295578890134910116559622115744045
```

```
1.414213562373095048801689623502530243614981925776197429

1.414213562373095048801688724209698078569671875377234002

1.414213562373095048801688724209698078569671875376948073

1.414213562373095048801688724209698078569671875376948073
```

It is known that a Newton iteration will converges quadratically provided $f'(x_k)$ is not close to zero and x_k is sufficiently close to a root of f, as illustrated in the above example. For very high precision though, i.e. Digits > 1000, one does not want to do every iteration at full precision as the early steps are not accurate to full precision. Why do all that work when you are only getting a few digits correct? Modify the Newton iteration used in the SqrtNewton procedure appropriately so that it doubles the number of Digits at each step. How much faster does this make the iteration run?

Because Maples floating point model is implemented in software, it is much slower than hardware floating point arithmetic. Maple also has a function called `evalhf` for `eval`uating in `h`ardware `f`loating point arithmetic. This function uses the builtin C library routines for floating point arithmetic. Consequently it is much faster than Maples software floats, but it is slower than hardware floats. See `?evalhf` for details.

4.3 Computing with Polynomials in Maple

Computing with polynomials and also rational functions is Maples forte. Here is a program that computes the Euclidean norm of a polynomial. I.e. given the polynomial $a(x) = \sum_{i=0}^{n} a_i x^i$ it computes the $\sqrt{\sum_{i=0}^{n} a_i^2}$.

```
EuclideanNorm := proc(a)
    sqrt( convert( map( x -> x^2, [coeffs( expand(a) )] ), `+` ) )
end;
```

Reading this one liner inside out, the input polynomial is first expanded. The `coeffs` function returns a sequence of the coefficients which we have put in a list. Each element of the list is squared yielding a new list. Then the list squares is then converted to a sum.

Why is the polynomial expanded? The `coeff` and `coeffs` functions insist that the input polynomial be *expanded* because you really can't compute the coefficient(s) otherwise. The following example should clarify

```
> p := x^3 - (x-3)*(x^2+x) + 1;

                                3               2
                      p := x  - (x - 3) (x  + x) + 1

> coeffs(p);
Error, invalid arguments to coeffs
> expand(p);
```

```
                                   2
                                2 x  + 3 x + 1

> coeffs(expand(p));
                                   1, 3, 2

> EuclideanNorm(p);
                                     1/2
                                   14
```

> ATTENTION: if you want to compute the coefficients of a polynomial in some variable, you should always expand the polynomial in that variable first. You can use the `expand` function but this expands the polynomial in all variables. Alternatively, you can use the `collect` function. `collect(p,x)` expands a polynomial p in x only. See `?collect` for other details and for working with multivariate polynomials.

The `EuclideanNorm` procedure works for multivariate polynomials in the sense that it computes the square root of the sum of the squares of the numerical coefficients. E.g. given the polynomial $p = ux^2 + y^2 + v$ the `EuclideanNorm` procedure returns $\sqrt{3}$. However, you may want to view this polynomial as a polynomial in x, y whose coefficients are symbolic coefficients in u, v. We really want to be able to tell the `EuclideanNorm` routine the polynomial variables. We can do that by specifying an additional optional parameter, the variables, as follows

```
EuclideanNorm := proc(a,v)
    if nargs = 1 then
        sqrt( convert( map( x -> x^2, [coeffs(expand(a))] ), `+` ) )
    elif type(v,{name,set(name),list(name)}) then
        sqrt( convert( map( x -> x^2, [coeffs(expand(a),v)] ), `+` ) )
    else ERROR(`invalid 2nd argument (variables)`)
    fi
end;
```

Here we are allowing the 2nd parameter v to be a single variable, or a set of variables or a list of variables. That is what the type `{name,set(name),list(name)}` matches. Notice that the `coeffs` function itself accepts this argument as a 2nd optional argument. Finally, our routine doesn't check that the input is a polynomial. Let's add that too

```
EuclideanNorm := proc(a,v)
    if nargs = 1 then
        if not type(a,polynom) then
            ERROR(`1st argument is not a polynomial`,a) fi;
        sqrt( convert( map( x -> x^2, [coeffs(expand(a))] ), `+` ) )
    elif type(v,{name,set(name),list(name)}) then
        if not type(a,polynom(anything,v)) then
```

```
            ERROR(`1st argument is not a polynomial in`,v) fi;
        sqrt( convert( map( x -> x^2, [coeffs(expand(a),v)] ), `+` ) )
    else ERROR(`invalid 2nd argument (variables)`)
    fi
end;
```

The polynom type has the following general syntax

$$\text{polynom}(\ R, X\ )$$

which means a polynomial whose coefficients are of type R in the variables X. An example is `polynom(rational,x)` which specifies a univariate polynomial in x with rational coefficients, i.e. a polynomial in $Q[x]$. If R and X are not specified, as in the first case above, that means that the expression must be a polynomial in all its variables.

Basic functions for computing with polynomials are `degree`, `coeff`, `expand`, `divide`, and `collect`. There are lots of other functions for computing with polynomials in Maple, including facilities for polynomial division, greatest common divisors, resultants, etc. Maple can also factor polynomials over different number fields including finite fields. See the on-line help for ?polynom for a list of facilities.

We conclude with a program to compute the first *primitive trinomial* of degree n over $GF(2)$ if one exists to illustrate that facilities also exist for computing with polynomials over finite fields. I.e. find an irreducible polynomial a of the form $x^n + x^m + 1$ such that x is a primitive element in $GF(2)[x]/(a)$.

```
trinomial := proc(n) local i,t;
    for i to iquo(n+1,2) do
        t := x^n+x^i+1;
        if Primitive(t) mod 2 then RETURN(t) fi;
    od;
    FAIL
end;
```

4.4 Reading and Saving Procedures: `read` and `save`

You can write one or two line Maple programs interactively. But for larger programs you will want to save them in a file. Typically, one would use an editor to write the Maple program and save it into a file, then read the program into Maple before using it. A program can be read into Maple using the read command. For example, if we have written a Maple procedure `MAX` in the file *MAX*, in Maple we read this file into Maple by doing

```
read MAX;
```

You can save your Maple procedures or any formulae that you have computed in your Maple session in a file from inside Maple using the `save` statement, which has the form

```
save f1, f2, ..., filename;
```

This saves the values of the variables `f1, f2, ...` are saved in text format in the file `filename`. You can also save Maple data in internal format or the so called ".m" format. This format is more compact and can be read faster by Maple. One simply appends ".m" to the filename. I.e.

```
save f1, f2, ..., `filename.m`;
```

This saves the values of `f1, f2, ...` in the file `filename.m` You can use the read command to read them back in to Maple, i.e.

```
read `filename.m`;
```

4.5 Debugging Maple Programs

The simplest debugging tool is the *printlevel* facility. `printlevel` is a global variable that is initially assigned 1. If you set it to a higher value, a trace of all assignments, procedure entry and exits are printed. The higher the value `printlevel` is assigned, the more levels of procedure execution will be traced. Often, however, the output from the printlevel facility will be too much. The `trace` function allows you to trace the execution of the specified functions only. Examples of these two tools have already been given in this document. We mention here one further tool. If you set the `printlevel` variable to 3 or higher, then if an run-time error occurs, Maple will print a stack trace of the calling sequence of where the error occurs. Specifically, it will print the arguments to all procedures currently being executed, and the values of the local variables and the statement being executed in the procedure where the error occurs. This is simplest illustrated by an example.

```
> f := proc(x) local y; y := 1; g(x,y); end:
> g := proc(u,v) local s,t; s := 0; t := v/s; s+t end:
> printlevel := 4:
> f(3);
Error, (in g) division by zero
        executing statement: t := v/s
        locals defined as: s = 0, t = t
        g called with arguments: 3, 1
        f called with arguments: 3
```

4.6 Interfacing with other Maple Facilities

We have shown how one can tell Maple properties about a function f by coding them as a procedure. Suppose instead you wish to teach Maple to differentiate a formula involving f for example. You may want to teach Maple how to numerically evaluate f so that it can be plotted, or how to simplify expressions involving f etc. What do you need to do? Many Maple routines have interfaces that allow you to teach these routines about your f function. This includes diff, evalf, expand, combine, simplify, series, latex, etc. To teach Maple how to differentiate a function W one writes a routine called `` `diff/W` ``. If the `diff` routine is called on an expression $f(x)$ which contains $W(g)$ then the `diff` routine will invoke `` `diff/W`(g,x) `` to compute the derivative of $W(g)$ w.r.t. x. Suppose we know that that $W'(x) = W(x)/(1 + W(x))$. Then we can write the following procedure which note explicitly codes the chain rule.

```
`diff/W` := proc(g,x) diff(g,x) * W(g)/(1+W(g)) end;
```

Hence we have

```
> diff(W(x),x);
                                W(x)
                              --------
                              1 + W(x)

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

As a second example, suppose we want to manipulate the Chebyshev polynomials of the first kind $T_n(x)$ symbolically. We can represent these in Maple as `T(n,x)`. Suppose also that for a particular value of n we want to expand `T(n,x)` out as a polynomial in x. We can tell Maple's `expand` function how to do this by writing the routine 'expand/T'. When `expand` sees `T(n,x)`, it will invoke `` `expand/T`(n,x) ``. Recall that $T_n(x)$ satisfies the linear recurrence $T_n(0) = 1$, $T_1(x) = x$, and $T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x)$. Hence we can write

```
`expand/T` := proc(n,x) option remember;
    if n = 0 then 1
    elif n = 1 then x
    else expand(2*x*T(n-1,x) - T(n-2,x))
    fi
end;
```

Even though this routine is recursive, because of the use of option remember, this routine will compute $T(100, x)$ quite quickly even though the coefficients get quite large. Here are some examples

```
> T(4,x);
                              T(4, x)

> expand(T(4,x));
                              4      2
                           8 x  - 8 x  + 1

> expand(T(100,x));

          2                                                    100
  1 - 5000 x  ... output deleted ...  + 633825300114114700748351602688 x
```

One can also tell Maple how to numerically evaluate your own function f by defining a Maple procedure `` `evalf/f` `` such that `` `evalf/f`(x) `` numerically computes $f(x)$. For example, suppose we wanted to use the Newton iteration shown earlier for computing the square root of a numerical value. Our function might look something like

```
`evalf/Sqrt` := proc(a) local x,xk,xkm1;
    x := evalf(a); # evaluate the argument in floating point
    if not type(a,numeric) then RETURN( Sqrt(x) ) fi;
    if x<0 then ERROR(`square root of a negative number`) fi;
    Digits := Digits + 3; # add some guard digits
    xkm1 := 0;
    xk := evalf(x/2); # initial floating point approximation
    while abs(xk-xkm1) > abs(xk)*10^(-Digits) do
        xkm1 := xk;
        xk := (xk + x/xk)/2;
    od;
    Digits := Digits - 3;
    evalf(xk); # round the result to Digits precision
end;
```

```
> x := Sqrt(3);
                              x := Sqrt(3)

> evalf(x);
                              1.732050808

> Digits := 50;
                              Digits := 50

> evalf(Sqrt(3));

          1.7320508075688772935274463415058723669428052538104
```

4.7 Calling External Programs

Here we are interested in how to get Maple to call an external program. There are really two reasons we might want to do this. Obviously, if Maple can't so something, and another program can, it may make sense to use the other program, instead of reimplementing an algorithm in Maple. The other reason though is efficiency. Although Maple is efficient at symbolic computations, it is not generally efficient at machine precision numerical computations. E.g. you may want to use a Fortran library routine to compute numerical eigenvectors or to do many multiple numerical integrations.

Communication of data in Maple must be done via files. The Maple program must write the data needed by the external program into an input file, using the `writeto`, `appendto` and `lprint` commands. Maple can run the external program by starting it with the `system` command. The external program must read the data from Maple from the input file and must write its results into a output file. After the program has finished executing, Maple reads the results from the output file into Maple using the `read` command. An sketch of the Maple code to do this would be

```
interface(quiet=true);
writeto(input);
... # write any data into the file input
writeto(terminal);
interface(quiet=false);
system(...); # execute the external program
read output;
... # continue processing in Maple
```

Let us look at this more closely. The first command, `interface(quiet=true);` turns off all diagnostics from Maple, i.e. bytes used messages, warnings, etc. so that they will not be written into the `input` file. This is reset by interface(quiet=false); after the data has been written to the `input` file.

The next command `writeto(input);` opens the file named `input` for writing. All Maple output from this point will go into this file. Output will typically be created by the `lprint` command. Note the use of the command `writeto` here means that the file will be overwritten. If you want to just append some data to what is already in the file use the command `appendto` instead of `writeto`.

After the data has all been written to the file, the file is closed implicitly by reseting output back to the terminal with the command `writeto(terminal);`.

The `system` command is used to execute the external program. The exact call that the system command makes will depend on the system. Under Unix, the call might look something like

```
system(`foo < input > output`);
```

The external program `foo` is executed on the input file `input` and its results are written to the file `output`. Note that the system command returns a value indicating whether the external program terminated normally. Under Unix, it returns 0 for normal termination.

Finally, we read the results in the `output` file back into Maple using the `read` command and continue execution in Maple.

In this mode, Maple calls the external program like a subroutine. After reading the results back in to Maple, Maple continues execution. This is actually a very simple means for calling an external program. Simplicity is an advantage. However, the disadvantage is that communicating data via a file of text is not efficient. In many applications this inefficiency will not matter at all, whenever the work done by the external program is significant.

We wish now to discuss in more detail how to output data from Maple into the `input` file, and how the data must be formatted by the external program so that Maple can read it in from the `output` file. Unfortunately, Maples file I/O capabilities are almost non-existent so this is somewhat difficult. Maple version V also has no primitive reading capabilities for reading say a line of numbers separated by spaces, or even a line of text. The first difficulty then is that Maple will only read data in Maple syntax. The second difficulty is that Maple V cannot read floating point data which is printed in scientific "E" notation, e.g. the number 123.456 printed in scientific notation, in Fortran, would be

```
0.123456E3
```

Neither can Maple output floating point data in this standard notation. We will provide a utility routine for converting Maple floating point numbers to strings in the "E" notation below in order that numerical data can be output by Maple using the `lprint` function. The difficulty of reading data back into Maple is that it must be a valid Maple expression. E.g. suppose the program wants to return n numerical data values $x_1, ..., x_n$. Then the `output` file should output something like

```
result := [x1, x2, ..., xn];
```

I.e. an external program must include the assignment statement, the commas, and yes, the semicolon. This is not difficult to program in `C` or `Fortran`. What is not so easy is outputting numerical data so that Maple can read it. There are two possibilities. Firstly to use decimal notation, i.e. 123.456 instead of scientific notation 0.123456E3 . The second possibility is to write floating point numbers using Maples *Float* notation, i.e. `Float(123456,-3)`. Note this difficulty has been resolved in Maple V release II which and read and write floating point data in "E" notation. Here is a program to convert a Maple floating point number into `E` notation. The program takes a optional suffix to allow you to specify `D` for Fortrans double precision. It outputs a string.

```
FFF := proc(f)
local mantissa,exponent,letter,quotient,prefix;

  if nargs = 1 then letter := 'E'
  elif type(args[2],string) then letter := args[2]
  else ERROR(`2nd argument must be a letter`)
  fi;

  if type(f,integer) then mantissa := f; exponent := 0
  elif type(f,float) then mantissa := op(1,f); exponent := op(2,f)
  else ERROR(`integer or float expected`,f)
  fi;

  if mantissa = 0 then prefix := '`0`'
  elif mantissa < 0 then prefix := '`-`'; mantissa := -mantissa
  else prefix := '``'
  fi;

  if mantissa <> 0 then
      exponent := exponent + length(mantissa) - 1;
      prefix := cat(prefix,iquo(mantissa,10^(length(mantissa)-1),'mantissa'))
  fi;

  cat(prefix,'`.`',mantissa,letter,exponent)

end:
```

For example,

```
> x := 1.234;

                                   x := 1.234

> lprint(x); # this one prints in decimal notation okay
1.234

> x := 0.001234;

                                  x := .001234

> lprint(x); # but this one prints in the Float notation
Float(1234,-6)

> y := FFF(x);

                                  y := 1.234E-3

> whattype(y);

                                     string

> lprint(y);
1.234E-3
```

4.8 Fortran and C output

It is also possible using the `fortran` and `C` commands to generate output of Maple formulae in a format suitable for a Fortran or C compiler. This is useful when you have developed a formula in Maple, or perhaps a vector or matrix of formulae, and you wish to evaluate these formulae in a Fortran or C subroutine. How do you translate the Maple formulae into Fortran or C? The Maple functions `fortran` and `C` are written for this purpose. Here is an example. Suppose we have created the following polynomial in Maple which by the way is an approximation to the complementary error function $erfc(x) = 1 - erf(x)$ on the range [2,4] accurate to 5 decimal digits.

```
 f :=  - 3.902704411 x + 1.890740683 - 1.714727839 x  + 3.465590348 x

                          7                 4                 5                  6
     - .0003861021174 x  + .5101467996 x  - .09119265524 x  + .009063185478 x
```

It is not important to know what the complementary error function is for the purpose of this example. It is in fact related to the Normal distribution in statistics. Neither is it important here to know how we created the approximation f. What we needed was a rough approximation to this function in the given range because our Fortran and C libraries did not have this function built in. For the interested reader, we used the command `chebyshev(erfc(x),x=2..4,10^(-5)` to create a Chebyshev series approximation and used our `‘expand/T‘` routine that we wrote earlier to convert it to a polynomial. To evaluate the approximation f above efficiently, we want to write the polynomial in Horner form. Then we want to generate Fortran code. We could do

```
> h := convert(f,horner);

h := 1.890740683 + ( - 3.902704411 + (3.465590348 + ( - 1.714727839

 + (.5101467996 + ( - .09119265524 + (.009063185478 - .0003861021174 x) x) x) x

) x) x) x

> fortran(h);
      t0 = 0.1890741E1+(-0.3902704E1+(0.346559E1+(-0.1714728E1+(0.510146
     #8E0+(-0.9119266E-1+(0.9063185E-2-0.3861021E-3*x)*x)*x)*x)*x)*x)*x
```

Maple has generated two lines of Fortran code complete with continuation character since the formula is longer than one line. The floating point numbers have automatically been translated to single precision Fortran `E` notation and have been truncated to 7 decimal digits. And Maple has put the result in the variable `t0` for us. Let us now output C code into a file `temp.c` assigned to the variable r. The `C` function must note first be loaded into Maple.

```
> readlib(C):
> C([r = h],filename=`temp.c`);
```

If we look at the file 'temp.c' we find that it contains

```
      r = 0.1890740683E1+(-0.3902704411E1+(0.3465590348E1+(-0.1714727839E1+(
0.5101467996+(-0.9119265524E-1+(0.9063185478E-2-0.3861021174E-3*x)*x)*x)*x)*x)*
x)*x;
```

The reader can study the help pages for `?fortran` and `?C` for additional information capabilities and options of these commands.

5 Exercises

1. Write a Maple procedure called `remove1` where `remove1(x,L)` removes the first occurrence of the value x from the list L. If x is not in the list L return FAIL.

2. Write a Maple procedure called `variance` that computes the variance of a list of numbers. I.e. `variance(x)` computes

$$\frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2$$

   where n is the number of elements of the list x and μ is the average of the numbers in list x. Your routine should print an error if the list is empty.

3. Write a Maple procedure that computes the Frobenius norm of a matrix A. The Frobenius norm is $\sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |A_{i,j}|^2}$.

4. Write a Maple procedure which sorts an array of numbers faster than the *bubblesort* algorithm in the section on arrays, e.g. by using shell sort, quick sort, merge sort, or heap sort or some other sorting algorithm. Modify your Maple procedure to accept an optional 2nd argument f, a boolean function to be used for comparing two elements of the array.

5. Write a Maple procedure which computes the Fibonacci polynomials $F_n(x)$. These polynomials satisfy the linear recurrence $F_0(x) = 1$, $F_1(x) = x$, and $F_n(x) = xF_{n-1}(x) + F_{n-2}(x)$. Compute and factor the first 10 Fibonacci polynomials. Can your program compute $F_{50}(x)$?

6. Write a Maple procedure called `structure` that when given a Maple expression returns the structure of the expression as a tree, namely

   - for atomic objects (integers and strings) just output the object
   - otherwise, return a list with the type of the object as the first element, and the remaining elements the structure of the operands of the object.

   For example: `structure(x^3*sin(x)/cos(x));` should return

```
[*, [^, x, 3], [function, sin, x], [^, [function, cos, x], -1]]
```

   Your routine should handle the Maple types integer, fraction, float, ‘`+`‘, ‘`*`‘, ‘`^`‘, string, indexed, function, range, equation, set, list, series and table. Test your function on the following input

```
Int( exp(-t)*sqrt(t)*ln(t), t=0..infinity ) =
int( exp(-t)*sqrt(t)*ln(t), t=0..infinity );
```

7. Extend the DIFF procedure to differentiate general powers, the functions ln, exp, sin, cos. and to return unevaluated instead of giving an error when it does not know how to differentiate the given input. Extend DIFF to handle repeated derivatives, i.e. how to differentiate DIFF of a DIFF and that they commute. I.e. DIFF(DIFF(f(x,y),x),y) should output the same result as DIFF(DIFF(f(x,y),y),x).

8. Write a routine that given a set of values and a non-negative integer n returns a list of all combinations of the values of size n. For example

```
> comb( {a,b,c,d}, 3 );

                   { {a,b,c}, {a,b,d}, {a,c,d}, {b,c,d} }
```

   Modify your routine to work for lists which may contain duplicates. I.e.

```
> comb( [a,b,b,c], 2 );

                        [[a,b], [a,c], [b,b], [b,c]]
```

9. The Maple function degree computes the total degree of a polynomial in a given variable(s). For example, the polynomial

```
> p := x^3*y^3 + x^4 + y^5:
> degree(p,x); # degree in x

                                     4

> degree(p,{x,y}); # total degree in x and y

                                     6
```

   We showed how one could code the degree function in Maple with our DEGREE function. However, the degree function and our DEGREE function only work for integer exponents. Extend the DEGREE so that it computes the degree of a general expression, which may have rational or even symbolic exponents. I.e.

```
> f := x^(3/2) + x + x^(1/2);
                                3/2          1/2
                          f := x    + x + x

> DEGREE(f,x);
                                    3/2

> h := (x^n + x^(n-1) + x^2) * y^m;

                              n     (n - 1)     2   m
                      h := (x   + x          + x ) y

> DEGREE(h,x);
                                 max(n, 2)

> DEGREE(h,{x,y});
```

```
max(n, 2) + m
```

10. Write Maple procedures to add and multiply univariate polynomials $a(x) = \sum_{i=0}^{m} a_i x^i$ and $b(x) = \sum_{i=0}^{n} b_i x^i$ which are represented as

    (a) a Maple list of coefficients i.e. $[a_n, a_{n-1}, \ldots, a_1, a_0]$

    (b) a linked list of coefficients i.e. $[a_n, [a_{n-1}, \ldots, [a_1, [a_0, NIL]] \ldots]]$.

11. The data structures in the above exercise are inefficient for sparse polynomials because zeros are explicitly represented. E.g. consider the polynomial $x^{10} + x + 1$. Represented as a Maple list it would be

    ```
    [10,0,0,0,0,0,0,0,0,1,1]
    ```

    and as a linked list it is

    ```
    [10,[0,[0,[0,[0,[0,[0,[0,[0,[1,[1,NIL]]]]]]]]]]].
    ```

    A more efficient representation for sparse polynomials is to store just the non-zero terms. Let us store a term $a_i x^i$ as $[a_i, i]$. Now write procedures to add and multiply sparse univariate polynomials which are represented as

    (a) a Maple list of non-zero terms i.e. $[\ldots, [a_i, i], \ldots]$

    (b) a linked list of non-zero terms i.e. $[\ldots, [[a_i, i], [\ldots, NIL]], \ldots]$

12. The Gaussian integers $Z[i]$ are the set of complex numbers with integer coefficients, i.e. numbers of the form $a + bi$ where $a, b \in Z$ and $i = \sqrt{-1}$. What is most interesting about the Gaussian integers is that they form a Euclidean domain, i.e. they can be factored into primes, and you can use the Euclidean algorithm to compute the GCD of two Gaussian integers. Given a Gaussian integer $x = a + bi$ let $||x|| = a^2 + b^2$ be the norm of x. Write a Maple procedure `REM` that computes the remainder r of two non-zero Gausian integers x, y such that $x = qy + r$ where $||r|| < ||y||$. Use your remainder routine to compute the GCD of two Gaussian integers.

13. Write a Maple n-ary procedure `GCD` that does integer and symbolic simplification of GCD calculations in Z. E.g. on input of `GCD(b,GCD(b,a),-a)` your procedure should return `GCD(a,b)`. For integer inputs, your routine should compute the GCD directly. For symbolic inputs your `GCD` procedure should know the following

    (a) GCD(a,b) = GCD(b,a) (GCD's are commutative)

    (b) GCD(GCD(a,b),c) = GCD(a,GCD(b,c)) = GCD(a,b,c) (GCD's are associative)

    (c) GCD(0,a) = GCD(a) (the identity is 0)

    (d) GCD(a) = GCD(−a) = abs(a)

14. Write a Maple procedure `monomial(v,n)` which computes a list of the monomials of total degree n in the of variables $[v_1, v_2, \ldots, v_m]$. For example, `monomial([u,v],3)` would return the list $[u^3, u^2v, uv^2, v^3]$. Hint: consider the Taylor series expansion of the product

$$\Pi_{i=1}^{m} 1/(1 - v_i t)$$

in t to order t^n – see Maples `taylor` command.

15. Given a sequence of points $(x_0, y_0), (x_2, y_2), \ldots, (x_n, y_n)$ where the x_i's are distinct, the Maple library function `interp` computes a polynomial of degree $\leq n$ which interpolates the points using the Newton interpolation algorithm. Write a Maple procedure which computes a cubic C-spline for the points. A cubic C-spline is a piecewise polynomial where each interval $x_i \leq x \leq x_{i+1}$ is defined by the cubic polynomial

$$f_i(x) \;=\; a_i x^3 + b_i x^2 + c_i x + d_i \qquad 1 \leq i \leq n$$

where the $4n$ unknown coefficients are uniquely determined by the following $4n$ conditions.

$$f_i(x_{i-1}) = y_{i-1}, f_i(x_i) = y_i$$

$$f_i'(x_i) = f_{i+1}'(x_{i-1}),$$

$$f_i''(x_i) = f_{i+1}''(x_{i-1}), \qquad f_1''(x_0) = 0, f_n''(x_n) = 0$$

These conditions mean that the resulting piecewise polynomial is C^2 continuous. Write a Maple procedure which on input of the points as a list of (x_i, y_i)'s in the form `[x0, y0, x1, y1, ..., xn, yn]` and a variable outputs a list of the segment polynomials `[f1, f2, ..., fn]`. This requires that you create the segment polynomials with unknown coefficients, use Maple to compute the derivatives and to solve the resulting equations. For example

```
> spline([0,1,2,3],[0,1,1,2],x);

          3                  3      2                           3      2
[- 1/3 x  + 4/3 x, 2/3 x  - 3 x  + 13/3 x - 1, - 1/3 x  + 3 x  - 23/3 x + 7]
```

16. Design a data structure for representing a piecewise function. The representation for the cubic spline above does interface with other facilities in Maple. Let us represent a piecewise function as an unevaluated function as follows.

$$IF(c_1, f_1, ..., c_{n-1}, f_{n-1}, f_n)$$

- Write a Maple procedure called `IF` that simplifies such a piecewise function for conditions which are purely numerical.
- Write a procedure called 'evalf/IF' which evaluates an IF expression in floating point, hence allowing IF expressions to be plotted.

- Write a procedure called 'diff/IF' which differentiates an IF expression.

E.g. your procedures should result in the following behaviour.

```
> IF( x<0, sin(x), cos(x) );

                         IF(x < 0, sin(x), cos(x))
> diff(",x);
                        If(x < 0, cos(x), - sin(x))

> IF( Pi/3<0, sin(Pi/3), cos(Pi/3) );
                                            1/2
                       If(1/3 Pi < 0, 1/2 3    , 1/2)
> evalf(");
                                .5000000000
```

Modify your IF procedure to simplify nested IF expressions, and to handle constant conditions like the example above. E.g. your IF procedure should result in the following behaviour.

```
> IF( x<0, 0, IF( x<1, x^2, IF(x>2, 0, 1-x^2) ) );

                               2                  2
              IF(x < 0, 0, x < 1, x , 2 < x, 0, 1 - x )

> IF( Pi/3<0, sin(Pi/3), cos(Pi/3) );

                                  1/2
```

17. The following iteration is known as the Halley iteration.

$$x_{k+1} = x_k - f(x_k)/f'(x_k)/(1 - \frac{f(x_k)f''(x_k)}{2f'(x_k)^2}$$

Program this iteration in Maple and check that it converges cubically. Prove that the convergence is cubic.

18. The command `dsolve` solves ordinary differential equations analytically. But not all ODE's can be solved in closed form. But it is always possible to obtain a series solution which may be useful. Suppose we are given the ODE

$$y'(x) = f(x, y(x))$$

and the initial condition $y(0) = y_0$ and we want to find the first few terms in the series

$$y(x) = \sum_{k=0}^{\infty} y_k x^k$$

Write a Maple procedure which on input of $f(x, y(x))$ and the initial condition y_0 constructs a linear system of equations to solve. I.e. let

$$y(x) = y_0 + \sum_{k=1}^{n} y_i x^i$$

substitute this finite sum into the ODE, equate coefficients and solve for the unknowns y_i. Note, you will want to use the `taylor` command to truncate the result to order n. Test your Maple procedure on the following ODE

$$y'(x) = 1 + x^2 - 2xy(x) + y(x)^2, \quad y(0) = 1$$

You should get the following series

$$y(x) = 1 + 2\,x + x^2 + x^3 + x^4 + x^5 + O(x^6)$$

Compute the solution analytically using Maples dsolve command and check that your series solution agrees with Maple's analytic solution.

Develop a Newton iteration to solve for the series $y(x)$ which converges quadratically. I.e. the k'th approximation

$$x_{k+1} = x_k - f(x_k)/f'(x_k)$$

is accurate to $O(x^{2^k})$. Thus the iteration starts with $x_0 = y_0$ and computes $x_1 = y_0 + y_1 x$ then $x_2 = y_0 + y_1 x + y_2 x^2 + y_3 x^3$ and so on. Write a Maple procedure to compute the series.

19. Modify the GaussianElimination procedure so that it does a complete row reduction of a rectangular matrix of rational numbers reducing the matrix to Reduced Row Echelon Form. Now modify your procedure to work with a matrix of formulae. Use the `simplify` function to simplify the intermediate results.

20. The use of the Taylor series for computing $erf(x)$

$$\frac{\sqrt{\pi}}{2} erf(x) \;=\; \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{n!(2n+1)}$$

becomes increasingly inefficient for large x because it converges slowly. It also becomes increasingly inaccurate because of cancellation of positive and negative terms. Write a Maple procedure which sums the terms of the asymptotic series for $erf(x)$

$$\frac{\sqrt{\pi}}{2} erf(x) \;=\; 1 - \frac{2\sqrt{\pi}}{x} e^{-2x} \sum_{n=1}^{\infty} \frac{(-1)^n 1 \times 3 \times \ldots \times 2n-1}{2^n x^{2n}}$$

for large x until it converges. How large must x be before this series converges at `Digits` decimal digits of precision? Combine the use of both the Taylor series and the asymptotic series to write a procedure that computes $erf(x)$ for all x accurate to `Digits` decimal digits of precision.