

## 8. Maple I/O; the Library; Reading and Writing Files

### 8.1 Maple Input and Output

Unless you suppress the output with a colon, Maple at least confirms your command, by default in a pretty print format. With `lprint` we get a more basic output format (may be suitable for low level communication with other programs). Adjusting the `printlevel`, we make Maple more verbose.

```
[> p := a*x^2 + b*x + c;
[> sols := solve(p,x);
```

In some situations, a linear print may be more convenient:

```
[> lprint(sols);
```

Linear printing of results may also be a way to compactify huge output produced by Maple. More explicitly, we get the same with converting to strings:

```
[> convert(sols[1],string);          # convert 1st solution to string
[> s1 := parse(%);                  # reverse the convert
```

The conversion of mathematical data to strings of characters is a useful format for a human readable interface between different computer algebra systems or other programs which permit symbolic input.

If you are curious to know about the inner workings of the solve command, then we can adjust the `printlevel`.

```
[> default_printlevel := printlevel; # to restore later
[> printlevel := 5:
[> solve(p,x);
```

Adjusting the `printlevel` is very useful for debugging your own Maple procedures.

```
[> printlevel := default_printlevel: # restore to default
```

### 8.2 The Maple Library

Maple offers a wealth of mathematical knowledge in its standard library, through various packages and the share library with user contributions. For an overview:

```
[> ?index,packages;
```

To use packages invoke the `with` command. Instead of loading an entire package (which may overwrite names you want to use), we can load only one function, e.g.:

```
[> with(orthopoly,T);          # use Chebychev orthogonal polynomials
[> T(4,x);                     # 4-th Chebychev orthogonal polynomial in x
```

Instead of loading at all (if we want to use the symbol T for something else), we may call it with its full name:

```
[> orthopoly[T](4,y);         # 4-th Chebychev orthogonal polynomial in y
```

To avoid tedious typing of long names, we can work with an `alias`:

```
[> alias(Cheby = orthopoly[T]); # give other name to orthopoly[T]
[> Cheby(4,z);                 # 4-th Chebychev orthogonal polynomial in z
```

### 8.3 Reading and Writing Files

Although worksheets can store data and instructions permanently, we may want to use Maple procedures or important data in several worksheets. First we illustrate how we can save and retrieve the data from subsection 8.1 above. For this example, we save in the directory MCS320 on the C drive:

```
[> save p, sols, "C:\\MCS320\\myfile";
```

Make sure you know why the double slashes are needed!

```
[> p := 'p': sols := 'sols':          # clear the data to test retrieval
[> p; sols;                          # check if data is gone
[> read "C:\\MCS320\\myfile";        # retrieve data from file
[> p; sols;                          # see if retrieval was fine
```

To see the file, without leaving Maple :

```
[> ssystem("type C:\\MCS320\\myfile");
```

An alternative to **save** is to redirect the output of Maple to a file:

```
[> writeto("C:\\MCS320\\myfile");
[> p;                                # writes p to the file
[> writeto(terminal);                # output back to screen
```

We can also append to a file with the command **appendto**. Notice that this is the Maple output as it appears on screen. Usually this format is only suitable for printing and not for further Maple processing.

Our second example involves a procedure:

```
[> sum_list := proc(l::list)
>   description 'returns sum of elements in the list l':
>   local i,s:                # local variables
>   s := 0:                  # initialization of result
>   for i from 1 to nops(l) do # run over all operands in l
>     s := s + l[i]:         # add the i-th entry in l
>   end do:
>   RETURN(s):              # explicit return
[> end proc;
```

Observe for a moment the long square bracket along the code of the procedure. This long square bracket indicates that the whole procedure is considered as one execution group. When we type in the procedure, we do not hit enter (or return), but use the **Insert -> Execution Group -> After Cursor** menu. After a few times, we use the short cut, hitting **j** while holding the **Ctrl** key. At the end of the procedure, we use the **Edit -> Split or Join -> Join Execution Groups** (or **F4**) to create one execution group.

Symbolic computation pioneered “generic programming”: we can use the same piece of code for summing up integers, polynomials, or just symbols. Let us see how this works:

```
[> sum_list([1, 2, 3]);
[> sum_list([a, b, c]);
[> save sum_list, "C:\\MCS320\\sumproc";
[> sum_list := 'sum_list': sum_list; # clear to test retrieval
[> read "C:\\MCS320\\sumproc";
```

## 8.4 Assignments

1. Learn about the interface variable **echo** (see the help pages).  
What is it good for? Give a good illustration about its use.
2. The command **stats[random,uniform[0,1]](20)** returns 20 random numbers, uniformly distributed between 0 and 1. Since this command is so long, we wish to create an abbreviation: **ud**.  
Give the Maple command to create **ud**, so that **ud(20)** is equivalent to **stats[random,uniform[0,1]](20)**.