B. How to Program CoCoA

The code was willing.
It considered your request,
but the chips were weak.
(Anonymous)

B.1 Defining New CoCoA Functions

Apart from using CoCoA interactively as explained in Appendix A, the system also gives you the possibility to extend it by writing your own CoCoA functions. It has a fully-fledged programming language, called CoCoAL, and provides means for packaging, storing and reusing the functions you create.

A new CoCoA function is created using a command of the form

```
Define FunctionName(Arguments)
   <Commands>
EndDefine;
```

Here `FunctionName` is the name by which you later want to call your function, `Arguments` is the (possibly empty) list of arguments it takes, and `<Commands>` denotes a group of CoCoA commands.

Let us look at an example. We want to write a function which computes the ideal generated by the p^{th} powers of the generators a given ideal, where the prime number p is the characteristic of the base field.

```
Define Frobenius(I)
   P:=Characteristic();
   L:=Gens(I);
   For N:=1 To Len(L) Do
      L[N]:=L[N]^P;
   EndFor;
   Return Ideal(L);
EndDefine;
```

Note that in order to enter this function definition on a version of CoCoA having a graphical interface, you should highlight all lines using the mouse and then evaluate this selection. Afterwards we can use our new function like any built-in CoCoA function. For instance, we can type

```
Use S::=Z/(5)[x,y,z];
I:=Ideal(xy,yz,x-y);
Frobenius(I);
```

and we get the answer

```
Ideal(x^5y^5, y^5z^5, x^5-y^5)
-------------------------------
```

After the execution of this function, your original ideal `I` is unchanged, although it has been tampered with inside the function! The reason for this is that the ideal `I` inside the function `Frobenius(I)` is just a "local" object to which CoCoA passed the value of your original ideal `I`. Instead, if you want the function to operate directly on the original ideal, you should type `Define Frobenius(Var I)` in the first line of the function definition.

B.2 Program Development

As soon as you start to write CoCoA functions more frequently, you will feel the need to develop some techniques for doing this. Since the available methods vary considerably from one operating system to another, and since almost every CoCoA programmer has his own favourite procedures, we shall restrict ourselves to pointing out the most common ways.

Many CoCoA users work with a text-based CoCoA interface inside a graphical operating system environment, e.g. inside X-Windows or Windows. In this case we recommend the "Copy and Paste" technique for creating CoCoA functions:

- Open two windows, one for CoCoA and one for a text editor. The editor should be able to save pure ASCII files.
- Create the CoCoA function inside the editor.
- Use "Copy" and "Paste" to transfer the definition to CoCoA, so that CoCoA can check it for syntax errors.
- Repeat the previous steps until the program is correct. Then you can save it using the editor in a file whose file name extension is customarily taken to be `.coc`.

Another possibility for Unix users is to run CoCoA in a shell of a suitable editor program like Emacs. (Some useful files for doing this are on the CoCoA website.) If you run a text-based version of CoCoA under a text-based operating system like DOS, you will have to revert to a more primitive technique:

• Create the function using a text editor and save it in a file, e.g. in `myfile.coc`.

• Modify the file `userinit.coc` to include a source command which reads your file at startup, i.e. include the line

```
<< 'myfile.coc';
```

• Now start CoCoA. It reads and checks your program. If there is still an error, you have to quit CoCoA and return to the editor.

A much more advanced way of using CoCoA is possible if you are able to use one of the CoCoA versions with a graphical user interface, e.g. the new graphical interface for X-Windows and Windows, or CoCoA on a Macintosh. In this case you can edit your function inside the lower window, select it and send it to CoCoA until it is correct. Probably you may not want to do this in the interactive window, but rather in a file window which you can create using `<File> + <New>`. When you are content with your function, you can then save it to a file. And don't forget to write some documentation for your functions, or else you will soon be unable to reuse them!

B.3 Input and Output

In Section B.1 we saw an example of how a CoCoA function can pass the result of its computations back to the system. The command

```
Return <Object>;
```

exits the function and returns the value of `<Object>` to the place from where the function was called.

Passing objects to a CoCoA function is obviously achieved through its argument or list of arguments. Notice that the name which you give to a function argument in the function definition does not have to agree with the name of the object on which you call the function. Moreover, the argument name in the function definition denotes a variable for which you do not have to specify a data type and which is local to the function, i.e. it may agree with the name of an object elsewhere and it will be deleted after the function has been executed. Finally, it is important to remember that objects outside the CoCoA function which are not passed to it through the list of arguments are not known inside the function. For instance, if we have

```
I:=Ideal(x^2,y^2);
Define IdealSquare(J)
   I:=J^2;
   Return I;
EndDefine;
```

then the ideal `I:=Ideal(x^2,y^2);` is not destroyed or changed during the execution of `IdealSquare(Ideal(z));` and we can call `IdealSquare(I);` without getting into trouble.

Another way of obtaining feedback from a CoCoA function, for instance during its execution, is to have it write output on the screen. This can be achieved by using the commands

```
Print <Object>,...,<Object>;
PrintLn <Object>,...,<Object>;
```

The first command is used to print a sequence of objects to the screen, and the second one does the same thing but adds a line break afterwards. For example, to print out the value of `F:=x^2+y^2-1`, we could write

```
Print 'The value of F is F = ',F;
The value of F is F = x^2 + y^2 - 1
-------------------------------
```

Finally, using CoCoA you can also write data to files and read them again. We have already mentioned the source command `<<` which allows you to read a file containing function definitions, e.g. `<< 'myfile.coc';` For reading and writing other data, you need to open an input or an output file and to write to such a "device". Since this is a very advanced use of CoCoA, we refer you to `Man('Print On');` for details about printing on a file.

B.4 Program Flow Control

Our next topic is to discuss methods for controlling the way in which a sequence of commands is executed within a CoCoA function. In order to repeat a certain sequence of commands a prescribed number of times, you can use a construction like

```
N:=10;
For I:=1 To N Do
   <Commands>
EndFor;
```

Here `I` is the name of an integer variable which is initially one, is increased by one each time the `<Commands>` have been executed, and is `N` (or an expression having an integer value) when the loop ends. For instance, the following function creates a random vector over the base field whose length is the number of indeterminates of the base ring.

```
Define RandomVector()
   N:=NumIndets();
   L:=NewList(N);
   For I:=1 To N Do
```

```
      L[I]:=Rand();
   EndFor;
   Return Vector(L);
EndDefine;
```

Another way to create a loop is provided by the command

```
ForEach M In L Do
   <Commands>
EndForEach;
```

where `M` is an object name which you can choose and `L` is a list. Here the `<Commands>` are executed for each element of the list (whose elements are examined in the obvious order), and during that execution the element is the value of the "read-only" object `M`. This loop construction is particularly useful if you do not know beforehand how many elements `L` is going to have, i.e. how often the `<Commands>` have to be performed. For example, the function

```
Define TermProduct(D)
   L:=Support(DensePoly(D));
   Result:=1;
   ForEach M In L Do
      Result:=Result*M;
   EndForEach;
   Return Result;
EndDefine;
```

computes the product of all terms of degree `D` in the current ring. Even more common is the situation that a loop has to be repeated until a certain logical condition is satisfied. This can be achieved by using

```
While <logical condition> Do
   <Commands>
EndWhile;
```

Here the `<logical condition>` is some expression which evaluates to a Boolean value. If this Boolean value is `TRUE`, then the `<Commands>` are executed, otherwise the loop ends. For instance, the following example computes the largest power of two dividing a given natural number.

```
Define MaxPower(N)
   I:=1;
   While Mod(N,2)=0 Do
      I:=I*2;
      N:=N/2;
   EndWhile;
   Return I;
EndDefine;
```

Finally, CoCoA also provides methods for executing one sequence of commands or another, depending on the value of some logical condition. The two most frequently used methods are

```
If <logical condition> Then <Commands> EndIf;
If <logical condition> Then <Cmds1> Else <Cmds2> EndIf;
```

In the first case, the `<Commands>` are executed only if `<logical condition>` evaluates to `TRUE`. Otherwise they are skipped and program execution continues after the `EndIf;` In the second case, the commands `<Cmds1>` are executed if the `<logical condition>` is `TRUE`, and the commands `<Cmds2>` are executed if it is `FALSE`. For instance, the function

```
Define MaxOdd(N)
   If Mod(N,2)=1 Then
      Return N;
   Else
      M:=MaxPower(N)
   EndIf;
   Return N/M;
EndDefine;
```

computes the largest odd divisor of a natural number `N`. Notice that the command `Return N;` leads to an immediate termination of this function if it is encountered.

Should these hints be not sufficient to solve your specific program flow control problem, or should you intend to get the "CoCoA bug", we invite you to study the CoCoA manual which contains several more possibilities and a host of examples.

B.5 List Constructions

The creators of CoCoA spent a lot of effort to make the system as user-friendly as possible. In particular, CoCoA allows you define lists of objects in ways which are very similar to the usual mathematical definition of sets in books or on the blackboard. This list construction can take a number of different forms, the most important of which are

```
[ <Object> | X In <List> ]
[ <Object> | X In <LIST> And <logical condition> ]
[X In <List> | <logical condition>]
```

In the first form, `<Object>` is a CoCoA expression which usually depends on the variable `X` and evaluates to a CoCoA object. Thus this form constructs the list of all such objects such that `X` is an element of the `<List>`. The second form does essentially the same thing, except that only those elements `X` of `<List>` are used to construct objects for which the `<logical condition>`

evaluates to TRUE. The third form creates the sublist of <List> consisting of all of its elements X for which the <logical condition> is TRUE.

Let us see some examples of list constructions. The first function returns a list of N random linear forms in the current ring, where N is a specified natural number.

```
Define RandomLinear(N)
   L:=[Randomized(DensePoly(1)) | I In 1..N];
   Return L;
EndDefine;
```

Notice that in this case I is not used in the definition of the created list elements, and observe the use of the range operator 1..N which generates the list [1,2,3,...,N]. An example for the second form of list constructions is

```
Define EvenIntegers(N)
   L:=[I | I In 1..N And Mod(I,2)=0];
   Return L;
EndDefine;
```

a function which returns the list of even integers between 1 and N. Finally, the function

```
Define IrrePolys(L)
   Irre:=[F In L | Len(Factor(F))=1];
   Irre:=[F In Irre | Comp(Factor(F),1,2)=1];
   Return Irre
EndDefine;
```

computes the sublist of a given list of monic polynomials consisting of the irreducible ones.

Maybe you have also noticed that in this example program we have a command which does not end in a semicolon, apparently contradicting what we said in Appendix A. Without falling into the abyss of computer jargon, let us just say that you can drop the semicolon if it immediately precedes

the End;

of some command.

B.6 Recursive Programming

Next we show you the perfect way for creating CoCoA programs which run into an infinite loop. More seriously, CoCoA allows a function to call itself. Of course, if you use this feature you have to make sure that the process terminates eventually. For example, a compact implementation of the computation of the factorial of a natural number is

```
Define Factorial(N)
   If N=0 Then
      Return 1;
   Else
      Return N*Factorial(N-1);
   EndIf;
EndDefine;
```

In this example finiteness is achieved by checking whether `N=0` before the recursive function call which applies to a smaller integer `N-1`. Clearly, recursive function calls are both powerful and dangerous. Let us give one more example. The following function recursively computes the nondecreasing list of prime factors of a natural number.

```
Define FactorList(N)
   If N=1 Then
      Return [];
   EndIf;
   I:=2;
   While Mod(N,I)<>0 Do
      I:=I+1;
   EndWhile;
   L:=FactorList(N/I);
   Return Concat([I],L);
EndDefine;
```

And now off to your own experiments!

B.7 Improving Your Programs

To end our whirlwind tour of CoCoA, let us show you some ways to improve and manage your CoCoA programs. The command

```
Set Timer;
```

turns on an automatic timing mechanism. Every time you enter a command, CoCoA displays the CPU time that was consumed for its execution, e.g.

```
Set Timer;
Factor(x^233-3x^223+x^222-7x^12+x^11+21x^2-10x+1);
[[x^11 - 3x + 1, 1], [x^222 - 7x + 1, 1]]
-------------------------------
Cpu time = 1.81, User time = 1
-------------------------------
```

You can use this feature to test various versions of your programs and optimize them for fastest performance. If you get sick of those timing messages, you can turn them off again by typing (guess what?) `Unset Timer;` To time a single command, you can also use `Time <Command>;` For instance, above we could have written

```
Time L:=Factor(x^233-3x^223+x^222-7x^12+x^11+21x^2-10x+1);
```

After a while, you will have a number of CoCoA functions, and if you do not use them regularly, you will forget their exact names, syntax, etc. To find their names again after you loaded the file containing them, you can use the CoCoA command

```
Starting( <String> );
```

It displays a list of the names of all CoCoA functions starting with `<String>`, including you own creations. Do not forget to enclose the string in quotes, though! And if you have a hard time remembering the syntax of the functions you wrote, we suggest that you write a help function for each of them. If the function is called `MyFunction()`, you should create a new function like

```
Define Help_MyFunction()
   <Commands>
EndDefine;
```

Then this function is executed every time you type `Help('MyFunction');` Of course, the `<Commands>` it contains will usually be commands to print some help text.

After another long while, you will have so many CoCoA functions that it becomes difficult to keep track of all their names, etc. If you come this far, you can get out of trouble by collecting your functions in packages. A CoCoA package is a file containing a lot of CoCoA functions. Usually it has a name ending with `.pkg`. Its content is of the shape

```
Package <Packagename>
   <Function Definitions>
EndPackage;
```

Later you can share your package with others, e.g. people in the CoCoA User Group or other students in your class sweating over the same homework. As this is again a very advanced way of using CoCoA, we suggest that you look at `Man('Package');` for details.

In the next appendix you can see some examples in which the previous hints are applied. If you need more precise information, we suggest that you look carefully at the CoCoA manual mentioned in Appendix A. And if all else fails, you may write an e-mail to

```
cocoa@dima.unige.it
```

But for the time being, we wish you good luck and say

```
Ciao;
-- Bye
```