A. How to Get Started with CoCoA

No keyboard present.
Hit `F1` *to continue.*
(DOS Error Message)

A.1 Getting CoCoA

The computer algebra system CoCoA is available free of charge via the internet. You may download it from the WWW page

`ftp://cocoa.dima.unige.it/cocoa/index.html`

Choose the file corresponding to your machine and operating system, and do not forget to get a copy of the manual from the directory `/doc`. Then decompress the file ending in `.zip`, or `.tgz`, or `.hqx`. Depending on your system, you will have to use a command like `unzip` or `tar -zxvf`, or the graphical interface of a decompression tool like `Winzip`.

As a result you should have an appropriately named directory or folder, for instance `/usr/local/cocoa/`, which contains the executable file of the program (called `cocoa` or `cocoa.exe`), a file named `userinit.coc`, and a subdirectory called `lib`. If, instead, you find hundreds of files in your folder, you forgot to uncompress in such a way that all paths are restored!

A.2 Starting CoCoA

Now let us try the program. You can start it by simply typing

```
cocoa
```

while you are in the folder containing the executable file. Or, if you have a graphical operating system, you can also click on its icon. And if there is no icon, create a link or shortcut to the executable file first. Then the current version of CoCoA starts up and displays a screen like

```
------------------ -------------------------------------
------   )))   ----  CoCoA 4.0
--   ___(((_ _   --
--  |       | )  --  by L. Robbiano, A. Capani, G. Niesi,
--  | CoCoA |/   --  J. Abbott, A. Bigatti, M. Caboara,
--  |       |   ---  M. Kreuzer, D. Perkinson
---  \_____/  -----  online help   : type "Man();"
-----      -------  release notes : type "RelNotes();"
------------------ -------------------------------------

-- Current ring is R = Q[t,x,y,z]
-------------------------------
```

Below we shall explain how you can enter commands by going with you through a sample session. But first, let us show you how you can exit the program. Simply type

```
Quit;
```

hit `<Enter>`, and that's it! Or maybe not. If you are using WinCoCoA (i.e. CoCoA for Windows), you can either send a line to CoCoA by typing `<Ctrl + Enter>` at the end of the line, or you can select a line or a group of lines and send it to CoCoA using the menu item `"Evaluate"`. And if you are using CoCoA on an Macintosh, you should make sure you hit the `<Enter>` key, not the one marked `<Return>`. And if, by any chance, you are in a mediterranean mood, you can also leave CoCoA with a cordial

```
Ciao;
```

A.3 Using CoCoA Interactively

In fact, when you typed `Quit;` above, you entered your first CoCoA command. A CoCoA command has two outstanding features: it starts with a capital letter, and it ends with a semicolon. If you want to define a new object, you have to give it a name starting with a capital letter. Then you can assign it a value by using `:=` as in the following example.

```
F:=x^2+y^2-1;
```

This command defines a new polynomial `F` in the current ring. (We will be talking about rings later, but for the moment we assume that you just started CoCoA and that the current ring is the default ring `R = Q[t,x,y,z]`.) The value of the new polynomial `F` is `x^2+y^2-1`. As you can see, the indeterminates of a polynomial ring in CoCoA have lower-case letters as names and are raised to powers using the `^` key. Sometimes they carry indices like `x[1],x[2],x[3]`. All other names of objects and functions in CoCoA start with a capital letter.

Now let us print out the value of our polynomial F. For this we could type `Print(F);` or simply

```
F;
```

Then CoCoA answers

```
x^2 + y^2 - 1
-------------------------------
```

We could also perform some calculation before typing its result, e.g.

```
F^2-F+1;
```

yields the output

```
x^4 + 2x^2y^2 + y^4 - 3x^2 - 3y^2 + 3
-------------------------------
```

Next we apply one of the built-in CoCoA functions.

```
Factor(F-2xy+1);
```

CoCoA's answer

```
[[x - y, 2]]
-------------------------------
```

means that $\mathtt{F} - 2\mathtt{xy} + 1 = (\mathtt{x} - \mathtt{y})^2$.

A.4 Getting Help

A moment ago we saw an example of the use of the CoCoA function `Factor()`. If you want to know more about this function, type

```
Man('Factor');
```

and CoCoA will display the page of its on-line manual describing the function `Factor()`. You may even type

```
Man('fact');
```

and achieve the same result, since CoCoA looks for all topics in its on-line manual containing the string `'fact'`. For a more extensive use of the manual, type

```
Man();
```

After that, you get a list of possible commands for getting help. For instance, `H.Toc();` will give you the table of contents of the online manual, `H.Tips();` will give you tips for using the online manual, and `H.Commands('Topic');` summarizes all commands associated with `'Topic'` (the quotes are needed to mark a string).

There are several other ways to acccess the CoCoA manual. In the directory `doc` on the ftp-server mentioned in Section A.1 you will find a html-version for reading with an internet browser and a postscript-version for printing. Finally, if you use WinCoCoA, you can also access the manual through the Windows help system.

Since this introduction to CoCoA has to be limited in space and scope, we strongly urge you to look up all the topics we discuss in the manual. You will find much more and much preciser information. Moreover, there is a section of the manual called "Tutorial" which complements these appendices and should be studied as well.

A.5 Data Types

Let us get back to our example session. Suppose we now want to define a polynomial ideal. Using the manual, we find that the correct command is `Ideal()` and we can type

```
I:=Ideal(F,x-y);
J:=Ideal(x,y,z)^2;
Intersection(I,J);
```

CoCoA will reply to the last line by printing

```
Ideal(2y^2z^2 - z^2, 2y^3z - yz, 2y^4 - y^2, x^2 - y^2,
 xz - yz, xy - y^2)
-------------------------------
```

which means that the intersection of the ideals I and J has been computed. Sometimes it may be advisable to use the command `Set Indentation;` before typing out a list or an ideal. In our case, we'd get

```
Ideal(
  2y^2z^2 - z^2,
  2y^3z - yz,
  2y^4 - y^2,
  x^2 - y^2,
  xz - yz,
  xy - y^2)
-------------------------------
```

Many CoCoA objects like polynomials, ideals, and modules are defined over a base ring. The command `CurrentRing();` tells you the value of the current ring. E.g. in our example session we would get the answer `Q[t,x,y,z]`. A new ring can be defined by a command like

```
Use S::=Z/(5)[x,y,z];
```

Here `::=` is the instruction to create a new ring, `S` is its name, `Z/(5)` is its field of coefficients, and `x,y,z` are its indeterminates. After you have defined the new ring `S`, you are also making it the current base ring via the command `Use <Ringname>;` You can switch back to your initial ring with the command `Use R;` and if you type `RingEnvs();` you get a list of all the base rings defined so far.

Whenever you create a new polynomial, ideal, or module, it is defined over the current ring. To see a list of all objects in the local memory, you can type `Memory();` and to see also the values of those objects you can use `Describe Memory();` At this point you may get a list of descriptions of objects of different types. Some of the most common data types in CoCoA are

`INT, RAT`	arbitrarily large integers resp. rational numbers
`ZMOD`	elements of a finite field $\mathbb{Z}/(p)$ where p is a prime number
`POLY`	polynomials
`IDEAL`	polynomial ideals
`RING`	polynomial rings
`VECTOR`	vectors whose entries are polynomials
`MODULE`	submodules of finite free modules over polynomial rings
`LIST`	Lists of objects of the form `[Object1, Object2, Object3]`
`MAT`	matrices of polynomials
`STRING`	strings (sequences of characters)
`BOOL`	Boolean variables (i.e. their value is either `TRUE` or `FALSE`)

In CoCoA, you usually do not have to specify the data types of the objects you create. Most sensible data type conversions are done automatically. The command `Type(Object)` displays the data type of an `Object`. If necessary, you can also force type conversions using `Cast()` e.g.

```
L:=[x,y,z];
Type(L);
```

shows that `L` is of type `LIST`, whereas

```
Cast(L,IDEAL);
```

leads to the answer

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

For special type conversions, there are frequently individual commands, e.g. the above conversion could have been done by typing `I:=Ideal(L);` Then the ideal `I` can be converted back to a list by using `Gens(I);` which brings us to our next topic: What is a list?

A.6 Lists

In CoCoA, a list is a sequence of objects, separated by commas and enclosed in square brackets. Lists are very common and useful objects in CoCoA. Examples of lists are for instance `[1,2,3]` and `[x,y,z]`. The different components of a list do not necessarily have to have the same data type, e.g. as in the list `[1,TRUE,x]`. Given a list `L:=[x,y,z];` you can access its second component by typing

```
L[2];
```

which yields the result `y`. Another way to achieve the same result is to type `Comp(L,2);` This is especially convenient is the list `L` is not directly given by its name but the result of some other evaluation, e.g. as in `Comp(Gens(I),1);` where `I` is an ideal.

Besides listing all the elements, there are a number of other ways to create a list. Two simple ones are the range operator `..` and the Cartesian product `><` The range operator `N..M` creates the list of all integers between `N` and `M`. For instance,

```
1..5;
```

creates the list

```
[1, 2, 3, 4, 5]
-------------------------------
```

The range operator can also create a list of indeterminates of the current ring (in the current term ordering), e.g. if the current ring is `Q[x,y,z,t]`, then `y..t` creates the list `[y,z,t]`, and if the current ring is `Q[x[1..8]]`, then `x[1]..x[4]` creates the list `[x[1],x[2],x[3],x[4]]`. The Cartesian product of two lists is the list of all pairs, e.g. the product `[1,2,3] >< [x,y]` yields the list `[[1, x], [1, y], [2, x], [2, y], [3, x], [3, y]]`.

There are many built-in CoCoA function for dealing with lists. We invite you to print a list of them using `Man('Functions for Lists');` In our opinion, some of the most useful ones are

a) `Append(L,O)` to append an object `O` to a list `L`,
b) `Concat(L,M)` to combine two lists `L` and `M` into one,
c) `First(L,N)` and `Last(L,N)` to retrieve the list of the first and last `N` elements of a list `L`, respectively,
d) `Diff(L,M)` to create the list of all elements of the list `L` which are not in the list `M`, and
e) `Len(L)` to determine the number of components of a list `L`.

If you want to build a new list by computing (or entering) its elements one by one, you can either apply commands such as `Append(L,x);` repeatedly, or you can first define a new "empty" list of a specified length `N` via `L:=NewList(N);` and then assign its entries using commands such as `L[1]:=x;`

CoCoA tries to be clever about operations with lists. For instance, if `L:=[x,y,z];` then `2L;` yields the list `[2x, 2y, 2z]`. But of course, it has to leave some thinking to you, and a construction like `[x,y,z]^3;` makes no unique sense.

Finally, we note that more advanced ways to construct lists will be discussed in Appendix B.

A.7 It

Finally, we would like to mention a special CoCoA variable called `It`. At each point in time it contains the last output. In particular, if you print the result of a computation and then discover that you still need it, you can assign it to a variable using `It`, e.g.

```
Factor(x^9-1);
[[x - 1, 1], [x^2 + x + 1, 1], [x^6 + x^3 + 1, 1]]
-------------------------------
L:=It;
L;
[[x - 1, 1], [x^2 + x + 1, 1], [x^6 + x^3 + 1, 1]]
-------------------------------
```

Of course, we encourage you to develop a habit of always assigning the result of a computation to a variable, e.g. using `L:=Factor(x^9-1);` and then to print it. As one CoCoA programmer put it:

Never use `It`!

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 full-fledged programming language, called CoCoAL, and provides means for packaging, storing and reusing the functions you create.

A new CoCoA function is created using the command

```
Define Functionname(Arguments)
   <Commands>
End;
```

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 End;
   Return Ideal(L)
End;
```

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)[xyz];
I:=Ideal(xy,yz,zx);
Frobenius(I);
```

and we get the answer

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

B.2 Program Development

As soon as you start to write CoCoA functions more frequently, you will feel the need to develop some technique 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. with Unix versions of CoCoA under X-Windows, or with WinCoCoA under Windows9x. 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, enter it into the system and have it checked.
• 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. CoCoA on a Macintosh or WinCoCoA. In this case you can edit your function inside the

CoCoA window, select it and send it to CoCoA until it is correct. When you are content with it, you can select it and save it to a file. Probably you will find it useful to regularly clean up the mess you created in your CoCoA window!

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 for 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
End;
```

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');` 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>
End;
```

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 numer of indeterminates of the base ring.

```
Define RandomVector()
   N:=NumIndets();
   L:=NewList(N);
   For I:=1 To N Do
      L[I]:=Rand()
   End;
   Return Vector(L)
End;
```

Another way to create a loop is provided by the command

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

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 `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
   End;
   Return Result
End;
```

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>
End;
```

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 less than or equal to a given natural number.

```
Define MaxPower(N)
   I:=1;
   While 2^I <= N Do
      I:=I+1
   End;
   Return 2^(I-1)
End;
```

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> End;
If <logical condition> Then <Commands1> Else <Commands2> End;
```

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 `End;` In the second case, `<Commands1>` are executed if the `<logical condition>` is `TRUE`, and `<Commands2>` 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)
   End;
   Return N/M
End;
```

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 plan to become a true "CoCoA bug", we invite you to study the CoCoA manual which contains several more possiblities 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 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
End;
```

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
End;
```

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

```
Define IrredPoly(L)
   Irred:=[F In L | Len(Factor(F))=1 ];
   Return Irred
End;
```

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

Maybe you have also noticed by now that in our example programs we have sometimes commands which do 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)
   End;
End;
```

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 ascending list of divisors of a natural number.

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

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. Everytime 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 funtions 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>
End;
```

Then this function is executed everytime 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>
End;
```

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 informations, we suggest that you look carefully at the CoCoA manual mentioned in Appendix A. And if all 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
```