A. The ABC of CoCoA

A: *Writing a good program*
is like writing a good paper,
but a program also needs to be fast.
(John **A**bbott)

B: *Writing a good program*
is like writing a good paper,
but a compiler is a lot fussier than a referee.
(Anna Maria **B**igatti)

C: *In mathematics*
important things are very simple,
but simple things are very difficult.
(Massimo **C**aboara)

Since the first volume of this book appeared, the computer algebra system CoCoA has evolved further and many new features and functions have been added. In this appendix we bring you up-to-date and give you more hints on how to make the most out of the program. First we recapitulate the crucial information: you can download CoCoA freely from the web page

`http://cocoa.dima.unige.it/`

Not only will you find there a download area with versions of the program for various operating systems, but you can also access a large amount of additional material: installation instructions, information about CoCoA conferences and the international CoCoA schools, links to the CoCoA discussion groups, and so on.

Before getting down to the nitty-gritty details of CoCoA usage and CoCoA programming, let us delve a bit into the philosophy of CoCoA. If you have tried your hand at some of our 99 tutorials, you will already have experienced one of the main purposes of CoCoA, namely that of providing a user-friendly and easily accessible tool for teaching Computational Commutative Algebra. A further goal is to assist mathematicians in their research. Even if they are not experts in computer programming it enables them to access the power and versatility of a modern computer algebra system without having to create machine-language-like code. For instance, in CoCoA a loop is created by typing

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

rather than something like

```
for(i=1;i<=n;i++){<Commands>}
```

And what happens if you are past the prototyping stage? Suppose you want to implement a function which is not interpreted but compiled and executed at the same speed as the built-in functions. With most computer algebra systems, you have reached the end of the line. Their source code is proprietary or incomprehensible. Not so with CoCoA: the latest version, CoCoA 5, is available as a C++ library, ready for you to expand or include in your own application. A detailed discussion of the CoCoA 5 library would not only exceed the scope of this appendix, it would also violate the down-to-earth spirit of this book. However, you can easily obtain it from the above web site.

In the next two appendices, we are going to discuss the following topics:

- how to use the graphical interface of CoCoA
- further tips and tricks for program development
- additional CoCoA programming techniques
- how to use CoCoA in your research
- how to interpret and use the parser's error messages

B. One Graphical Interface for Everybody

We are M CENSORED *t.*
Resistance is futile.
You will be assimilated.
(Anonymous)

Whether you have a computer operating under Linux, Macintosh OS X, or a current version of the Microsoft Windows operating system, there is a version of CoCoA whose graphical interface behaves essentially the same. In this section we describe this interface, including some of its lesser known but useful features, and give you hints how to employ it to full advantage.

After installing the graphical interface of CoCoA, you normally start it by double-clicking on its desktop symbol. You are greeted by a text similar to

```
-------------------------------------------------------
---           ___/       ___/         \             ---
--           /     _ \  /     _ \    , \             --
--           \    |   | \    |   |  ___ \            --
---          ____, __/  ____, __/ _/    _\          ---
-------------------------------------------------------
--     Version      : 4.4                            --
--     Online Help  : type  ?   or   ?keyword        --
--     Web site     : http://cocoa.dima.unige.it     --
-------------------------------------------------------
```

and a line telling you about the current base ring. In fact, this message is displayed in the upper part of a new window. This region is called the *output pane*. Not surprisingly, the lower region is called the *input pane*. Initially the input pane sports the title `Interactive (0)`. Let us discuss the working of these panes separately.

B.1 The Output Pane

As you may already have guessed, the purpose of the output pane is to display the results of CoCoA's computations. For instance, if we type `1+1;` in the input pane and hit `<Ctrl + Enter>`, we get the output

```
1+1;
2
-------------------------------
```

in the output pane. It shows that CoCoA passed what is affectionately known as its "sanity test". In the output pane you cannot do much editing, but you can save its contents to a file (using the command `Save Output As` in the `File` menu), or you can copy and paste text from the output pane to the input pane. To achieve the latter task, mark the text in the output pane using the mouse, hit `<Ctrl + C>`, focus on the input pane by clicking on it, and hit `<Ctrl + V>`. That's it!

B.2 The Input Pane

Enter any 11-digit prime number to continue...
(Anonymous)

The input pane offers a much more varied environment than the output pane. Its initial page is part of what is called the *interactive document*. You can open up to nine further documents, called CoCoA *documents*, which have a slightly different behaviour.

The interactive document consists of one or more pages. After typing some CoCoA commands in one of these pages, you can pass them to CoCoA's parser in several ways: by pressing `<Ctrl + Enter>`, by clicking on the gear-wheel in the toolbar, or by using the entry `Execute current command set` in the `CoCoA` menu. In each case CoCoA will perform the required computations and open a new interactive page.

A CoCoA document consists of only one page. You can execute the whole page in the same way you execute an interactive page, or you can mark part of the page with the mouse and execute it analogously. No new page is created after you execute a command set in a CoCoA document. You can save CoCoA documents to a file using `<Ctrl + S>`, and you can read them back into your CoCoA session by pressing `<Ctrl + O>` and selecting the desired file. If you have several documents open, you can move from one to the other by clicking on the appropriate tab above the input pane or via the command `<Alt + N>` where `N` is the number of the document.

In each page you have all the usual editing commands available, including `<Ctrl + C>` for copying the selected text to the clipboard, `<Ctrl + X>` for cutting the selected text and moving it to the clipboard, `<Ctrl + V>` for pasting the text from the clipboard, and `<Ctrl + Q>` for quitting the application.

B.3 The History

Those who fail to learn history
are doomed to repeat it;
those who fail to learn history correctly –
why, they are simply doomed.
(Achem Dro'hm, C.Y. 4971)

The idea underlying the division of the interactive document into separate pages is to introduce the concept of command history. You can imagine the history in the following way:

$$\boxed{\textit{page}\ 1}\quad \boxed{\textit{page}\ 2}\quad \boxed{\boxed{\textit{page}\ 3}}\quad \boxed{\textit{page}\ 4}$$

where the doubly framed rectangle represents the *history cursor*, i.e. the current position in the history. The history cursor can be moved using the keys `<Alt + Left Arrow>` and `<Alt + Right Arrow>` or the corresponding entries in the `CoCoA` menu. The current position of the history cursor is shown in the status bar of the input pane next to the label `HI`. While working on various pages of the command history, you have a number of useful shortcuts available:

1) `<Ctrl + Enter>` Execute the current command set, append it to the end of the history, and move the history cursor there.
2) `<Alt + Enter>` Append the current command set to the history, but do not execute it.
3) `<Shift + Enter>` Replace the current interactive page, but do not execute the command set.
4) `<Shift + Ctrl + Enter>` Execute the current command set, append it to the end of the history, but do not move the history cursor there. (This is useful when you are editing a command set and executing it from time to time to see whether it works or not.)
5) `<Ctrl + Del>` Delete the current page from the history.

The interactive document with its command history pages enables you to quickly enter and debug sequences of CoCoA commands. While you are doing this, the output pane keeps track of your inputs and CoCoA's replies. Take your time to learn the history *correctly* and you will not be doomed.

B.4 More Bells and Whistles

The number you have entered is imaginary.
Please divide by 0 *and try again.*
The number you have entered is irrational.
Please rotate your keyboard by 90° *and try again.*
(Anonymous)

The graphical interface of CoCoA offers you several further conveniences. In the `Settings` menu you can choose whether the input and output lines are wrapped at the window border (option `Normal Wordwrap`), at column 100 (option `Fixed Wordwrap`), or at the maximal possible number of columns (option `No Wordwrap`). Moreover, by activating the option `Autoindenting` in this menu, you can simplify the input of user-defined functions: whenever the interface recognizes a command like `Define` or `For`, it automatically indents the following lines by four additional spaces.

Last but not least you can also use the word completion mode. Activation is achieved via the entry `Autocompleting` in the `Settings` menu, or by typing `<Ctrl + Ins>`. When autocompletion is on, the interface suggests possible completions for any keyword whose initial characters you type. You can cycle through the possible completions using `<Up Arrow>` and `<Down Arrow>`.

For instance, suppose you want to compute the vanishing ideal of a projective point set whose coordinates you know. After you type `Ide`, CoCoA starts offering possible command completions starting with `Ideal`. If you press the `<Down>` key several times, it offers the possible completion `IdealOfProjectivePoints`. To accept this completion, you have two possibilities. If you hit the `<Right>` key, the desired completion is performed and the cursor is positioned at the end of the word. If you hit the `<Left>` key, the word is completed, a pair of parentheses is appended, and the cursor is positioned after `(`. In this way, you are immediately ready to input the coordinates of the points and you never forget to close the parentheses.

B.5 The Programming Environment

The owl and the pussycat went to sea
In a beautiful pea green boat
They took some honey
And plenty of money
Wrapped up in a five pound note
(Edward Lear)

To wrap up the discussion above, let us suggest a way to use the graphical interface for developing CoCoA functions and packages, e.g. for solving the programming parts of the tutorials in this book.

1) Open a new CoCoA document using the entry New in the File menu or using the key combination <Ctrl + N>.
2) Type your first function into the input pane, using Autoindenting and Autocompleting as described above.
3) When you have finished typing, save your file using the entry Save in the File menu, or using the key combination <Ctrl + S>. Give it the desired name, e.g. MyFile.coc.
4) Send the contents of the document containing your program to CoCoA by pressing <Ctrl + Enter>.
5) Activate the interactive document. Define the ring over which you intend to work, and enter the data to which you want to apply your function. Use separate interactive pages for the base ring and the data.
6) On the next interactive page, call your function with the necessary arguments. Check the parser's messages (if any) and the output.
7) If something is still wrong, activate MyFile.coc and correct your function definition. Resend it to CoCoA. Then return to the interactive document and try your function again. If the result is now the expected one, try out the function using modified inputs by going back to he appropriate page in the command history.
8) Finally, when you are done correcting, save your file MyFile.coc again.

After that you can try your hand at the next function. If you work on the same package for a number of CoCoA sessions, you may also want to put a suitable source command into your file userinit.coc. In this way, your programs are automatically loaded each time you start CoCoA.

Finally, after a lot of effort, you have done it! CoCoA has completed a hard computation, and a huge Gröbner basis G consisting of 2000 polynomials is in the memory. What next? Of course, you can print the Gröbner basis in the output pane by typing G; in the interactive document. From there you could cut-and-paste it to a standard text editor. But if the output is many thousands of lines long, this is quite cumbersome. A much better idea is to use CoCoA's built-in facilities for this purpose. To print the list G nicely to a file, you should use something like the following sequence of instructions.

```
Set Indentation;
D:=OpenOFile("MyGBasis.coc");
PrintLn "G:=" On D;
Print G On D;
PrintLn ";" On D;
Close(D);
```

The end result will be a file called MyGBasis.coc on your hard disk which contains the list G in a nicely formatted way. To read it back, it suffices to make sure that you are using the correct base ring and to type

```
<< "MyGBasis.coc";
```

C. More on CoCoA Programming

The best way to accelerate a M$\boxed{\text{CENSORED}}$*h*
is at $9.8\ m/s^2$.
(Marcus Dolengo)

After having worked their way through the appendices A, B, C of Computational Commutative Algebra 1, many readers were encouraged to try to solve their own problems with CoCoA. However, sometimes they had to contend with unforeseen obstacles: the computation started but took forever, essential CoCoA functionality seemed to be missing, or there were bugs which successfully escaped detection. If you were one of those readers, this appendix is dedicated to you. We want to show you that it is neither advisable to speed up your computer at $9.8\ m/s^2$ nor necessary to erase CoCoA from memory. Instead, we suggest you work your way through yet another appendix. The reward will be plenty of tricks for watching and managing your computations, for finding that elusive crucial example, for moving between rings effortlessly, for creating and utilizing new data types, as well as for exposing and eradicating errors.

C.1 Trust Is Good, Control Is Better

Computing Gröbner bases, Hilbert bases, border bases, SAGBI bases, and even SuperG bases is not always as easy as we would like it to be. On occasion, you pass the corresponding commands to CoCoA, the program starts computing, continues computing, keeps computing ... After waiting patiently for a while, you begin to wonder whether the computation will ever stop. Will your memory suffice? Is your life-span going to be long enough? And just how long are you willing to wait? Clearly, you need to keep tabs on what is happening inside your chips. There are several possibilities at your disposal.

Firstly, you can watch the computation from the outside. Most operating systems offer you some program to monitor the system performance: the amount of memory allocated to the CoCoA program, the percentage of CPU usage it gets, and the amount of allocated memory which has been swapped to the hard disk. Using this information, you can frequently get the big picture, i.e. you can make educated guesses as to whether you may be running out of memory soon, or whether CoCoA is mainly computing with data stored on the hard disk and the processor is mostly swapping these data in and out of the main memory.

Secondly, you can furnish your programs with `Time` commands to determine the execution time of an instruction. For instance, if you include the commands

```
Time G:=GBasis(I);
PrintLn;
```

in your function then CoCoA will display the time it took to compute the Gröbner basis `G` on a separate line, and then continue with the remaining commands of your function.

Thirdly, you can place suitable `Print` or `PrintLn` commands at strategic locations in your program. For instance, suppose that a possibly difficult toric ideal has to be found. In this case you could type something like

```
T:=Toric(M);
PrintLn "Toric ideal T has been computed ...";
```

Upon execution of your function you get some feedback about how many lines of code were processed successfully and where the computation got stuck.

Fourthly, if the problem is a particularly nasty Gröbner basis (and it often is!), you should resort to the wonderful *interactive Gröbner framework* of CoCoA. To start an interactive Gröbner basis computation of an ideal `I`, you type

```
GB.Start_GBasis(I);
```

Then there are several ways to proceed. For instance, to perform the first 1000 steps of the Gröbner basis computation and be fully informed what is going on, you should use

```
Set Verbose;
GB.Steps(I,1000);
```

Then CoCoA will print a dot for each step of the computation it has finished, and every 100 steps you will get some intermediate statistics:

```
 IPs  IVs Gens GBases MinGens MinDeg
------------------------------------
 335    0   41     86      37      6
------------------------------------
```

From this you see how far your computation has progressed. The data have the following meaning:

`IPs`	number of critical pairs still to be processed
`IVs`	number of generators still to be processed

`Gens`	number of generators of the original ideal or module
`GBases`	number of Gröbner basis elements found so far
`MinGens`	number of minimal generators found so far
`MinDeg`	minimal degree of a generator or critical pair still to be processed

Depending on this information, you decide how to continue. For instance, you can complete the Gröbner basis computation using `GB.Complete(I);` The part of the Gröbner basis computed so far is contained in `I.GBasis`. After possibly appending the original generators, you can also try to compute the Gröbner basis of the ideal or module generated by this list.

The following CoCoA program automates some of these interactive commands. It computes `N` steps of the Gröbner basis of the ideal or list of polynomials `I`, prints additional intermediate statistics (namely the minimal degree of a Gröbner basis element and the number of polynomials of minimal degree) every 100 steps, and returns the partial Gröbner basis. It is easy to adapt it to your own needs.

```
Define StepsGB(I,N)
  If Type(I)=LIST Then I:=Ideal(I) EndIf;
  Set Verbose;
  $cocoa/gb.Start_GBasis(I);
  TotalSteps:=0;
  For J:=1 To Div(N,100) Do
    $cocoa/gb.Steps(I,100);
    TotalSteps:=TotalSteps+100;
    PrintLn "Total steps computed: ",TotalSteps;
    DegList:=[Deg(F) | F In I.GBasis];
    MinDegGB:=Min([D| D In DegList]);
    PrintLn "Minimal degree in GB: ",MinDegGB;
    PrintLn "Number of polys of min degree: ",
      Len([D In DegList | D=MinDegGB]);
  EndFor;
  Return I.GBasis;
EndDefine;
```

Observe that we had to use the command `$cocoa/gb.Start_GBasis(I);` because the global alias `GB` is not recognized inside a function. Furthermore, observe that the result of an interactive Gröbner basis computation is not always the reduced Gröbner basis; it is necessary to append the command `ReducedGBasis(I);` to achieve this result.

A further useful technique available in the interactive Gröbner framework is *degree truncation*. Before starting the computation, you enter the command `I.DegTrunc:=D;` where `D` is the degree in which you want to truncate. In this way, no generators or critical pairs of degree larger than `D` will be processed. For instance, this technique is useful if `I` is homogeneous and you need to solve an ideal membership problem.

Finally, let us mention that the Gröbner framework can also be applied to compute minimal generators, syzygies, and free resolutions interactively. The corresponding commands

`GB.Start_MinGens` – start an interactive minimal generator calculation
`GB.Start_Syz` – start an interactive syzygy computation
`GB.Start_Res` – start an interactive resolution computation

and their options are explained in the CoCoA manual.

C.2 The Best Place to Find a Helping Hand

The above is the result of
exhaustive research, careful analysis,
and prolonged deliberation...
after which I flipped a coin.
(Anonymous)

In this section we propose some methods for using CoCoA to support your mathematical research. How can a computer algebra system do that? Normally, CoCoA is not much help in proving theorems — except for the geometric theorems mentioned in Section 6.7, of course. More often it is possible to decide whether a statement has a chance of becoming a theorem at all, either by checking many examples or by discovering a counterexample. Hence, to study a certain subject, it may be useful to create a collection of pertinent computer algebra routines. In particular, we provide explicit suggestions for the following three tasks.

1) Create and examine "generic" or "random" examples.
2) Search systematically for examples having special properties and stop if you find one.
3) Create auxiliary rings, do computations there, and move data to and fro.
4) Implement new data types and functions to handle them.

Seek and You Shall Find

Anyone who considers arithmetical methods
of producing random digits is, of course,
in a state of sin.
(John von Neumann)

Suppose we want to study the following *Artinian version* of the Minimal Resolution Conjecture (MRC) we discussed in Tutorial 89.

Conjecture. *Let K be an infinite field, let $P = K[x_1, \ldots, x_n]$ be standard graded, let $s \geq n+2$, let $\alpha \geq 1$ be chosen such that $\binom{\alpha-1+n}{n} \leq s < \binom{\alpha+n}{n}$, and let $V \subseteq P_\alpha$ be a generic K-vector subspace of dimension $\binom{\alpha+n}{n} - s$. Then* *the minimal graded free resolution of the ideal* $I = V \oplus P_{\alpha+1} \oplus P_{\alpha+2} \oplus \cdots$ *has the shape*

$$\begin{aligned} 0 \longrightarrow P(-\alpha-n+1)^{a_n} \oplus P(-\alpha-n)^{b_n} \longrightarrow \cdots \longrightarrow \\ \longrightarrow P(-\alpha)^{a_1} \oplus P(-\alpha-1)^{b_1} \longrightarrow I \longrightarrow 0 \end{aligned}$$

where the numbers a_i, b_i *are the numbers predicted by MRC.*

To test this conjecture in many cases, we could do with three auxiliary CoCoA functions: `CreateI`(...) to generate random ideals I of the desired kind, `GrBetti`(...) to compute the matrix of graded Betti numbers of I, and `ExBetti`(...) to list the matrix of expected Betti numbers. The first function requires us to create random homogeneous polynomials of some degree `D` via `Randomized(DensePoly(D))`. Otherwise, it appears to be straightforward to write.

```
Define CreateI(S)
   N:=NumIndets();
   A:=Alpha(S);
   V:=[Randomized(DensePoly(A)) | J In 1..(Bin(A+N,N)-S)];
Return Ideal(V)+Ideal(Indets())^(A+1);
EndDefine;
```

Here `Alpha`(...) is the support function

```
Define Alpha(S)
   N:=NumIndets();
   A:=0;
   While S>= Bin(A+N,N) Do
      A:=A+1;
   EndWhile;
Return A;
EndDefine;
```

The artifice to make the second function work is to employ the Gröbner framework. After calling `Res(I)`, we can access the matrix of graded Betti numbers via `GB.GetBettiMatrix(I)`. Short as it may be, this function includes three subtleties:

```
Define GrBetti(I)
   R:=Res(I);
   M:=Untagged($cocoa/gb.GetBettiMatrix(I));
   N:=NumIndets();
   A:=Min([J In 1..Len(M) | M[J,N]<>0]);
   B:=NewMat(2,N);
   For J:=1 To N Do
      B[1,J]:=M[A-1+J,N+1-J];
      B[2,J]:=M[A+J,N+1-J];
```

```
    EndFor;
Return B;
EndDefine;
```

Firstly, by assigning the resolution to an object R, we avoid having it printed in the output pane. Secondly, the global alias `GB` does not work inside a function and has to be substituted with `$cocoa/gb`. And thirdly, the result of `GB.GetBettiMatrix(I)` is a tagged matrix. By untagging it, we get a normal matrix (see below). The function `ExBetti`(...) was written by you in Tutorial 89.t.

Now let us shift the point of view somewhat. Suppose we work over a finite field. We want to find a homogeneous ideal in $P = K[x_1, \ldots, x_n]$ for which P/I has generic Hilbert function and for which the *linear strand* of the minimal graded free resolution is as long as possible. This means that we want the longest subtuple of $(a_1, \ldots, a_n)$ with non-zero values. Our strategy to find such examples is to create random ideals and compute the length of the linear strand of the resolution until we find interesting ones.

One way of doing this is to use a `While`- or `Repeat`-loop and compute until we find something. For instance, in the case $K = \mathbb{Z}/(3)$, $n = 5$ and $s = 10$ we can write

```
Use S::=Z/(3)[x[1..5]];
Repeat
    I:=CreateI(10);
    B:=GrBetti(I);
    L:=NonZero(B[1]);
Until Len(L)<>3;
```

After thousands of iterations we find an example whose resolution has a linear strand of length four. Another way of searching goes as follows. First we define a random number seed in the global memory. For instance, the command

```
MEMORY.CurrentSeed:=12345;
```

creates such a seed and the function

```
Define NewSeed()
    C:=MEMORY.CurrentSeed;
    S:=Mod(C*37,32003);
    MEMORY.CurrentSeed:=S;
Return S;
EndDefine;
```

generates a new seed in $\{1, 2, \ldots, 32002\}$ and updates the global variable. Then we include the command

```
    Seed(NewSeed());
```

in the function which creates our examples. If the search uncovers an interesting example and stops, the value of the global random number seed enables us to reconstruct the example. This method is particularly useful if the desired example consists of a large amount of data or a long computation.

Do we really have to compute hundreds or even thousands of examples until we discover a new species? Clearly, our chances of success depend on how exotic an animal we look for. But in this game cheating is very much allowed. For example, if you want more syzygies, it may prove worthwhile to fudge your random number generator so that it creates sparse polynomials.

When You're Through Changing You're Through

All truth passes through three stages.
First, it is ridiculed.
Second, it is violently opposed.
Third, it is accepted as being self-evident.
(Arthur Schopenhauer)

Another common task is to perform computations in various rings during the execution of a user-defined function. For instance, suppose we want to implement the method of Tutorial 51.f to compute the ideal I of algebraic relations of a tuple of polynomials $(f_1,\dots,f_n) \in K[y_1,\dots,y_m]^n$ where K is a field. The ring $P = K[x_1,\dots,x_n]$ which contains I should be constructed inside the function. Moreover, we need the auxiliary ring $\overline{Q} = K[x_0,\dots,x_n,y_1,\dots,y_m]$ which ought to be destroyed when we have finished using it. Several CoCoA commands come in handy here:

`Using <RingName> Do <Commands> EndUsing` – do computations in another ring without changing the current ring

`BringIn <Object>` – bring objects from another ring into the currently used ring

`Destroy <RingName>` – destroy a ring which is no longer needed

Let us see how we can combine these ingredients to cook up the desired function.

```
Define Implicit(F)
   M:=NumIndets();
   N:=Len(F);
   DF:=[Deg(F[J]) | J In 1..N];
   D:=Concat([1],DF,[1|J In 1..M]);
   C:=Characteristic();
   If C=0 Then
      P::=Q[x[1..N]];
      Qbar::=Q[x[0..N],y[1..M]],Weights(D);
   Else
      P::=Z/(C)[x[1..N]];
```

```
      Qbar::=Z/(C)[x[0..N],y[1..M]],Weights(D);
    EndIf;
    Using Qbar Do
      FQ:=BringIn(F);
      Fhom:=Homogenized(x[0],FQ);
      Jbar:=Ideal([x[I]-Fhom[I] | I In 1..N]);
      E:=Elim(y[1]..y[M],Jbar);
      Edeh:=Subst(E,x[0],1);
    EndUsing;
    Using P Do
      I:=BringIn(Gens(Edeh));
    EndUsing;
    Destroy Qbar;
  Return P::Ideal(I);
  EndDefine;
```

For instance, using the ring `S::=Q[y[1..2]]` and the tuple `F=[y[1]y[2], y[1]^2+y[2]^2, y[1]^3+y[2]^3]` we obtain

```
  Implicit(F);
  P :: Ideal(-2x[1]^3 + 3x[1]^2x[2] - x[2]^3 + x[3]^2)
  -------------------------------
```

With the help of `Memory();` and `RingEnvs();` you can check that not only has the ring `Qbar` been destroyed, but also all objects defined over this ring. To harness the full power of the function `Implicit(...)` the call to `Elim(...)` should be substituted with the insertion of a special elimination function for homogeneous settings.

Sometimes the simple way of moving polynomials from one ring to another via the `BringIn`(...) command does not work. In this case you have to define a ring map. For example, assume that we want to define the $\mathbb{Q}$-algebra homomorphism $\varphi : \mathbb{Q}[x_1, x_2, x_3] \longrightarrow \mathbb{Q}[y_1, y_2]$ given by $\varphi(x_1) = y_1y_2$, $\varphi(x_2) = y_1^2 + y_2^2$, and $\varphi(x_3) = y_1^3 + y_2^3$ in CoCoA, and we want to compute $\varphi(-2x_1^3 + 3x_1^2x_2 - x_2^3 + x_3^2)$. To achieve this, we apply the CoCoA commands `RMap`(...) and `Image`(...) as follows.

```
  P1::=Q[x[1..3]];
  Use P2::=Q[y[1..2]];
  Phi:=RMap(y[1]y[2],y[1]^2+y[2]^2,y[1]^3+y[2]^3);
  Image(P1::Poly(-2x[1]^3+3x[1]^2x[2]-x[2]^3+x[3]^2),Phi);
```

The result is zero, in agreement with the above implicitization. This self-evident truth shows that we are through with changing rings!

Only Dead Fish Swim with the Stream

A man can surely heat his stove by burning his furniture,
yet he should not be deluded into believing
that he has discovered a wonderful new method
for heating his premises.
(Ludwig von Mises)

When man discovered object oriented programming and its applicability to the construction of computer algebra systems, he thought that all he had to do is create a *class* for every data type he encountered in Computational Commutative Algebra, and that he had discovered a wonderful new method for computing everything. Alas, it turned out that these general classes are too inefficient to be of practical value. CoCoA offers you a few important data types which are highly optimized for good performance.

But what if the application you have in mind is not mainstream? Is there a way to make your own data types in CoCoA? How is it possible to define functions which apply to these new data types?

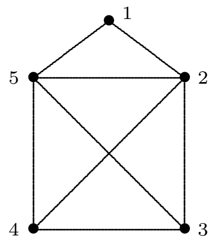
Let us assume that you are working with graphs, e.g. that you are trying to solve Tutorial 26 or 71. It is natural to define a new data type called "Graph". In CoCoA, you can use the built-in data type Record(...) for this purpose. A record consists of a sequence of fields. A field is defined by an equation <FieldName> = <Object> where <FieldName> is an identifier and <Object> is, well, a CoCoA object. The function

```
Define Graph(N,L)
   If Not (Type(N)=INT And N>0) Then
      Error("Positive integer expected.");
   EndIf;
   If Not Type(L)=LIST Then Error("List expected.") EndIf;
   M:=Len(L);
   G:=Record(VertNo=N,EdgeNo=M, Edges=L);
Return Tagged(G,"Graph");
EndDefine;
```

does some type checking and creates a record for a graph. It expects the graph to be entered in the format

```
G:=Graph(5,[[1,2],[1,5],[2,3],[2,4],[2,5],[3,4],[3,5],[4,5]]);
```

This definition corresponds to the following graph.



Notice that the above function does not return the record directly, but attaches a *tag* first. A tag is a string which you can think of as an identifier for your new data type. E.g., the query `Type(G);` now yields

```
TAGGED("Graph")
-------------------------------
```

Tags allow you to define special routines for printing and describing objects of your new data type. If you want to be fancy, create functions
`Print_Graph()` – to print graphs in the format you want; if you then type `G;` the graph `G` will be printed according to the new fashion
`Describe_Graph()` – to output an explicit description of the graph
`Help_Graph()` – to generate the reply to the request `Help("Graph");`

So, what can we do with this newly defined data type? For example, the following function creates its graph ideal in a suitable polynomial ring `P`.

```
Define GraphIdeal(G)
   If Not Type(G)=TAGGED("Graph") Then
      Error("Expected a graph");
   EndIf;
   N:=G.VertNo;
   M:=G.EdgeNo;
   P::=Z/(2)[x[1..N]];
   Using P Do
      I:=[x[Pair[1]]*x[Pair[2]] | Pair In G.Edges];
   EndUsing;
Return P::Ideal(I);
EndDefine;
```

Note how we accessed the fields of the record `G` containing our graph: the object `G.EdgeNo` contains the value of the field `EdgeNo` of the graph `G`. Next we could define a function

```
Define MaxDisconnect(G)
   I:=GraphIdeal(G);
   Using P Do
      Return Dim(P/I);
   EndUsing;
EndDefine;
```

to compute the size of a maximal disconnected subgraph (see Tutorial 71.k), and so on.

In the hope that these hints will help you cross the sea of data, record, types and functions, we now invite you to venture out on your own. And keep in mind that if you want to get back safely, you will sometimes have to swim against the stream.

C.3 To Err Is Human

Errare humanum est –
perseverare diabolicum.
(Latin Proverb)

No matter how hard we try, occasionally even the best CoCoA programmers produce an error. In this section we want to point out some ideas on how you can minimize the number of your errors, how you can find and correct them, and how you can avoid making the same error again in the future.

Better Safe than Sorry

Let us begin with the topic of *error prevention.* It is often a good idea to check the data type of the objects you deal with. Make sure that the ideal you were supposed to use as a function argument is actually an ideal by starting your function with a command such as

```
If Type(I)<>IDEAL Then Error("Expected an ideal!") EndIf;
```

If you plan to use your function with several types of arguments, perform the necessary type conversions first. To generate a function which works for both ideals and lists of polynomials, the above example could be extended to

```
If Not(Type(I) IsIn [IDEAL,LIST]) Then
  Error("Expected an ideal or a list!");
ElsIf Type(I)=LIST Then
   I:=Ideal(I);
EndIf;
```

Another method to prevent errors is to modularize your programs. Suppose you are implementing the Homogeneous Buchberger Algorithm 4.5.5. We can think of a number of useful subroutines which you may want to write: `NextDegree(...)`, `PairLoop(...)`, `GensLoop(...)`, or `UpdatePairs(...)` come to mind. By debugging those functions separately, you can decrease the likelihood of an error in your main program substantially. Furthermore, you can then collect these subroutines in a CoCoA package and reuse them on other occasions.

So, what is a CoCoA package? From the outside, it is nothing but a file having the extension `.cpkg` and containing CoCoA functions. You can load it with the `source` command just as you would load any `.coc` file. From the inside, it starts with something like `Package $contrib/MyPkgName`, continues with some function definitions, and ends with `EndPackage;` To access a function called `MyFunc(...)`, you have to type `MyPkgName.MyFunc(...)`. The advantage of doing this extra typing is that if there are two functions with the same name, the first one will not be overwritten when you load the second. To keep the overhead to a minimum, you can also define an alias for your package, e.g. you can enter

```
Alias  My := $contrib/MyPkgName;
```

and thenceforth refer to your function by typing `My.MyFunc(...)`.

The third method for preventing errors is the do-it-yourself method. (The best place to find a helping hand is at the end of your arm.) To apply this method, you proceed as follows. First you think of an example which is neither too difficult nor too trivial for your function. Then you compute this example step by step in the interactive document. Finally, you copy the commands you used one by one into the CoCoA document, all the time modifying them suitably to fit the general scheme. In this way you should get a function which computes the correct result in at least one case.

Last but not least there is a clever way to produce good CoCoA code: look through the packages which come with CoCoA and get inspired!

Keep Your Wits and Get the Message

`ERROR: parse error in line 911 of device`
(CoCoA Error Message)

Our second topic is *error discovery*. The most frequent way you find out that you committed an error is via the above well-known CoCoA error message. Its subtlety is revealed when you realize that the line number does not necessarily correspond to the actual line in which the error occurred. But don't get angry. In most cases you can detect the error by following one simple instruction:

Starting from the line number in the error message,
scan your file upwards until you find the line
where you forgot to put the `;` *Done!*

In the remaining cases, ask yourself the following questions.

1) Did I remember to close all parentheses? (If this error happens frequently, use autocompletion and `<Left>` whenever possible, or use an editor which automatically creates parentheses in pairs.)
2) Did I put the arguments of the built-in CoCoA functions in their correct order? (The CoCoA manual is really helpful here...)
3) Did I close every control structure (e.g. `For` or `While` or `If`) with its corresponding `End`-command (e.g. `EndFor` or `EndWhile` or `EndIf`)?
4) Did I use `:=` for assignments and `=` for logical comparisons and `::=` for ring definitions correctly everywhere?

Another trick the parser uses to confuse you is to claim that it does not recognize a user-defined variable although you have defined it absolutely correctly. What is behind is probably a forgotten `;` or `)` which led the parser astray. If you don't find the error by simply reading the input carefully, you might want to insert suitable `PrintLn` commands to check intermediate results of your program, or you could comment out parts of the code to localize the problem.

This bring us to the topic of *local* versus *global* objects which is a recurrent source of errors. Most CoCoA objects you create are stored in the working memory. For instance, if you have an ideal `I := Ideal(x, y);` over the ring `R` and you switch to the ring `S`, typing `I;` leads to the reply

```
R :: Ideal(x, y)
-------------------------------
```

If you define a function which uses an ideal `J` as its argument and contains an assignment to an ideal `I`, it is possible to apply this function to the ideal `I` in `R` because the names `J` and `I` are local to the function. If you want to use some objects globally, i.e. in the working memory and in user-defined functions, you have to write a statement such as

```
MEMORY.N:=4;
```

which creates the integer `N` in the global memory. Then you can access `N` from inside a function by a command such as

```
NewN:=MEMORY.N;
```

Finally, we remind you that the command `Use <Ringname>` is not allowed in user-defined functions and that global aliases (such as `GB` or `HP`) cannot be used either.

This list of sources for trouble is not intended to be comprehensive; maybe it is not even comprehensible. But we hope that it enables you to limit your search for remaining errors to the really difficult ones: mathematical errors.

Good Riddance to Those Beastly Bugs

Just because you're paranoid doesn't mean
they aren't out to get you.
(Anonymous)

Our last topic is *error digestion*. What happens after you have successfully discovered and eliminated an error? Does life continue as usual? We hope not. To prevent yourself from introducing the same error again in a later stage of the programming project, you should keep the following tenet in mind.

Every bug becomes a test.

This means that it is good practise to write a short test function which applies your program to compute a case in which the error occurred. The test function checks whether the computed result is correct. All test functions are combined in a package called the *test suite* for your program. In the main CoCoA program, test suites have usually a filename ending in `.ts` and can be managed using the `ts_lib.cpkg` package. Here is the official way to perform the CoCoA sanity test:

```
Alias TSL := $ts/ts_lib;
Test := Record[ Id = "SanityTest"];
Test.Input := "1+1;";
Test.ExpectedOutput :=
"2
-------------------------------
";
TSL.RegisterTest(Test);
TSL.Do();
```

The result should be a file called `SanityTest.result` on your computer which contains the message `Succeeded` and the message

```
-- TestSanityTest : Succeeded
```

in the output pane. Alas, if the test fails, you receive the message `FAILED!` (What else?)

Lastly, if all attempts fail and you get a bit paranoid, there is still a final resort. Get a good night's sleep! In the bright light of a new day, bugs will be spirited away. And remember that even the most powerful man on earth has to admit:

You cannot win the war on error.

C.4 Can't Stop!

A trend in motion continues
until it actually stops!
(James Dines)

Has this appendix been all work and no play? Well, this is about to stop. You can now use CoCoA to play the board game `Can't Stop!` invented by Sid Sackson. As for the rules and the necessary equipment, check out the link provided in the CoCoA manual entry of `CantStop.cpkg`. You get it by typing the command

```
$contrib/CantStop.Man();
```

Moreover, this manual entry will inform you about the available functions and computer opponents. To start a game against one of the computer opponents, simply type

```
CantStop.Play("Me","ComputerAnna");
```

and off you go! The package `CantStop.cpkg` also offers you advice on the probability of success for different moves and allows you to play computer assisted games against human opponents.