Version date: Friday 25th January, 2019 (07:39)

# CS4 Computer Algebra (2018-2019) Exercises 1
# Exploring Axiom

**Deadline:** Monday 4 February, 4.00pm.
**Note:** The three sets of exercises in this course will be allocated a period of three weeks each, however they can all be done within two weeks at most[1]. The extra week is to allow you to schedule your work taking other commitments into account. You should never leave things till the last few days, the earlier you start the better, if you need help then do ask but do this in good time.
**Total practical credit contribution:** 20%; the practical marks will add up to 100, this exercise contributes 20 towards the total.

**§1. Introduction.** The purpose of these exercises is to give you an opportunity to become acquainted with Axiom. Recall that the course web page is at

http://www.inf.ed.ac.uk/teaching/courses/ca.

Files for exercises will be placed there (amongst other course information). All links mentioned in this document can be reached by clicking on them in the pdf file. Open a terminal window and use `axiom` to run Axiom (see the bottom of the course home web page on how to get your Informatics DICE account details if you do not have them already). This handout takes you through some basics, make sure you know how to write simple code by the end of your explorations. A good way to get a feeling of what Axiom does is to download the free book at http://www.axiom-developer.org/axiom-website/bookvol1.pdf and go through the first chapter 'Introduction to Axiom'[2]. Keep this pdf for reference, note also that §5 at the end of this document gives you a summary of some useful Axiom commands (if you find commands that are frequently useful to you but not listed do please let me know).

This document also discusses various aspects of Axiom which accounts for most of its length, the tasks you are asked to carry out are fairly modest.

**Note:** Multi-line input to Axiom in an interactive window is possible but is perhaps confusing: you either carry on typing letting the line wrap or use an underscore (i.e., _) before hitting return. The problem (for me) is that using _ then return does not prevent Axiom from outputting a step number, you just type over that! This is either a bug from porting the system or a really bad design decision (most probably the former). By contrast, multi-line definitions in an input file are very convenient and mercifully free of syntactic clutter (you use indentation for grouping). Therefore it is best not to enter the rules example at the bottom of p. 8 directly[3]. The simplest thing to do is to have a file called `temp.input` (say), enter multi-line input to that and then issue `)read temp.input` or just `)read temp` (the extension `.input` is assumed).

A more detailed set of examples is given on Chapter 1 'An Overview of Axiom' of the book and you should look through this so that you know where to find some relevant examples. For programming, Chapter 5 'Overview of Interactive Language' is most relevant. You can also click on `Reference` in the Axiom X11 help window then on `Language` and look through the various topics. You should explore other aspects: e.g., in the Axiom X11 help window click on `Topics`, this will bring up a new window with several topics you can explore[4]. There is no need to do everything, use your judgment; given Axiom's scope there are very many areas that you will not

[1]The deadline for the second set is 4 weeks from the date of issue because of Flexible Learning Week.

[2]Just a couple of points to note: in examples exponentiation is shown as `**` but you can also use `^`, which is now more standard (early keyboards didn't have this symbol). Also Axiom output is shown in mathematical notation in the book but will be in rather ugly terminal format in interactive sessions. Axiom has an option to produce output in TEX command format which is how the book's output was produced.

[3]Note that the rules are the first five lines shown, the stuff in braces is the output from Axiom upon processing the rules.

[4]Unfortunately the X11 help setup is far from perfect and sometimes crashes or does not bring up a promised window. However the things you need to explore are generally sound.

be able to cover (and do not need to do so for this course). So for now you can, if you like, make sure you know the basics (obviously at least enough for these exercises) and where to find out more as needed.

This assignment requires you to

1. carry out some experiments and answer 4 simple questions given below, *[4 marks ]*
2. complete some supplied code (a few lines), *[6 marks ]*
3. write some further fairly simple Axiom code and use it. *[10 marks ]*

For the tutorial part, no preparation ahead of time is required but try to be methodical in your exploration and make brief notes (it is recommended that you read this entire document before starting). In particular, make sure you grasp the basics of programming in Axiom. You do not need to go into sophisticated features, for this course we will only write straightforward functions using things such as loops, lists, records, recursion etc. So you need to understand these. However you do not need to go through all possible variations of, e.g., `for` loops (make sure you understand how to iterate for a number of times and also over the elements of a list). You also need to pay attention to the way that Axiom handles types and domains, see below for a brief discussion.

The purpose of the code part of this exercise is to provide you with some early practice and feedback. Please note that you should type your code into the single plain text file called `ex1.input` which can be downloaded from the course web page (use this name *exactly*, note the extension). Use `)read ex1.input` or just `)read ex1` in Axiom to load it (this assumes that you are running Axiom in the directory that contains your file which is the best option). Do *not* put anything other than the code in this file, so if you carry out tests either do them interactively or put them in a separate file which you should not submit.

You might like to keep the directive `)set messages interponly off` at the head of the code file to avoid seeing messages from Axiom that do not help you at this stage (or even later on); experiment to see what difference this makes. Alternatively you can put this and other customising commands in a file `.axiom.input` in your home directory, Axiom looks for and reads such a file on startup so it is a good way to customise it[5]. Note that officially using the option `)quiet` with `)read` gets Axiom to read a file without echoing it in the interactive window; unfortunately this option is broken in some versions. For this course it is perhaps best to see the input echoed anyway, look through it in case Axiom gives any error messages about part of the input.

Each function is given a guide number of lines of well laid out code (comments are not included in this guide count but you must supply sensible comments for each function or face a penalty). The following example is reasonable convention for comments

```
-- Input:
--   e.....an object.
--   L.....a list.
-- Output:
--   A new list with e as its first element and the elements of L as the rest.
-- Assumptions:
--   The type of e is compatible with the type of objects in L, if not then
--   Axiom will issue an error at runtime.
ins(e,L)==cons(e,L)
```

(Obviously there is no need for such a function other than as an illustration.) Note that excessive comments in the code itself can get in the way of understanding and maintaining it. Use them judiciously. The meaning of the guide line count is that an implementation has been carried out in that many lines, no ingenuity was required for this and it is given as a form of further guidance. Precede each of your functions with a description of the required input, the output and any assumptions made, in the style shown above. There will be a very heavy penalty for incorrect,

[5]For example, by default Axiom asks you to confirm after issuing `)quit`, if you don't like this then put `)set quit unprotected` in `.axiom.input`.

unclear or very inefficient code though it would be hard to produce such bad code for the simple tasks in this assignment (the natural way to do things is fine).

**Warning:** Take care that you only ever enter plain text into Axiom. For example, if you convert your file `ex1.input` into rtf format and then issue `)read ex1.input` Axiom will not work as expected. With a file that is not plain text it might appear that it has been processed normally till you try to use a function defined in the file (you are likely to get confusing error messages). This is something to bear in mind if you copy and paste from anything other than plain text; importing unexpected invisible characters will cause problems (so, e.g., do not cut and paste commands from a pdf file to an interactive session). If you import non plain text by mistake it is best to quit, using `)quit`, and restart.

A slightly annoying problem is that while you can use the arrow keys to recall commands in an interactive session and then move into the text to edit, the odd character is occasionally lost or inserted! Keep an eye open for this, the cause is unclear. By now it might seem that Axiom is far too unreliable for serious use. This is not the case, the various small problems are interface related and probably stem from changes since Axiom was implemented or from porting it to new platforms. The underlying system of mathematical algorithms seems very robust

**Note:** Here and throughout the course we will adopt the standard convention that `typewriter` font indicates computer code or input at a line terminal. So `a:=2*x^2-x+3` indicates assignment to the named variable. When discussing the code it is at times clearer to indicate objects in mathematical notation so we can say that $a = 2x^2 - x + 3$ after the assignment.

**Submission:** Hand in a copy of your report with the four answers, also your answer to Exercise 4.1 and attach a printout of your code so that I can give you feedback on it. You are also required to submit your code (and nothing else) electronically, remember to submit it in the single file called `ex1.input` without any other extension. Your file should contain nothing other than the code and relevant comments to it. The report can be handwritten or typed, as you prefer but must be submitted in paper copy. *Consult the course web page instructions for details of how to submit.*

**Good Scholarly Practice:** Please remember the University requirement as regards all assessed work for credit. Details and advice about this can be found at:

http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct

and links from there. Note that, in particular, you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work on a public repository then you must set access permissions appropriately (generally permitting access only to yourself, or your group in the case of group practicals).

## §2. Some experiments.

Axiom gains strength from a notion of inheritance so that operations have a common name but become specialised as data does the same. Consider for example the notion of greatest common divisors. Technically these make sense in any unique factorisation domain (UFD), of which the archetype is the ring of integers $\mathbb{Z}$. As a consequence Axiom supplies just one operation called `gcd` whose effect is determined by the *type* of its arguments (this tells Axiom in which UFD it is working[6]). In our examples $\mathbb{Z}$, $\mathbb{Z}[x]$, $\mathbb{Q}[x]$ are specialisations of the general notion of UFD so gcd applies. Since we have $\mathbb{Z} \subset \mathbb{Z}[x] \subset \mathbb{Q}[x]$ and these are all UFDs Axiom knows that given an integer and a polynomial in $x$ with integer coefficients (i.e., an element of $\mathbb{Z}[x]$) we can take the gcd (the integer can be regarded as a constant polynomial from $\mathbb{Z}[x]$). Consider the following session:

```
(1) -> a:=1111
(1) ->
   (1)  1111
```

[6]In axiom `Domain` is a technical term that denotes kinds of objects. Each object belongs to a unique domain but can have more than one type corresponding to specialisations or extensions of its parent domain. So, e.g., 8 belongs to `Integer` and has this type but it can also have type `PositiveInteger` or `UP(x,Integer)`, i.e., univariate polynomial in `x` with integer coefficients, etc.

```
                                                         Type: PositiveInteger
(2) -> b:=111111
b:=111111
(2) ->
   (2)  111111
                                                         Type: PositiveInteger
(3) -> gcd(a,b)
gcd(a,b)
(3) ->
   (3)  11
                                                         Type: PositiveInteger
(4) -> a:=2*(x^2-1)
a:=2*(x^2-1)
(4) ->
          2
   (4)  2x  - 2
                                                     Type: Polynomial(Integer)
(5) -> b:=6*(x+1)*(x^2+2)
b:=6*(x+1)*(x^2+2)
(5) ->
          3     2
   (5)  6x  + 6x  + 12x + 12
                                                     Type: Polynomial(Integer)
(6) -> gcd(a,b)
gcd(a,b)
(6) ->
   (6)  2x + 2
                                                     Type: Polynomial(Integer)
(7) -> a:=a::UP(x,Fraction(Integer))
a:=2*(x^2-1)::UP(x,Fraction(Integer))
(7) ->
          2
   (7)  2x  - 2
                               Type: UnivariatePolynomial(x,Fraction(Integer))
(8) -> b:=b::UP(x,Fraction(Integer))
b:=6*(x+1)*(x^2+2)::UP(x,Fraction(Integer))
(8) ->
          3     2
   (8)  6x  + 6x  + 12x + 12
                                           Type: Polynomial(Fraction(Integer))
(9) -> gcd(a,b)
gcd(a,b)
(9) ->
   (9)  x + 1
                                           Type: Polynomial(Fraction(Integer))
(10) -> a:=10
a:=10
(10) ->
   (10)  10
                                                         Type: PositiveInteger
(11) -> b:=6*(x+1)*(x^2-2*x+1)
b:=6*(x+1)*(x^2-2*x+1)
(11) ->
           3     2
   (11)  6x  - 6x  - 6x + 6
                                                      Type: Polynomial(Integer)
(12) -> gcd(a,b)
gcd(a,b)
(12) ->
   (12)  2
```

Note that `UP` is shorthand for UnivariatePolynomial, quite a few abbreviations are built in (see §6 for a list of the most commonly used). The type `Fraction(Integer)` denotes the rational numbers, we can use `Fraction(D)` for any integral domain `D`[7]. We could abbreviate `Fraction(Integer)` as `FRAC(INT)`, or even `FRAC INT` (parentheses around a single argument can be omitted when the argument is a number or a symbol). Note also that `::` is used to convert from one type to another when this is possible (as would be expected, `:` is used to declare the type of an object). For the first gcd, Axiom recognises the arguments as being integers[8] and finds their gcd there, i.e., in $\mathbb{Z}$. For the next example it assigns the most general type it can to the polynomials and finds the gcd in $\mathbb{Z}[x]$. For the third example we keep the same polynomials but tell Axiom that in fact we are thinking of them as elements of $\mathbb{Q}[x]$. The effect of this is that the multiplicative constant 2 is thrown away as it is invertible in $\mathbb{Q}$ (but not in $\mathbb{Z}$, which is why it is kept for the gcd computed in $\mathbb{Z}[x]$).

Finally note the following behaviour:

```
(1) -> a:=1/2
a:=1/2
(1) ->
        1
   (1)  -
        2
                                                        Type: Fraction(Integer)
(2) -> a:=1
a:=1
(2) ->
   (2)  1
                                                          Type: PositiveInteger
(3) -> a:Fraction(Integer):=1/2
a:Fraction(Integer):=1/2
 3) ->
   You cannot declare a to be of type Fraction(Integer) because either
      the declared type of a or the type of the value of a is different
      from Fraction(Integer) .
(3) -> a:=1/2
a:=1/2
(3) ->
        1
   (3)  -
        2
                                                        Type: Fraction(Integer)
(4) -> a:Fraction(Integer):=1
a:Fraction(Integer):=1
```

[7]This is Axiom's main way of building domains for computation. So for example the field $\mathbb{Z}_n$ for $n$ a prime number is built by using the constructor `PrimeField`, i.e., `PrimeField(n)` (which can be abbreviated to `PF(n)` or `PF n`). Note that the word "domain" in Axiom denotes the single place in which an item of data resides and this is not necessarily an integral domain or indeed any algebraic structure.

[8]It is an important aspect of Axiom's design that the user is not forced to declare types in situations where they can be deduced.

```
(4) ->
   (4)  1
                                                      Type: Fraction(Integer)
(5) -> a
a
(5) ->
   (5)  1
                                                      Type: Fraction(Integer)
(6) -> a:=x^2+1
a:=x^2+1
 6) ->
   Cannot convert right-hand side of assignment
    2
   x  + 1

      to an object of the type Fraction(Integer) of the left-hand side.
(7) -> a:=1::UP(x,FRAC INT)
a:=1::UP(x,FRAC INT)
(7) ->
   (7)  1
                                                      Type: Fraction(Integer)
```

Initially we assign a fraction to `a` so Axiom sees that variable as having this type (i.e., the type is `Fraction(Integer)` or `FRAC INT` for short), natural and convenient. Next we assign an integer to `a` and so Axiom changes the type of `a` to be `Integer` (abbreviated to `INT`). We then try to assign 1/2 to `a` but do this by *declaring* the type of `a` to be `Fraction(Integer)` which is not true, it is currently `Integer`. As a result Axiom complains. We proceed to assign 1/2 to `a` at which point Axiom changes the type of `a` to be `Fraction(Integer)`. Subsequently we assign 1 to `a` but prevent any change of the type by declaring the type to be still `Fraction(Integer)`. The last attempted assignment shows that automatic changes of type are only possible with compatible ones, Axiom will not change a numeric type to one of polynomials. Perhaps the simplest thing to do when you want assign a value of a new type or to convert the value of a variable from its current type to a new one and keep the value as the new type is to store it in a brand new variable (or one you know to be of the new type).

As the examples show, Axiom does a good job of deducing type information so that it is not necessary to give it for many uses. This is convenient especially for interactive sessions. However under some circumstances it cannot deduce the type (largely for efficiency reasons) and the user must therefore supply enough information to help the system. You are unlikely to come across this for the exercises in this course but it is important to bear it in mind.

**Question 1:** What does Axiom do if in the first example we use `a:=1111::Fraction(Integer)` but keep the declaration of `b` as before (i.e., `b:=111111`? Explain why Axiom's answer is correct. *[4 marks]*

**Question 2:** In Axiom an expression such as `[1,4,9]` denotes a list (in this case of integers). Suppose `L` is a list of numbers (or indeed elements from any ring). How would you write a single line command involving a `for` loop to print out the square of each entry in `L`? Note that the way to get the `i`th entry of a list in Axiom is `L.i` (this is a fairly uniform notation for various compound structures, e.g., it works the same way for records). However there is a neater way to write the loop that does not involve explicit indexing into the list.

**Question 3:** Enter the single line function definition `insert(L,i,e)==(L.i:=e; L)` (in an interactive session). This takes as arguments a list `L` an index `i` to the list (assumed to be valid) and an element `e` (assumed to be of the correct type). The function returns the list obtained by replacing the `i`th entry of the list `L` by `e`. Test the function with some valid calls. Now declare `L:=[1,2,3]` and try the command `insert(L,5,1)` what happens? Issue the command `insert(L,1,x)` what happens? Finally issue the command `insert(L::List(POLY INT),1,x)`. Comment very briefly on the appropriateness of these behaviours.

Note that `==` in Axiom is delayed assignment, it tells the system not to evaluate things until necessary (e.g., for a function definition evaluate at the time of calling the funciton). We will use this for the definition of functions though it can be used in other contexts as well.

**Question 4:** We could have specified types for the function `insert` as follows:

```
insert:(List(Integer),PositiveInteger,Integer)->List(Integer)
insert(L,i,e)==(L.i:=e; L)
```

or as

```
insert(L:List(Integer),i:PositiveInteger,e:Integer):List(Integer)==(L.i:=e; L)
```

Explain an application advantage of *not* specifying the types.

Note carefully the two mechanisms for declaring the types of functions, you will need them in the future. Which one you use is partly a matter of taste but also of practicality, the first method is arguably preferable in most cases as it keeps function code tidy. On the other hand if the function header and the types are fairly brief then the second method seems fine.

**§3. Sturm sequences.** In this section you will complete the implementation of an algorithm that we will study later on in the course, most of the code is supplied in the file `ex1.input` which you can download from the course web page you only need to add a few lines to the function `Sturm` (see Figure 1). The amount of code is fairly modest but it brings up some subtle issues connected with Axiom's type system; you are encouraged to experiment interactively to see the various points in action.

An age old problem that is still of interest is to count the number of real roots of a univariate polynomial with real coefficients in a given interval. In 1834 Sturm published a remarkable theorem that shows us how to do this exactly without ever trying to find any roots. Before we state this we need a simple definition: a polynomial is said to be *square free* if it has no repeated non constant factors, this is the same as saying that it has no repeated roots (real or complex). We will see later on that for any non-zero polynomial $p$ the polynomial $p/\gcd(p,p')$ has the same factors (hence roots) as $p$ but without repetitions (here $p'$ is the derivative of $p$). We now assume that $p$ is square free. Consider the sequence

$$\mathrm{Sturm}(p) = p_0, p_1, \ldots, p_n$$

defined by

$$\begin{aligned} p_0(x) &= p, \\ p_1(x) &= p', \\ p_i &= -\,\mathrm{remainder}(p_{i-2}, p_{i-1}), \quad \text{for } i \geq 2. \end{aligned}$$

where 'remainder' means the remainder from polynomial division. The process stops when we get to a constant non-zero polynomial, so this happens after at most $\deg(p)$ steps. For a number $a$ define $\mathrm{Sturm}_a(p)$ to be the sequence of numbers obtained by evaluating the sequence $\mathrm{Sturm}(p)$ at $a$ (i.e., substitute $a$ for the indeterminate). We define the *variation* of a sequence of numbers to be the number of times we change sign in going from the first number to the last, ingnoring any occurrences of 0. Thus the sequence $1, -1, 1$ has 2 variations as does $0, 1, 0, 0, 0, -1, 0, 1$. Define $V(p, a)$ to be the variation of the sequence $\mathrm{Sturm}_a(p)$.

Sturm's theorem states that if $p$ is square free and $a \leq b$ are real numbers then the number of roots of $p$ in the half open interval[9] $(a, b]$ is exactly $V(p, a) - V(p, b)$. As we will see in the course,

[9]By definition $(a, b]$ is the set of real numbers $r$ that satisfy $a < r \leq b$, note that if $a = b$ then the set is empty.

we can use this to approximate the real roots of $p$ to any degree of accuracy. For now we will just focus on counting the number of roots in an interval. Figure 1 shows most of the required code. It is also possible to use Sturm sequences directly to count the total number of real roots, this will be discussed later in the course.

There is a subtlety in connection with `rootsCount` that is important to appreciate. At first sight it might seem odd that we use the type `POLY(FRAC(INT))` which allows polynomials in any number of indeterminates so long as they have rational coefficients whereas we want only such polynomials that are univariate. So it seems better to use the type `UP(x,FRAC(INT))`. This would work fine[10] but only so long as the indeterminate is `x`, so it accepts $x^2 - 1$ but complains about $y^2-1$. This is because the `x` in the type `UP(x,FRAC(INT))` of the header is a literal and does not get bound at the time of call. In the code itself we can get around this by assigning the indeterminate to a local variable `x` after which uses of the type `UP(x,FRAC(INT))` will be interpreted as referring to the value that `x` holds. This observation is useful for the function `Sturm` whose code you will complete. The specification is:

**Declaration:** `Sturm(p:POLY(FRAC(INT))):List(POLY(FRAC(INT)))`

*[6 marks ]*

**Parameter:** `p` a polynomial with rational coefficients

**Output:** A list consisting of the Sturm sequence of the square free part of `p` provided that this is a univariate non-zero polynomial with rational coefficients, otherwise an error is signalled.

**Number of lines:** 20 (this is the total number of lines of my complete implementation, so this counts the ones you are given not just the few you need to write).

**Remark:** The code checks that the conditions on `p` are satisfied and, if not, then an error is signalled. Note that although we carry out checks in `rootsCount` it is reasonable to expect that `Sturm` will be called on its own by users and so checking is carried out here as well.

Take care to have the correct type assigned to objects when applying an operation, this is particularly important for `rem`. Consider the following session:

```
(1) -> f1:=x^2-1
(1) ->
          2
   (1)  x  - 1
                                                 Type: Polynomial(Integer)
(2) -> g1:=2*x
g1:=2*x
(2) ->
   (2)  2x
                                                 Type: Polynomial(Integer)
(3) -> f1 rem g1
f1 rem g1
   There are 2 exposed and 1 unexposed library operations named rem
      having 2 argument(s) but none was determined to be applicable.
      Use HyperDoc Browse, or issue
                                )display op rem
      to learn more about the available operations. Perhaps
      package-calling the operation or using coercions on the arguments
      will allow you to apply the operation.
```

[10] There is another subtlety that can be confusing. The symbol `x` of a polynomial `p` passed to the alternative function becomes implicit and different from any other `x` we might use in the body. So Axiom complains if we use `D(p,x)` in this version but will work fine with `D(p)`. To see this try `test1(f:UP(x,FRAC(INT))):UP(x,FRAC(INT))==D(f,x)` as compared to `test2(f:UP(x,FRAC(INT))):UP(x,FRAC(INT))==D(f)`.

```
--You MUST supply appropriate comments here, see below for a style.
Sturm(p:POLY(FRAC(INT))):List(POLY(FRAC(INT)))==
  local V,x,S
  if p=0 then error("The polynomial must not be 0")
  else if totalDegree(p)=0 then [p]
  else
    V:=variables(p)
    if #(V)>1 then error("The polynomial must be univariate")
    x:=V.1
    q:=p/gcd(p,D(p,x))
    q1:???:=q   -- replace the two ??? by the appropriate type.  Remove this comment.
    q2:???:=D(q,x)
    S:=[q1,q2]
       repeat
         --Fill in the next few lines to complete finding the Sturm sequence.  Remove this comment.
         ???
         ...
         ???
    S

--Input:
--  L.....a list of rationals.
--Output:
--  The number of sign variations in L.
variation(L:List(FRAC(INT))):NNI==
  local v
  if L=[] then 0
  else
    v:=0
    p:=L.1
    for n in L repeat
       if n~=0 then
         if p~=0 and sign(p)~=sign(n) then v:=v+1
         p:=n
    v

--Input:
--  p.......a polynomial with rational coefficients.
--  a,b.....rational numbers.
--Output:
--  The number of real roots of p in (a,b].
--Checks:
--  If p is not univariate or a>b then an error is signalled by the code.
rootsCount(p:POLY(FRAC(INT)),a:FRAC(INT),b:FRAC(INT)):Union(NNI,String)==
  local V,x,S
  if a>b then
    error("The first endpoint of the interval cannot be strictly larger than the second.")
  V:=variables(p)
  if #(V)>1 then error("The polynomial must be univariate")
  if p=0 then "infinite"
  else if totalDegree(p)=0 then 0
  else
    if #(V)=1 then x:=V.1
    S:=Sturm(p)
    variation(eval(S,x=a))-variation(eval(S,x=b))
```

Figure 1: Code in the file `ex1.input` from the course web page.

```
   Cannot find a definition or applicable library operation named rem
      with argument type(s)
                              Polynomial(Integer)
                              Polynomial(Integer)

      Perhaps you should use "@" to indicate the required return type,
      or "$" to specify which version of the function you need.
(3) -> f2:UP(x,FRAC(INT)):=x^2-1
f2:UP(x,FRAC(INT)):=x^2-1
(3) ->
          2
   (3)  x  - 1
                                Type: UnivariatePolynomial(x,Fraction(Integer))
(4) -> g2:UP(x,FRAC(INT)):=2*x
g2:UP(x,FRAC(INT)):=2*x
(4) ->
   (4)  2x
                                Type: UnivariatePolynomial(x,Fraction(Integer))
(5) -> f2 rem g2
f2 rem g2
(5) ->
   (5)  - 1
                                Type: UnivariatePolynomial(x,Fraction(Integer))
```

The behaviour is *not* a bug. It does not make sense to ask for the remainder of objects of type `POLY(INT)` (mathematically) whereas it does do so for objects of type `UP(x,FRAC(POLY)` (again mathematically).

**Useful Axiom functions:** `totalDegree`, `gcd`, `variables`, `error`, `rem`, `append`, `break`.

**Examples:** `(12) -> Sturm(x^2-1)`

```
Sturm(x^2-1)
(12) ->
           2
   (12)  [x  - 1,2x,1]
                                      Type: List(Polynomial(Fraction(Integer)))
(13) -> Sturm((x^2-1)^4)
Sturm((x^2-1)^4)
(13) ->
           2
   (13)  [x  - 1,2x,1]
                                      Type: List(Polynomial(Fraction(Integer)))
(14) -> Sturm(x^2+1)
Sturm(x^2+1)
(14) ->
           2
   (14)  [x  + 1,2x,- 1]
                                      Type: List(Polynomial(Fraction(Integer)))
(15) -> Sturm(x^3-7*x+7)
Sturm(x^3-7*x+7)
(15) ->
           3             2     14       1
   (15)  [x  - 7x + 7,3x  - 7,-- x - 7,-]
                               3        4
                                      Type: List(Polynomial(Fraction(Integer)))
```

```
(16) -> Sturm(1)
Sturm(1)
(16) ->
   (16)  [1]
                                        Type: List(Polynomial(Fraction(Integer)))
(17) -> rootsCount(x^2-1,-1,1)
rootsCount(x^2-1,-1,1)
(17) ->
   (17)  1
                                             Type: Union(NonNegativeInteger,...)
(18) -> rootsCount(x^2-1,-1,2)
rootsCount(x^2-1,-1,2)
(18) ->
   (18)  1
                                             Type: Union(NonNegativeInteger,...)
(19) -> rootsCount(x^2+1,-100,100)
rootsCount(x^2+1,-100,100)
(19) ->
   (19)  0
                                             Type: Union(NonNegativeInteger,...)
(20) -> rootsCount(x^3-7*x+7,-3,3)
rootsCount(x^3-7*x+7,-3,3)
(20) ->
   (20)  2
                                             Type: Union(NonNegativeInteger,...)
(21) -> rootsCount(x^3-7*x+7,-10,3)
rootsCount(x^3-7*x+7,-10,3)
(21) ->
   (21)  3
                                             Type: Union(NonNegativeInteger,...)
(22) -> rootsCount(0,-10,3)
rootsCount(0,-10,3)
(22) ->
   (22)  "infinite"
                                                     Type: Union(String,...)
(23) -> rootsCount(1,-10,3)
rootsCount(1,-10,3)
(23) ->
   (23)  0
```

**§4. Modular Arithmetic with Polynomials.** Consider a field $k$ (e.g., the rational numbers $\mathbb{Q}$ or the integers modulo a prime number $n$, denoted by $\mathbb{Z}_n$), an indeterminate $x$ over $k$ and a non-zero polynomial $p(x) \in k[x]$. We will see that the the Euclidean algorithm holds here: if $h(x)$ is any polynomial we may write it uniquely as

$$h(x) = p(x)q(x) + r(x)$$

where $r(x) = 0$ or $\deg(r(x)) < \deg(p(x))$. We call $r(x)$ the *remainder* of $h(x)$ modulo $p(x)$ and write

$$h(x) \equiv r(x) \pmod{p(x)}.$$

Alternatively we may denote $r(x)$ by $h(x) \bmod p(x)$. Now put

$$k[x]/(p(x)) = \{h(x) \bmod p(x) \mid h(x) \in k[x]\},$$

i.e., the set of all remainders modulo $p(x)$. Note that the parenthesis around $p(x)$ in $k[x]/(p(x))$ are *not* optional (we will see why later in the course; they denote the *ideal* of $k[x]$ generated by $p(x)$). In contrast we frequently write just $p$ instead of $p(x)$, we have kept the 'argument' in this early exercise to stress that we are dealing with polynomials in $x$. We have simplified matters here: formally the elements of $k[x]/(p(x))$ are the equivalence classes of the relation

$$f(x) \sim g(x) \iff (f(x) \bmod p(x)) = (g(x) \bmod p(x)) \text{ in } k[x].$$

The right hand side is equivalent to $p(x) \mid f(x) - g(x)$ in $k[x]$. What we have done above is to replace each equivalence class with a unique representative from it. For example take $k = \mathbb{Q}$ and $p(x) = x^2 + 1$. Then $x$ as an element of $\mathbb{Q}[x]/(x^2+1)$ stands for the class $\{(x^2+1)h(x) + x \mid h(x) \in \mathbb{Q}[x]\}$. So, when viewed as elements of $\mathbb{Q}[x]/(x^2+1)$, the polynomials $x$, $x + (x^2+1)$ and $x + (x^3 - 3x^2 + 5x - 6)(x^2+1)$ are equal and represented by the remainder $x$. It is useful to compare this practice with the case of the integers modulo an integer. In fact we could choose to work with any set of representatives, one from each equivalence class. Bearing the analogy with the integers modulo an integer in mind we define addition and subtraction on our new set in the obvious way:

$$\begin{aligned} r_0(x) + r_1(x) \quad &\text{means} \quad (r_0(x) + r_1(x)) \bmod p(x), \\ r_0(x)r_1(x) \quad &\text{means} \quad (r_0(x)r_1(x)) \bmod p(x). \end{aligned}$$

More accurately we are defining the operations on equivalence classes. Using $[f]$ to denote the equivalence class of $f$ (i.e., $\{g \mid p \mid f - g\}$), we have defined $[f] + [g] = [f + g]$ and $[f][g] = [fg]$. It is not hard to show the following: if $f_1$, $f_2$ belong to the same equivalence class and similarly for $g_1$, $g_2$ then $f_1 + g_1$ belongs to the same equivalence class as $f_2 + g_2$; likewise for $f_1g_1$ and $f_2g_2$. Thus the operations on equivalence classes are well defined; we will return to these points in full generality later on in the course.

Recall that in the case of the integers modulo a given integer we obtain a commutative ring with identity. In fact the same happens in the polynomial case and the notation $k[x]/(p)$ is used to denote the ring thus obtained (note that we have dropped the 'argument' from $p(x)$ since it is clear from the context). The rather simple programming task of this part is to implement the operations of this ring for the case $k = \mathbb{Z}_n$ where $n$ is a prime number.

You are required to write two simple functions as specified below. These are largely an exercise in type conversion with a bit of checking thrown in Axiom can do most of for you. Note that you can use variables in a function block without declaration, they are implicitly local but this is not good practice. You can declare a list of variables `v1`, `v2`, ... as local by using `local v1, v2, ...`. If you want to use a global variable (not recommended) then you can declare it by using `free`.

First an important point. We want our functions to work with univariate polynomials, all in the same indeterminate. It is tempting to declare the type of these to be `UP(x,INT)`. This would indeed work but it has the drawback that our code will work only with polynomials in $x$ (the type declaration is fixed, not a template). We can get around this by declaring the types of the polynomials to be `POLY(INT)` and then add a bit of code to check they are as needed. The following does the job for $f$ and signals an error if necessary otherwise it stores the indeterminate in the local variable `x` (see the function `Sturm` on p.10).

```
V:=variables(f)
if #(V)>1 then error "Polynomials must be univariate in the same indeterminate."
else if #(V)=1 then x:=V.1
```

Think about what happens if $f$ has no indeterminates and why the code is correct. Use the code above appropriately in the functions you write (there is a neat way to test that the three polynomial parameters of the functions pass the test).

Write a function `padd` with the following specification:

**Declaration:**

```
padd:(POLY(INT),POLY(INT),POLY(INT),PositiveInteger)->Any
padd(q1,q2,p,n).
```

[*4 marks* ]

| | | |
|---|---|---|
| **Parameters:** | `q1,q2,p` | univariate polynomials in the same indeterminate with integer coefficients where `p` is non-zero in $\mathbb{Z}_n[x]$. |
| | `n` | a prime number. |

**Output:** The representative of $q_1 + q_2$ in the ring $\mathbb{Z}_n[x]/(p)$.

**Number of lines:** 10 (including type declaration and header).

**Remarks:** For this and the next function we are using Axiom's type checking to the full. Experiment to see what Axiom does if any of the parameters is not as required (e.g., $p = 0$ in $\mathbb{Z}_n[x]$, note that this is *not* the same as requiring $p$ to be the non-zero polynomial in $\mathbb{Z}[x]$).

It is at first surprising that the return type is given as `Any` rather than, e.g., `UP(x,PF n)` (recall that `PF` is the abbreviation for `PrimeField`). The difficulty here is that the constructor `PrimeField` needs to know the value of its argument so that it can check it is indeed a prime number. The use of the special domain and associated type `Any` gets around this. We get our function to return a result of type `UP(x,PF n)` by arranging for this in its body, here we can declare things to be of this type because by the time it is used `n` has a value. At this stage Axiom will check if `n` is a prime and proceed if it is otherwise signal an error. The trick of using `Any` is of course not ideal since the user cannot determine the result type without checking the code, similarly for the compiler.

**Useful Axiom functions:** `error`, `rem` (this is infix, i.e., `f rem g`, take care as regards its precedence).

**Examples:**

```
(3) -> )read paddTest
)read paddTest
padd(3*x^5-4*x^2+x-2,5*x^4+10*x-1,2*x^2+1,11)


   (7)  9x + 3
                                          Type: UnivariatePolynomial(x,PrimeField(11))
padd(x^2+1,2*x^3-x+10,x^3+x+1,7)


         2
   (8)  x  + 4x + 2
                                           Type: UnivariatePolynomial(x,PrimeField(7))
padd(y^3-2*y+1,y^4+1,y-1,3)


   (9)  2
                                           Type: UnivariatePolynomial(y,PrimeField(3))
padd(z^2+1,2*z^3-z+10,z^3+z+1,5)


          2
   (10)  z  + 2z + 4
                                           Type: UnivariatePolynomial(z,PrimeField(5))
padd(x^2+1,2*y^3-y+10,x^3+x+1,5)


   Error signalled from user code:
      Polynomials must be univariate in the same indeterminate.
```

You can download the test file `paddTest.input` from the course web page. Of course you should test your implementation further. Note that the test causing an error has been placed last since Axiom stops processing input when as error is found; in an interactive session you can carry on with more input.

Write a function `pmul` with the following specification:

**Declaration:**
`pmul:(POLY(INT),POLY(INT),POLY(INT),PositiveInteger)->Any`
`pmul(q1,q2,p,n).`

| | | | |
|---|---|---|---|
| **Parameters:** | `q1,q2,p` | univariate polynomials in the same indeterminate with integer coefficients where p is non-zero in $\mathbb{Z}_n[x]$. | *[2 marks ]* |
| | `n` | a prime number. | |

**Output:** The representative of $q_1q_2$ in the ring $\mathbb{Z}_n[x]/(p)$.

**Number of lines:** 10 (including type declaration and header).

**Remarks:** See the remarks for the function `padd`.

**Useful Axiom functions:** see the entry for `padd`

**Examples:**

```
(6) -> )read pmulTest
)read pmulTest
pmul(3*x^5-4*x^2+x-2,5*x^4+10*x-1,2*x^2+1,11)


   (7)  8x + 5
                                          Type: UnivariatePolynomial(x,PrimeField(11))
pmul(x^2+1,2*x^3-x+10,x^3+x+1,7)


         2
   (8)  x  + 4
                                           Type: UnivariatePolynomial(x,PrimeField(7))
pmul(y^3-2*y+1,y^4+1,y-1,3)


   (9)  0
                                           Type: UnivariatePolynomial(y,PrimeField(3))
pmul(z^2+1,2*z^3-z+10,z^3+z+1,5)


           2
   (10)  3z  + 1
                                           Type: UnivariatePolynomial(z,PrimeField(5))
pmul(x^2+1,2*y^3-y+10,x^3+x+1,5)


   Error signalled from user code:
      Polynomials must be univariate in the same indeterminate.
```

Just as above, you can download the test file `pmulTest.input` from the course web page and you should test your implementation further. Once again the test causing an error has been placed last.

**Note:** In all cases the code is very simple, focus on how Axiom does things and work with it not against it. A common error is to ignore the type system, either in the use of variables or the production of results, this leads to frustration and contorted code.

**Exercise 4.1** *Let $p = x^2 + x + 1$ so that $\mathbb{Z}_2[x]/(p)$ has four elements: 0, 1, $x$, $1 + x$ (to be more accurate the four elements are the equivalence classes represented by the given expressions). Write the addition and multiplication tables of this ring. That is, fill in the tables:* [*4 marks* ]

| $+$ | $0$ | $1$ | $x$ | $1+x$ |
|---|---|---|---|---|
| $0$ | $0$ | $1$ | $x$ | $1+x$ |
| $1$ | | | | |
| $x$ | | | | |
| $1+x$ | | | | |

| $\times$ | $0$ | $1$ | $x$ | $1+x$ |
|---|---|---|---|---|
| $0$ | | | | |
| $1$ | | | | |
| $x$ | | | | |
| $1+x$ | | | | |

*Is $\mathbb{Z}_2[x]/(p)$ a field? Justify your answer briefly, you may assume that $\mathbb{Z}_2[x]/(p)$ is a commutative ring with identity (we will justify this in a general setting later on).*

As a goodwill gesture here is a nice way to produce the addition table (some Axiom output has been deleted):

```
(3) ->  p:=x^2+x+1
 p:=x^2+x+1
(3) ->
          2
   (3)  x  + x + 1
                                                   Type: Polynomial(Integer)
(4) ->  A:Matrix(UP(x,PrimeField 2)):=new(4,4,0)
 A:Matrix(UP(x,PrimeField(2))):=new(4,4,0)
(4) ->
                                 Type: Matrix(UnivariatePolynomial(x,PrimeField(2)))
(5) -> L:=[0,1,x,1+x]
L:=[0,1,x,1+x]
(5) ->
   (5)  [0,1,x,x + 1]
                                              Type: List(Polynomial(Integer))
(6) -> for i in 1..4 repeat for j in 1..4 repeat A(i,j):=padd(L.i,L.j,p,x,2)
for i in 1..4 repeat for j in 1..4 repeat A(i,j):=padd(L.i,L.j,p,x,2)
                                                                   Type: Void
(7) -> A
A
(7) ->
```

In your handin just give the tables, no need to show how you got them.

**§5. Some useful Axiom commands.** You can get a full description of commands starting with `)` from the X11 help facility via, e.g., `Browse>Commands`.

- `)quit`: Quit Axiom.
- `)hd`: Relaunch the X11 hyper document help system. Useful if you close it by mistake or it crashes.
- `)set`: Customises Axiom's behaviour in many ways. Issuing `)set` by itself will output a message showing the available topics on which further information can be obtained. For example, `)set output` will produce a list of the ways in which output can be modified. Thus `)set output tex on` will produce TeX output to commands, useful if you want to import the result to a document. Issuing `)set output tex off` will revert to the standard output format.

- `)clear`: This can be used to undo various things, see the online help for all the options. One useful option is `)clear all` which in effect starts a new session (but with the advantage that you can recall previous entries using the up arrow key). Another good use is to 'reset' a variable name, e.g., `)clear properties x` will forget any assignment or type associated with `x`.

- `)read`: Used to read in a file which is given as the next argument, e.g., `)read ex1` (note that it is assumed the file has the extension `.input`. The file name can include a directory path in Unix style.

- `%`: Returns the last result. Use `%%(n)` to return results of other steps. Here `n` is either a negative number (denoting how many steps to go back) or a positive one giving the actual step number as shown by Axiom. Thus `%%(-1)` is the same as just `%`.

  In general `%` is used to denote the start of a macro definition. Axiom has various ones built in, e.g., `%i` denotes the square root of $-1$, `%pi` denotes $\pi$ and `%infinity` denotes $\infty$.

- `_`: Indicates continuation of an input line but see the first note in §1 (the symbol is an underscore).

- `--`: Indicates the start of a comment which lasts till the end of the line (the symbol consists of two minus signs).

- `++`: Same as `--`, i.e., start of a comment.

- `Ctrl+C`: Interrupt the current computation and return to interactive session.

- Tab Key: Provides name completion for Axiom operations. Repeated pressing of the key gives the alternative completions. For example if the tab key is pressed after entering `groebn` the name is completed to `groebner`. Pressing tab repeatedly produces `groebner?`, `groebnerFactorize` etc.

**§6. Some useful Axiom constructor abbreviations.** The following are some abbreviations taken from p.68 of the Axiom documentation book (as you can see a few are not really abbreviations but are there for consistency).

| | |
|---|---|
| COMPLEX abbreviates Complex | DFLOAT abbreviates DoubleFloat |
| EXPR abbreviates Expression | FLOAT abbreviates Float |
| FRAC abbreviates Fraction | INT abbreviates Integer |
| MATRIX abbreviates Matrix | NNI abbreviates NonNegativeInteger |
| PI abbreviates PositiveInteger | POLY abbreviates Polynomial |
| STRING abbreviates String | UP abbreviates UnivariatePolynomial |
| PF abbreviates PrimeField | |

You can combine both full constructor names and abbreviations in a type expression, for example:

POLY INT is the same as Polynomial(INT)
POLY(Integer) is the same as Polynomial(Integer)
POLY(Integer) is the same as Polynomial(INT)
FRAC(COMPLEX(INT)) is the same as Fraction Complex Integer
FRAC(COMPLEX(INT)) is the same as FRAC(Complex Integer)

Kyriakos Kalorkoti, January 2018