Quelques lignes d'input dans Axiom

Daniel.Augot@inria.fr

Abstract

Ce documaine est un première présentation d'Axiom, prévue pour un novice. On montrera surtout le fonctionnement de l'interprête, en présentant d'abord quelques élements de langage (affectation, macros, boucles). Le notion principale en Axiom est celle de *type*, et de signature de fonction. L'accent est mis sur des exemples de calcul dans le corps finis, dont on présentera les différentes implantations.

Enfin, l'écriture de fichiers interprétés (`.input`), et de fichiers pour la compilation (`.spad`) est légérement abordée.

Contents

1 Usage de l'interprête

L'interprêtre suit une boucle READ-EVAL-PRINT classique : une expression est lue, évaluée, et le résultat de l'évaluation est retourné. La principale différence avec les systèmes usuels est la notion de typage fort. Il est en général plus important de définir les types que l'on doit manipuler, que les objets eux-mêmes. En Axiom, un type s'appelle *domaine*.

1.1 Affectation

1.1.1 Résolution de type de l'interprête

La plupart du temps, les règles mises dans Axiom lui permettent de déterminer un type "satisfaisant" à chaque entrée de l'utilisateur. Pour les corps finis ce n'est pas le cas, et il faut lui spécifier quel type on veut obtenir.

```
->x:=1

   (1)  2
                                                Type: PositiveInteger
```

Axiom affiche le résulat (`2`), le numéro du résultat (`(1)`), ce qui permet de le réutiliser plus tard dans la session, ainsi que le *type* du résultat (`PositiveInteger`). Dans la terminologie Axiom, de tels types sont des *domaines*, que l'on peut se représenter comme des ensembles mathématiques munis d'une structure et d'opérations.

On remarque aussi qu'il n'y a pas de séparateur de fin d'expression.

```
->x:=x+1

   (2)  3
                                                Type: PositiveInteger
->x+y+z

   (3)  z + y + 3
                                              Type: Polynomial Integer
->pol:=%

   (4)  z + y + 3
                                              Type: Polynomial Integer
```

Le type du résultat peut être présenté en abrégé, par le commande `)set output abbreviate on`. De telle commandes, dont le premier caractère est `)`, sont des *commandes systèmes*, qui permettent de controler l'environnement de travail de l'interprête. Par exemple, pour quitter l'interprête, il faut saisir le commande `)quit`.

```
->)set output abbreviate on
->pol

   (5)  z + y + 3
                                                    Type: POLY INT
```

Le mécanisme d'affectation est le suivant :

1. Evaluation de la partie droite de l'affectation. Celà implique calcul de la valeur, détermination de son type.
2. Un variable est créée, portant le nom donné par la partie droite de l'affectation, et le type résolu par l'interprête.
3. La valeur de cette variable est le résultat de l'évaluation du terme de gauche.

Une variable c'est donc trois choses : un nom, un type, une valeur (éventuellement). Il vaut mieux parler des *indéterminées* d'un polynôme ou d'une expression, mais pas de ses variables. Pour rejoindre la présentation faite dans Maple, il n'y a dans Axiom que des variables informatiques.
Remarque importante

On ne peut pas calculer avec une variable qui n'a pas de valeur

Une conséquence d'un tel mécanisme d'évaluation c'est qu'il n'y a pas de dépendance possible de variables entre elles.

Comparons les deux sessions Axiom et Maple suivantes :

Maple :

```
> x:=a*b;

                                    x := a b
```

```
> a:=2;

                                        a := 2

> x;

                                         2 b

> a:='a';

                                        a := a

> x;

                                         a b
```

Axiom :

```
->)clear all
   All user variables and function definitions have been cleared.

->x:=a*b

   (1)  a b
                                                        Type: POLY INT
->a:=1

   (2)  1
                                                              Type: PI
->x

   (3)  a b
                                                        Type: POLY INT
```

Le *variable* `a`, qui vaut `1`, est un objet du type `PositiveInteger`, sans rapport avec l'*indéterminée* `a`, dans le polynôme $x = ab$, qui est une indéterminée au sens mathématique du terme. Pour obtenir le résultat désiré, il faut donc *évaluer* le polynôme en la variable `a`, pour la valeur 1.

```
->eval(x,a=1)
```

```
>> Error detected within library code:
failed cannot be coerced to mode (Symbol)
You are being returned to the top level of the interpreter.
```

C'est bien normal : `a` est un entier positif, qui ne peut être transformé en symbole. La macro "'" (comme en Lisp) permet de bloquer le mécanisme d'évaluation, pour obtenir l'objet non évalué.

```
->eval(x,'a=1)

   (4)  b
                                                    Type: POLY INT
```

Il existe en Axiom le mécanisme des `macro`, qui permet de maintenir des dépendances entre symboles.

1.1.2 Objets "mutables"

Cette notion est la notion le plus "informatique" d'Axiom(au niveau de l'utilisateur). C'est essentiellement l'idée de pointeur qui est reproduite ainsi, et conserne essentiellement les structures qui collectent des objets (listes, tableaux, matrices...).

On forme des listes avec la fonction `construct` qui syntaxiquement est équivalent à `[` arguments `]`.

```
->construct(1,2,3)

   (56)  [1,2,3]
                                              Type: LIST PI
->l:=[1,2,3]

   (1)  [1,2,3]
                                                    Type: LIST PI
```

On accéde aux él'ements d'une liste avec la fonction `elt` qui syntaxiquement est équivalente aux formes suivantes :

```
->elt(l,1)

   (2)  1
                                                         Type: PI
```

```
->l.1

   (3)  1
                                                            Type: PI
->l(1)

   (4)  1
                                                            Type: PI
->l 1

   (5)  1
                                                            Type: PI
```

On change les éléments d'une liste au moyen de la fonction `setelt`, qui peut prendre les formes suivantes :

```
->setelt(l,1,11111)

   (58)  11111
                                                            Type: PI
->l

   (61)  [11111,2,3]
                                                       Type: LIST PI
->l.1:=33

   (62)  33
                                                            Type: PI
->l(1) :=22

   (63)  22
                                                            Type: PI
->l 1 := 1  --mais ou sont les syntaxes d'antan ?

   (64)  1
                                                            Type: PI
->l

   (65)  [1,2,3]
                                                       Type: LIST PI
```

Les listes sont en fait des références à des collection d'objets, ce qui explique le fonctionnement suivant :

```
->l2:=l

   (6)  [1,2,3]
                                                        Type: LIST PI

->l2.1:=11111

   (7)  11111
                                                             Type: PI
->l

   (8)  [11111,2,3]
                                                        Type: LIST PI
```

La fonction `copy` permet d'avoir une liste égale à la première mais qui fait référence à un autre objet :

```
->l1:=copy l

   (9)  [11111,2,3]
                                                        Type: LIST PI
->l1.1:=1

   (12)  1
                                                             Type: PI
->l

   (13)  [11111,2,3]
                                                        Type: LIST PI
```

En gros, sont mutables les objets des types qui exportent la fonction `setelt`, i.e. tous les types qui sont des aggrégats d'objets. Les domaines: LIST, Array, Matrix, SquareMatrix, Table, Vector... sont des somaines dont les objets sont mutables. Qu'on se rassure, un domaine dont les objets sont mutables exporte en général la fonction `copy`.

Celà permet de faire des fonctions à effet de bord, car une fonction ne reçoit qu'une copie de l'argument réellement passé (comme en C).

1.2 Boucles

1.2.1 `repeat`

Le mot clé est `repeat` :

```
->i:=0

   (1)  0
                                                            Type: NNI
->repeat i:=i+1
```

(Pour interrompre CTRL c)

```
;;Interrupt delayed.

   >> System error:
   Console interrupt.
   You are being returned to the top level of the interpreter.

->i

   (2)  567
                                                            Type: NNI
->i:=0; while i<100 repeat i:=i+1
                                                           Type: VOID
->i

   (4)  100
                                                            Type: NNI
->a:=1

   (5)  1
                                                             Type: PI
->for i in 1..10 repeat a:=a*i
                                                           Type: VOID
->a

   (7)  3628800
```

L'indice d'itération est une variable liée dans le boucle. Ceci donne le comportement suivant :

```
->myGlobalVariable:=2**(2**11)-1
   (3)
  32317006071311007300714876688669951960444102669715484032130
  34542752465513886789089319720141152291346368871796092189801
  94941195591504909210950881523864482831206308773673009960917
  50197750389652106796057638384067568276792218642619756161838
  09433847617047058164585203630504288757589154106580860755239
  91239303855219143333896683424206849747865645694948561760353
  26322058077805659331026192708460314150258592864177116725943
  60371846185735759835115230164590440369761323328723122712568
  47108202097251571017269313234696785425806566979350459972683
  52998638215525166389437335543602135433229604645318478604952
  14819355585361105959623065 5
                                                        Type: PI

->for myGlobalVariable in 1..10 repeat print myGlobalVariable
   1
   2
   3
   4
   5
   6
   7
   8
   9
   10
                                                              Type: Void
->myGlobalVariable

   (5)
  32317006071311007300714876688669951960444102669715484032130
  34542752465513886789089319720141152291346368871796092189801
  94941195591504909210950881523864482831206308773673009960917
  50197750389652106796057638384067568276792218642619756161838
  09433847617047058164585203630504288757589154106580860755239
  91239303855219143333896683424206849747865645694948561760353
  26322058077805659331026192708460314150258592864177116725943
  60371846185735759835115230164590440369761323328723122712568
  47108202097251571017269313234696785425806566979350459972683
```

```
  529986382155251663894373355436021354332296046453184786049521
  481935558536110595962306551
                                                          Type: PI
```

Autre commodité :Il y a une fonction “tel que” qui permet d’attacher un prédicat à la boucle `for`

```
->for i in 1..30 | prime?(i) repeat print(2**(2**i)-1)

   15
   255
   4294967295
   340282366920938463463374607431768211455
  323170060713110073007148766886699519604441026697154840321301
  345427524655138867890893197201411522913463688717960921898019
  949411955915049092109508815238644828312063087736730099609179
  501977503896521067960576383840675682767922186426197561618380
  943384761704705816458520363050428875758915410658086075523919
  912393038552191433338966834242068497478656456949485617603531
  263220580778056593310261927084603141502585928641771167259431
  603718461857357598351152301645904403697613233287231227125681
  471082020972515710172693132346967854258065669793504599726831
  529986382155251663894373355436021354332296046453184786049521
  481935558536110595962306551
.
.
.
```

1.2.2 Itération sur une liste

Une commodité du langage est de pouvoir itérer sur les objets d’une liste, quels qu’ils soient. De plus on peut créer une liste, en combinant `construct` (i.e. `[some stuff]`) et `for`.

```
->l:=[i for i in 1..10 | prime? i]

   (1)  [2,3,5,7]
                                                    Type: LIST PI
->lp:=[ (x**i-1)/(x-1) for i in l]

               2          4    3    2          6    5    4    3    2
```

```
   (2)  [x + 1,x  + x + 1,x  + x  + x  + x + 1,x  + x  + x  + x  + x  + x + 1]
                                                 Type: LIST FRAC POLY INT
->lp1:LIST POLY INT:=lp

                2          4    3    2          6    5    4    3    2
   (3)  [x + 1,x  + x + 1,x  + x  + x  + x + 1,x  + x  + x  + x  + x  + x + 1]
                                                   Type: LIST POLY INT
->lpf:=[factor pol for pol in lp1]

                2          4    3    2          6    5    4    3    2
   (4)  [x + 1,x  + x + 1,x  + x  + x  + x + 1,x  + x  + x  + x  + x  + x + 1]
                                                  Type: LIST FR POLY INT
```

Il est aussi possible d'utiliser la fonction `map` : `map(func,list)` plutot que `[f(x) for x in l]`.

```
->lpf:=map(factor,lp1)

                2          4    3    2          6    5    4    3    2
   (5)  [x + 1,x  + x + 1,x  + x  + x  + x + 1,x  + x  + x  + x  + x  + x + 1]
                                                  Type: LIST FR POLY INT
```

Comme on peut faire des listes d'objets de toute nature, on peut vraiment itérer sur n'importe quoi :

```
->l:=[x+->x**2,x+->x**3,x+->x**4]

                     2            3            4
   (46)  [##1 +-> ##1 ,##1 +-> ##1 ,##1 +-> ##1 ]
                                                 Type: LIST ANON
->(l.1)(4)

   (47)  16
                                                          Type: PI
->l.2.4

   (48)  64
                                                          Type: PI
->l.3.4

   (49)  256
```

```
->l1:= l :: LIST (INT -> INT);
                                              Type: LIST (INT -> INT)
->[ func(19) for func in l1]

   (56)  [361,6859,130321]
                                                        Type: LIST INT
```

En général il est préférable d'utiliser `map` plutot que des boucles (implicites ou non). Il faut se rappeller que Axiom est bâti sur Akcl (Austin-Kyoto Common Lisp), et que `map` est à la Lisp. De même il y a une fonction `reduce` :

```
->p12:POLY INT:=1; for pol in lp repeat p12:=p12*pol;p12

   (6)
      13     12     11      10      9      8      7      6      5      4      3
     x   + 4x   + 9x   + 15x   + 21x  + 26x  + 29x  + 29x  + 26x  + 21x  + 15x
   +
       2
     9x  + 4x + 1
                                                        Type: POLY INT
->reduce(*,lp)

   (7)
      13     12     11      10      9      8      7      6      5      4      3
     x   + 4x   + 9x   + 15x   + 21x  + 26x  + 29x  + 29x  + 26x  + 21x  + 15x
   +
       2
     9x  + 4x + 1
                                                   Type: FRAC POLY INT
->reduce(*,[i for i in 1..10])

   (10)  3628800
                                                             Type: PI
```

Bien que le langage presente certains attraits pour l'écriture, il faut toutefois s'en garder, car certaines formes concises peuvent être coûteuses. Une belle écriture avec des boucles est beaucoup moins efficace que `map` ou `reduce`.

Note Les listes constituées d'objets de tout type, et ce sont des listes homogènes.

```
(1) ->l:=[ [1,2,3],x,2]

   (1)  [[1,2,3],x,2]
                                                        Type: List Any
```

Afin d'homogénéiser les objets, l'interprête les considère du type `Any`. Tout objet Axiom peut être tranformé en objet de type `Any`.

1.3 Comment obtenir une variable du type désiré

1.3.1 Déclaration d'une variable

La syntaxe est la même qu'en Pascal.

```
->x:INT
                                                        Type: VOID
->x:=1

   (2)  1
                                                         Type: INT
```

C'est la déclaration qui a guidé l'interprête dans da résolution de type. Autre exemple, avec F_2 (constructeur PF) :

```
->x2:PF(2)
                                                        Type: VOID
->x2:=1

   (4)  1
                                                        Type: PF 2
->2*x2

   (5)  0
                                                        Type: PF 2
->x2:=6

   (6)  0
                                                        Type: PF 2
```

Une fois le type déterminé, tous les calculs se sont d'une manière cohérente, i.e. le résultat est bien un élément de F_2, et de plus le calcul a été effectué d'une manière cohérente.

De même avec les polynômes :

```
->p2:POLY PF 2:=y+z+1

   (8)  z + y + 1
                                                        Type: POLY PF 2
```

L'interprête a travaillé pour nous, mais c'était facile pour lui. On va lui demander plus :

```
->pol:=z+y+1

   (11)  z + y + 1
                                                         Type: POLY INT
->p2+pol

   (13)  0
                                                        Type: POLY PF 2
```

Le polynôme p2 est un polynôme à coefficients dans F_2, qui se comporte correctement :

```
->2*p2

   (9)  0
                                                        Type: POLY PF 2
->p2+1

   (10)  z + y
                                                        Type: POLY PF 2
->p2**2

          2    2
   (11)  z  + y  + 1
                                                        Type: POLY PF 2
->factor %

                    2
   (12)  (z + y + 1)
                                                    Type: FR POLY PF 2
->p0:POLY INT:=p2**2

          2    2
```

```
   (14)  z  + y  + 1
                                                        Type: POLY INT
->factor(2::COMPLEX INT)

                       2
   (57)  - %i (1 + %i)
                                                    Type: FR COMPLEX INT
                            Time: 0.15 (IN) + 0.09 (EV) + 0.12 (OT) = 0.36 sec
->p0:=x**2-2*x+5

          2
   (54)  x  - 2x + 5
                                                         Type: POLY INT
                            Time: 1.04 (IN) + 0.02 (EV) + 0.83 (OT) = 1.89 sec
->factor p0

          2
   (55)  x  - 2x + 5
                                                      Type: FR POLY INT
                            Time: 0.25 (IN) + 0.23 (EV) + 0.17 (OT) = 0.65 sec
->factor(p0::POLY COMPLEX INT)

   (56)  (x - 1 - 2%i)(x - 1 + 2%i)
                                                 Type: FR POLY COMPLEX INT
                            Time: 0.37 (IN) + 1.15 (EV) + 0.31 (OT) = 1.83 sec
```

On voit que l'interprête a relevé un polynômemodulo 2 en un polynômesur les entiers. Evidemment ce polynômen'a pas la même factorisation :

```
->factor p0

          2    2
   (15)  z  + y  + 1
                                                      Type: FR POLY INT
```

Il n'y a donc pas de `flag` à la Macsyma, c'est le type de l'objet qui détermine la fonction `factor`. En fait ce n'est pas la même `factor` qui est exécuté, bien qu'elles portent le même nom. Autre fonction `factor` :

```
->factor(factorial(6)+1)
```

```
   (16)  7 103
                                                        Type: FR INT
```

Ceci nous améne à la notion de signature d'une fonction : de même qu'une variable une fonction a un type, on peut voir celui-ci comme la signature de la fonction, i.e. la déclaration des ses arguments. Une signature se déclare de la manière suivante :

```
->foo : INT -> INT
                                                        Type: VOID
->foo i == i-1
                                                        Type: VOID
->foo 9
   Compiling function factor with type INT -> INT

   (22)  8

->foo 10

   (22)  10
                                                        Type: PI
```

Là où l'interprête est souple (pour une fois!), c'est qu'il est possible de donner le corps d'une fonction, sans lui donner de signature :

```
->bar i == i**2
                                                        Type: Void
```

Ce n'est qu'à l'appel de la fonction avec des arguments donnés que la function va être compilée, avec des signatures différentes :

```
->bar 2
Compiling function bar with type PI ->PI

   (50)  4
                                                        Type: PI
->bar(x+1)
   Compiling function bar with type POLY INT -> POLY
   INT

          2
   (51)  x  + 2x + 1
```

```
                                                          Type: POLY INT
->bar(3/4)
   Compiling function bar with type FRAC INT -> FRAC INT

         9
   (52)  --
         16
                                                          Type: FRAC INT
```

On pourrait croire que la fonction `bar` a été à chauqe fois recompilée. Ce n'est pas vraiment ce qu'il se passe. A chaque fois Axiom a compilé une nouvelle fonction `bar`, avec une signature différente.

```
->bar 2

   (53)  4
                                                          Type: PI
```

Axiom n'a pas a eu à recompiler `bar` de nouveau, il y a vraiment trois fonctions `bar` pour l'interprête. On dispose ainsi d'une batterie de fonction `bar`, qui font la même chose sur des types différents, toutes compilées.

1.3.2 "Package Calling"

On peut spécifier précisément quelle fonction l'inreprête doit employer, au moyen du symbole $.

```
->K:=FF(2,4)

   (1)  FF(2,4)
                                                          Type: DOMAIN
->a:=random()$K

           3     2
   (2)  %C  + %C  + 1
                                                  Type: FF(2,4)
->a**16-a

   (4)  0
                                                  Type: FF(2,4)
```

Cela semble juste... Voilà d'autres fonctions permettant d'obtenir des éléments de K.

```
->b:=index(2)$K

   (12)  %C
                                                    Type: FF(2,4)
->b:=index(3)$K

   (13)  %C + 1
                                                    Type: FF(2,4)
->b:=index(9)$K

           3
   (14)  %C  + 1
                                                    Type: FF(2,4)
->c:=generator()$K

   (15)  %C
                                                    Type: FF(2,4)
->primitiveElement()$K

   (16)  %C
                                                    Type: FF(2,4)
->K1:=FF(2,5)

   (18)  FF(2,5)
                                                   Type: DOMAIN
->primitiveElement()$K1

   (19)  %D
                                                    Type: FF(2,5)
->K3:=FF(2,6)

   (20)  FF(2,6)
                                                   Type: Domain
->primitiveElement()$K3

   (21)  %E
                                                    Type: FF(2,6)
->order %

   (22)  63
```

```
                                                    Type: PI
```

On peut aussi spécifier le type désiré du résultat, au moyen de `@`

```
->generator()@K

   (32)  %C
                                                  Type: FF(2,4)
->generator()@K1

   (33)  %D
                                                  Type: FF(2,5)
```

Le symbole `$` spécifie explicitement quelle fonction utiliser, alors que `@` indique comment effectuer la résolution de type, comme lorque l'on déclare une variable. Tant que l'on déclare les variables, je ne vois pas bien la différence entre `@` et `:`, sauf dans expressions intermédiaires, pour guider l'interprête.

2 Corps finis

2.1 Qu'y a t-il sur les corps finis ? Autodocumentation

Commandes `)what`, `)show`, `)display` Quels sont les domaines qui contiennent la chaîne "finite" :

```
->)what domain finite
------------------------------- Domains --------------------------------

Domains with names matching patterns:
     finite

 FAMR-    FiniteAbelianMonoidRing&
 FAXF-    FiniteAlgebraicExtensionField&
 FDIV     FiniteDivisor
 FF       FiniteField
 FFCG     FiniteFieldCyclicGroup
 FFCGP    FiniteFieldCyclicGroupExtensionByPolynomial
 FFCGX    FiniteFieldCyclicGroupExtension
 FFIELDC- FiniteFieldCategory&
 FFNB     FiniteFieldNormalBasis
 FFNBP    FiniteFieldNormalBasisExtensionByPolynomial
 FFNBX    FiniteFieldNormalBasisExtension
 FFP      FiniteFieldExtensionByPolynomial
 FFX      FiniteFieldExtension
 FINAALG- FiniteRankNonAssociativeAlgebra&
 FINPF-   FinitePrimeField&
 FINRALG- FiniteRankAlgebra&
 FLAGG-   FiniteLinearAggregate&
 FSAGG-   FiniteSetAggregate&
 IFF      InnerFiniteField
 ITUPLE   InfiniteTuple
```

Autres formes pour cette commande système :

```
->)w do finite
->)wh do finite
```

La commande `)what` permet d'interroger sur d'autres objets, en particulier les fonctions :

```
->)what operation number

Operations whose names satisfy the above pattern(s):

numberOfComponents            numberOfComposites
numberOfComputedEntries       numberOfCycles
numberOfDivisors              numberOfFactors
numberOfFractionalTerms       numberOfHues
numberOfImproperPartitions    numberOfIrreduciblePoly
numberOfMonomials             numberOfNormalPoly
numberOfPrimitivePoly

   To get more information about an operation, say numberOfComponents ,
   issue the command )display op numberOfComponents
```

Pour se renseigner sur une fonction :

```
->)d op numberOfNormalPoly

There is one unexposed function called numberOfNormalPoly :
   [1] PositiveInteger -> PositiveInteger from
         FiniteFieldPolynomialPackage D2
         if D2 has FFIELDC

->)show FiniteFieldPolynomialPackage
 FFPOLY GF: FFIELDC  is a package constructor
 Abbreviation for FiniteFieldPolynomialPackage is FFPOLY
 This constructor is not exposed in this frame.
 Issue )edit ffpoly.spad to see algebra source code for FFPOLY

------------------------------ Operations ------------------------------
 createNormalPoly : PI -> SUP GF       createPrimitivePoly : PI -> SUP GF
 normal? : SUP GF -> BOOLEAN           numberOfIrreduciblePoly : PI -> PI
 numberOfNormalPoly : PI -> PI         numberOfPrimitivePoly : PI -> PI
 primitive? : SUP GF -> BOOLEAN        random : PI -> SUP GF
 random : (PI,PI) -> SUP GF
 createIrreduciblePoly : PI -> SUP GF
 createNormalPrimitivePoly : PI -> SUP GF
 createPrimitiveNormalPoly : PI -> SUP GF
 leastAffineMultiple : SUP GF -> SUP GF
 nextIrreduciblePoly : SUP GF -> Union(SUP GF,"failed")
```

```
 nextNormalPoly : SUP GF -> Union(SUP GF,"failed")
 nextNormalPrimitivePoly : SUP GF -> Union(SUP GF,"failed")
 nextPrimitiveNormalPoly : SUP GF -> Union(SUP GF,"failed")
 nextPrimitivePoly : SUP GF -> Union(SUP GF,"failed")
 reducedQPowers : SUP GF -> PRIMARR SUP GF
```

Le mieux est toutefois d'avoir recours à l'Hyperdoc, qui permet d'avoir une documentation commentée des functions d'un domaine.

2.2 Problème de l'input, ou du lecteur

On a vu qu'il existait un domaine `FF` : qu'exporte-t'il?

```
->K:=FF(3,4)

   (1)  FF(3,4)
                                                                Type: DOMAIN
->)sh K
 FF(3,4) is a domain constructor.
 Abbreviation for FiniteField is FF
 This constructor is exposed in this frame.
 Issue )edit ffp.spad to see algebra source code for FF

------------------------------ Operations ------------------------------

 ?*? : ($,$) -> $                      ?*? : ($,FRAC INT) -> $
 ?*? : ($,PF 3) -> $                   ?*? : (FRAC INT,$) -> $
 ?*? : (INT,$) -> $                    ?*? : (PF 3,$) -> $
 ?**? : ($,INT) -> $                   ?+? : ($,$) -> $
 -? : $ -> $                           ?-? : ($,$) -> $
 ?/? : ($,$) -> $                      ?/? : ($,PF 3) -> $
 ?=? : ($,$) -> BOOLEAN                Frobenius : $ -> $
 Frobenius : ($,NNI) -> $              1 : () -> $
 0 : () -> $                           algebraic? : $ -> BOOLEAN
 associates? : ($,$) -> BOOLEAN        basis : () -> VECTOR $
 basis : PI -> VECTOR $                characteristic : () -> NNI
 charthRoot : $ -> $                   coerce : $ -> $
 coerce : FRAC INT -> $                coerce : INT -> $
 coerce : PF 3 -> $                    coerce : $ -> OUTFORM
 coordinates : $ -> VECTOR PF 3        createNormalElement : () -> $
 createPrimitiveElement : () -> $      degree : $ -> ONECOMP PI
```

```
degree : $ -> PI                        dimension : () -> CARD
discreteLog : $ -> NNI                  euclideanSize : $ -> NNI
extensionDegree : () -> ONECOMP PI      extensionDegree : () -> PI
factor : $ -> FR $                      gcd : ($,$) -> $
gcd : LIST $ -> $                       generator : () -> $
inGroundField? : $ -> BOOLEAN           index : PI -> $
init : () -> $                          inv : $ -> $
lcm : ($,$) -> $                        lcm : LIST $ -> $
lookup : $ -> PI                        minimalPolynomial : $ -> SUP PF 3
nextItem : $ -> Union($,"failed")       norm : ($,PI) -> $
norm : $ -> PF 3                        normal? : $ -> BOOLEAN
normalElement : () -> $                 one? : $ -> BOOLEAN
order : $ -> ONECOMP PI                 order : $ -> PI
prime? : $ -> BOOLEAN                   primeFrobenius : $ -> $
primeFrobenius : ($,NNI) -> $           primitive? : $ -> BOOLEAN
primitiveElement : () -> $              ?quo? : ($,$) -> $
random : () -> $                        recip : $ -> Union($,"failed")
?rem? : ($,$) -> $                      represents : VECTOR PF 3 -> $
retract : $ -> PF 3                     size : () -> NNI
sizeLess? : ($,$) -> BOOLEAN            squareFree : $ -> FR $
squareFreePart : $ -> $                 trace : ($,PI) -> $
trace : $ -> PF 3                       transcendenceDegree : () -> NNI
transcendent? : $ -> BOOLEAN            unit? : $ -> BOOLEAN
unitCanonical : $ -> $                  zero? : $ -> BOOLEAN
charthRoot : $ -> Union($,"failed")
conditionP : MATRIX $ -> Union(VECTOR $,"failed")
coordinates : VECTOR $ -> MATRIX PF 3
definingPolynomial : () -> SUP PF 3
discreteLog : ($,$) -> Union(NNI,"failed")
divide : ($,$) -> Record(quotient: $,remainder: $)
expressIdealMember : (LIST $,$) -> Union(LIST $,"failed")
?exquo? : ($,$) -> Union($,"failed")
extendedEuclidean : ($,$) -> Record(coef1: $,coef2: $,generator: $)
extendedEuclidean : ($,$,$) -> Union(Record(coef1: $,coef2: $),"failed")
factorsOfCyclicGroupSize : () -> LIST Record(factor: INT,exponent: INT)
minimalPolynomial : ($,PI) -> SUP $
multiEuclidean : (LIST $,$) -> Union(LIST $,"failed")
principalIdeal : LIST $ -> Record(coef: LIST $,generator: $)
representationType : () -> Union("prime",polynomial,normal,cyclic)
retractIfCan : $ -> Union(PF 3,"failed")
```

```
 tableForDiscreteLogarithm : INT -> TABLE(PI,NNI)
 unitNormal : $ -> Record(unit: $,canonical: $,associate: $)

->a:K:=random()

   (2)  2%C
                                                                Type: FF(3,4)
->a**4

   (3)  2%C + 1
                                                                Type: FF(3,4)
->a**80-1

   (5)  0
                                                                Type: FF(3,4)
->a:=generator()

   (6)  %C
->p:=x**2+x+a

         2         3     2
   (5)  x  + x + %C  + %C  + 1
                            Type: Polynomial FiniteField(2,4)
->factor p

         2         3     2
   (6)  x  + x + %C  + %C  + 1
                   Type: Factored Polynomial FiniteField(2,4)

->p:=p+a

         2
   (7)  x  + x
                            Type: Polynomial FiniteField(2,4)
->p:=p+1

         2
   (8)  x  + x + 1
                            Type: Polynomial FiniteField(2,4)
->factor p
```

```
                2             2
   (9)  (x + %C  + %C)(x + %C  + %C + 1)
                                  Type: Factored Polynomial FiniteField(2,4)
->p0:POLY PF 2:=p

          2
   (10)  x  + x + 1
                                       Type: Polynomial PrimeField 2
->factor p0

          2
   (11)  x  + x + 1
                           Type: Factored Polynomial PrimeField 2
                                                                  Type: FF(3,4)
```

3 Différentes représentations sous la main

3.1 Groupe Cyclique

On peut indiquer à Axiom de construire un corps fini en utilisant en interne la représeantion multiplicative des éléments non nuls d'un corps fini, sous forme de puissances d'un él'ement primitif.

Pour l'utilisateur, les mêmes fonctions sont disponibles, ce qui est dû au fait que ce constructeur est dans la même catégorie que PF. Reprenons l'exemple de F_{3^4} :

```
->)wh do cyclic
-------------------------------- Domains --------------------------------

Domains with names matching patterns:
     cyclic

 FFCG     FiniteFieldCyclicGroup
 FFCGP    FiniteFieldCyclicGroupExtensionByPolynomial
 FFCGX    FiniteFieldCyclicGroupExtension
->K:=FFCG(3,4)

   (1)  FFCG(3,4)
                                                                  Type: DOMAIN
```

```
 ->a:=primitiveElement()$K

          1
   (2)  %C
                                                          Type: FFCG(3,4)
->a**30

          30
   (3)  %C
                                                          Type: FFCG(3,4)

 ->a**10+a

          78
   (4)  %C
                                                          Type: FFCG(3,4)
```

Comme on peut le voir tous les objets sont représentés comme puissances de l'élément gén'erateur, et affichés sous cette forme, même après un calcul additif.

```
->p:POLY PF 3:=x**2+x+2

         2
   (5)  x  + x + 2
                                                          Type: POLY PF 3
->factor p

         2
   (6)  x  + x + 2
                                                       Type: FR POLY PF 3

->p::POLY K

         2         40
   (7)  x  + x + %C
                                                     Type: POLY FFCG(3,4)
->factor %

               50        70
```

```
   (8)  (x + %C  )(x + %C  )
                                                    Type: FR POLY FFCG(3,4)
->(a**50)**3

          70
   (9)  %C
                                                          Type: FFCG(3,4)
```

Ainsi donc, il n'y a pas de différences "externes" entre FFCG, et FF, car les deux domaines appartiennent à la même catégorie. On peut exactement faire les "mêmes choses" avec les deux constructeurs. Celà permet d'écrire les algorithmes dans toute leur généralité, et de tester pour quelle représentation ils sont les plus efficaces.

3.2 Base Normale

Axiom fournit aussi un constructeur pour les corps finis qui utilise la représentation sous forme de décomposition en base normale.

```
->K1:=FFNB(3,4)

   (10)  FFNB(3,4)
                                                          Type: DOMAIN
->a1:=generator()$K1

   (11)  %D

->a1**3

            q
   (12)  %D
                                                          Type: FFNB(3,4)
->%+a1

            q
   (13)  %D  + %D
                                                          Type: FFNB(3,4)
->a1**4

             3      2
            q      q       q
```

```
   (14)  %D   + %D   + 2%D  + %D
                                                          Type: FFNB(3,4)
->%+a1

            3      2
           q      q       q
   (15)  %D   + %D   + 2%D  + 2%D
                                                          Type: FFNB(3,4)
->p

          2
   (17)  x  + x + 2
                                                          Type: POLY PF 3
->p:: POLY K1

                      3       2
         2           q       q       q
   (18)  x  + x + 2%D   + 2%D   + 2%D  + 2%D
                                                  Type: POLY FFNB(3,4)
->--rien d'etonnant :
->2::K1

            3       2
           q       q       q
   (19)  2%D   + 2%D   + 2%D  + 2%D
                                                          Type: FFNB(3,4)
->factor(%%(-2))

                2              3
               q              q      q
   (20)  (x + %D   + %D)(x + %D   + %D )
                                                Type: FR POLY FFNB(3,4)
```

4 Extension de corps finis

Axiom fournit trois constructeurs, qui permettent de définir une extension d'un corps fini, chaque constructeur fournissant une des trois représentations définies plus haut.

4.1 Extension usuelle

```
->k:=PF 3

   (21)  PF 3
                                                                Type: DOMAIN
->K:=FFX(k,4)

   (22)  FFX(PF 3,4)
                                                                Type: DOMAIN
->p:=x**3+x**2+x+2

          3    2
   (23)  x  + x  + x + 2
                                                           Type: POLY PF 3
->factor(p)

          3    2
   (24)  x  + x  + x + 2
                                                        Type: FR POLY PF 3
->factor(p::POLY K)

          3    2
   (25)  x  + x  + x + 2
                                                  Type: FR POLY FFX(PF 3,4)

->a2:=random()$K1

            3     2            2      3                  3
   (27)  (%F  + %F  + %F + 1)%G  + (%F  + %F + 1)%G + %F  + 2
                                                  Type: FFX(FFX(PF 3,4),3)
->minimalPolynomial(a2)

   (28)
    3       3      2            2       3      2                  3
   ?  + (2%F  + 2%F  + 2%F + 2)?  + (2%F  + 2%F  + 2%F + 1)? + 2%F  + %F + 2
                                                       Type: SUP FFX(PF 3,4)
->% a2
```

```
   (29)  0
                                                        Type: FFX(FFX(PF 3,4),3)
->p::POLY K1

          3    2
   (30)  x  + x  + x + 2
                                                  Type: POLY FFX(FFX(PF 3,4),3)
->factor %

   (31)
             2     2
     (x + %G  + %F %G + 2%F + 1)
  *
             2      3      2              3     2
     (x + %G  + (%F  + 2%F  + 2%F)%G + %F  + %F  + 2%F + 2)
  *
             2       3              3      2
     (x + %G  + (2%F  + %F)%G + 2%F  + 2%F  + 2%F + 1)
                                           Type: FR POLY FFX(FFX(PF 3,4),3)
->a2:=generator()$K1

   (32)  %G
                                                        Type: FFX(FFX(PF 3,4),3)
->a2**2

           2
   (35)  %G
                                                        Type: FFX(FFX(PF 3,4),3)
->a2**3

   (36)  2%G + 2%F
                                                        Type: FFX(FFX(PF 3,4),3)
->definingPolynomial()$K1

          3
   (37)  ?  + ? + %F
                                                           Type: SUP FFX(PF 3,4)
->% a2

   (40)  0
```

```
                                                    Type: FFX(FFX(PF 3,4),3)
```

Tout ce passe comme on pourrait le desirer, du point de vue du calcul en tout cas. Il est dommage que l'on ne puissse pas saisir plus facilement des éléments d'un corps finis, mais en est il vraiment besoin ?

4.2 Autres types d'extension

Il est possible de construire une extension d'un corps, en précisant quelle doit être la représentation :

4.2.1 Extension implantée comme groupe cyclique

```
->K

   (41)  FFX(PF 3,4)
                                                              Type: DOMAIN
->K1:=FFCGX(K,3)

   (42)  FFCGX(FFX(PF 3,4),3)
                                                              Type: DOMAIN
->a3:=generator()$K1

           1
   (44)  %H
                                                 Type: FFCGX(FFX(PF 3,4),3)
->(3**4)**3

   (46)  531441
                                                                   Type: PI
->a3**78574

           78574
   (48)  %H
                                                 Type: FFCGX(FFX(PF 3,4),3)
 ->%+a3

           260144
   (49)  %H
                                                 Type: FFCGX(FFX(PF 3,4),3)
```

4.2.2 Extension implantée en utilisant une base normale

```
->K2:=FFNBX(K,3)

   (52)  FFNBX(FFX(PF 3,4),3)
                                                                Type: DOMAIN
->index(3**4)$K2

           q
   (53)  %I
                                                  Type: FFNBX(FFX(PF 3,4),3)
->minimalPolynomial(%)

          3     2
   (54)  ?  + 2?  + 1
                                                       Type: SUP FFX(PF 3,4)
->random()$K2

   (55)
                              2
       3      2             q         3      2       q
   (2%F  + 2%F  + %F + 1)%I    + (2%F  + 2%F  + 1)%I  +

       3      2
   (2%F  + 2%F  + 2%F + 2)%I
                    Type: FFNBX(FFX(PF 3,4),3)
->%**(3**4)

   (56)
                         2
       3      2        q         3      2              q
   (2%F  + 2%F  + 1)%I    + (2%F  + 2%F  + 2%F + 2)%I  +

       3      2
   (2%F  + 2%F  + %F + 1)%I
                       Type: FFNBX(FFX(PF 3,4),3)
```

5 Extensions définies par l'utilisateur

Dans les exemples précédents, l'utilisateur n'avait pas le choix du polynômeirréductible choisi pour définir l'extension. Tous les constructeurs précédemment définis existent en donnant la possibilité à l'utilisateur de définir lui-même ce polynôme.

On ne donnera que l'exemple pour une simple extension.

```
 ->k

   (57)  PF 3
                                                              Type: DOMAIN
->K:=FF(3,2)

   (60)  FF(3,2)
                                                              Type: DOMAIN
->definingPolynomial()$K

          2
   (61)  ?  + 1
                                                         Type: SUP PF 3
->x:=monomial(1,1)$(SUP k)

   (63)  ?
                                                         Type: SUP PF 3
->irreducible?(x**2+x+2)

   (64)  true
                                                              Type: BOOLEAN
->K1:=FFP(k,x**2+x+2)

   (65)  FFP(PF 3,?*?+?+2)
                                                              Type: DOMAIN
->definingPolynomial()$K1

          2
   (66)  ?  + ? + 2
```

L'exemple de Sloane pour le corps a 9 éléments est bâti sur $x**2+x+2$.

6 Morphismes de corps

La fonction `coerce` qui dans toute sa généralité permet de changer de type, réalise ici les changements de type entre corps finis, c'est à dire les injections de corps, si elles existent :

Même corps, polynômesde définition différents

```
->K

   (67)  FF(3,2)
                                                                Type: DOMAIN
->K1

   (68)  FFP(PF 3,?*?+?+2)
                                                                Type: DOMAIN
->)cl pr a1
->a1:K:=random()

   (69)  0
                                                               Type: FF(3,2)
->a1:K:=random()

   (70)  2%J + 2
                                                               Type: FF(3,2)
->b1:K:=random()

   (75)  2%J
                                                               Type: FF(3,2)
->a2:K1:=a1

   (76)  2%K
                                                    Type: FFP(PF 3,?*?+?+2)
->b2:K1:=b1

   (78)  2%K + 1
                                                    Type: FFP(PF 3,?*?+?+2)
->(b1*a1)::K1-a2*b2

   (79)  0
```

```
                                                          Type: FFP(PF 3,?*?+?+2)
```

Même corps, représentations différentes

```
->K1:=FFNB(3,2)

   (82)  FFNB(3,2)
                                                                Type: DOMAIN
->K

   (83)  FF(3,2)
                                                                Type: DOMAIN
->a1

   (84)  2%J + 2
                                                              Type: FF(3,2)
->a1::K1

           q
   (85)  %L
                                                            Type: FFNB(3,2)
->minimalPolynomial %

          2
   (86)  ?  + 2? + 2
                                                             Type: SUP PF 3
->minimalPolynomial a1

          2
   (87)  ?  + 2? + 2
                                                             Type: SUP PF 3
\begin{verbatim}

On peut imaginer toutes les combinaisons possibles de repr\'esentation\ldots{}

\paragraph{Corps diff\'erents}

\begin{verbatim}
->K
```

```
   (88)  FF(3,2)
->K1:=FF(3,6)

   (98)  FF(3,6)
                                                                Type: DOMAIN
->a1:=random()

   (96)  2%J + 2
                                                                Type: FF(3,2)
->a1::K1

           4     3      2
   (99)  %N  + %N  + 2%N  + %N + 2
                                                                Type: FF(3,6)
->minimalPolynomial %

          2
   (101)  ?  + 2? + 2
                                                              Type: SUP PF 3
->minimalPolynomial a1

          2
   (102)  ?  + 2? + 2
                                                              Type: SUP PF 3
->a2:=primitiveElement()$K1

   (105)  %N
                                                                Type: FF(3,6)
->(3**6-1)/(3**2-1)

   (106)  91
                                                              Type: FRAC INT
->b2:=a2**91

             4      3     2
   (107)  2%N  + 2%N  + %N  + 2%N + 2
                                                                Type: FF(3,6)
->order %
```

```
   (108)  8

->b2::K

   (110)  %J + 2
                                                                Type: FF(3,2)
->minimalPolynomial %

            2
   (111)  ?  + 2? + 2
                                                              Type: SUP PF 3
->minimalPolynomial b2

            2
   (112)  ?  + 2? + 2
                                                              Type: SUP PF 3
->b2:=a2**32

              5     4     3      2
   (113)  2%N  + %N  + %N  + 2%N  + 2
                                                               Type: FF(3,6)
->minimalPolynomial b2

            6    5    4     3
   (114)  ?  + ?  + ?  + 2?  + ? + 1
                                                              Type: SUP PF 3
->b2::K

   >> Error detected within library code:
   element does'nt belong to smaller field
   You are being returned to the top level of the interpreter.

->b2

              5     4     3      2
   (115)  2%N  + %N  + %N  + 2%N  + 2
                                                               Type: FF(3,6)
->b2::FF(3,5)

   >> Error detected within library code:
```

```
   one extension degree must divide the other one
   You are being returned to the top level of the interpreter.

->b2::FF(7,3)

   Cannot convert from type FF(3,6) to FF(7,3) for value
      5     4     3      2
   2%N  + %N  + %N  + 2%N  + 2

   .
```

7 Ecriture de fichiers .input

7.1 "Statement", "blocks", indentation

Dans la syntaxe d'Axiom, les s"tatements" sont d'un ligne, ce sont les briques élémentaires d'un programme. Un retour chariot indique la fin d'un statement.

On peut continuer sur les lignes suivantes en neutralisant le caractère retour-charriot. Le neutralisateur dans axiom est _ (underscore en anglais, souligné en français).

Il est heureusement possible de regrouper sur une même ligne pusieurs statements, en les séparant par des points-virgule. Attention seule le résultat du dernier statement est affiché. C'est commode pour l'utilisation sous l'interprête.

Ceci nous amène à la définition de block. Un block est une suite de plusieurs statements, sépar'es par des `;`, mis entre parenthéses.

Il existe une écriture plus compacte des blocs, sous la forme de "pile" (à ne pas confondre avec la pile Lisp) de statements. Axiom reconnait comme faisant partie du même block les blocs indentés sur la même colonne, un statement étant un block. La valeur d'un block est la valeur du dernier statement (ou bloc) qui le compose.

Le type `Void` qui ne contient rien, est ke type que l'on donne à une boucle.

Dorénavant, comme il est pénible d'écrire les blocks indentés sous l'interprête, nous supposons avoir un emacs à coté, ou nous écrirons les fichiers ".input".

7.2 Lecture de fichiers ".input"

Un fichier .input est un fichier que l'on peut lire avec la commande `)read` . Le fichier est lu comme si on avait entré les commandes à la main sous l'interprête. Il est donc interprêté.

Les commandes du fichiers .input vont s'exécuter dans l'environnement donné par l'interprête au moment ou la commande `)read` est saisie. Cet environnement est changé au fur et à mesure du fichier.

7.3 Ecriture de fonctions

Une fonction se définit par une signature (ce qui lui donne un type), un corps de fonction (ce qui lui donne une valuer), et un nom. Attention à ne pas confondre le nom d'une fonction et le nom d'une variable égale à la fonction.

Définition de la signature d'une fonction :
Définition du corps d'une fonction :

7.3.1 Polynôme minimal d'un élément d'un corps fini

```
minPoly1: K-> SUP k
minPoly1(a)==
        p:=characteristic()$K
        pol:SUP k:=1
        for i in 0..(degree()$K -1) repeat pol:=pol*(x-a**(p**i)
        squareFreePart pol

minPoly2: K-> SUP k
minPoly2 a ==
        p:=characteristic()$K
        x:SUP K:=monomial(1,1)
        lelts := removeDuplicates! [ a**(p**i) for i in 0..degree(K)-1]
        lpols:=[x-toto for toto in lelts]
        pol:SUP k:=1
        for polbozo in lpols repeat pol:=pol*polbozo
        pol

minPoly3: K-> SUP k
minPoly3 a ==
        p:=characteristic()$K
        x:SUP K:=monomial(1,1)
        lelts := removeDuplicates [ a**(p**i) for i in 0..degree(K)-1]
        reduce(*, map( bozo +-> x-bozo , lelts) )

conjugues: K -> List K
conjugues a ==
        l:=[a]
        c:=frobenius(a)
        while c ^= a repeat
                concat_!(l,c)
                c:=frobenius(c)
        l

minPoly4: K -> List K
minPoly4 a ==
```

```
        x:SUP k:=monomial(1,1)
        reduce(*, map( bozo +-> x-bozo ,conjugues a) )

lift: K -> UP(%%dummy,k)
lift a ==
        pol:=0
        l:=coordinates a
reduce(+.map(x+-> (l.x)*%%dummy**x,[i for i in 1..%l])

minPoly5: K -> UP(X,k)
minPoly5 a ==
x:SUP k := monomial(1,1)
        defpol:=(definingPolynomial()$K)(%%dummy)
        squareFreePart resultant(X-lift(a),defpol,%%dummy)
```

7.3.2 Axiom en Batch

Encore une fois ce n'est pas documenté. Bon ce qui a l'air de marcher c'est :

```
myfavoriteRS6000.inria.fr% ls
toto.input
myfavoriteRS6000% axiom -nox -ws spadsys < toto.input > toto.ouput &
```

L'option `-nox` permet de préciser de fonctionner sans utiliser le serveur X.

8 Ecriture de fichiers .spad

8.1 Partie publique, partie privée

Sans rentrer dans les détails, un domaine est défini par sa partie publique, i.e. les fonctions (ou méthodes exportées par le domaine), que l'on définit par une catégorie. Vient ensuite la capsule qui définit l'implémentation des fonctions exportées.

Le fichier doit contenir les lignes suivantes :

```
)abb domain MYDOM2 MyDomainSuperMieuxEtToutEtTout2
```

En effet, le but est de produire un module binaire, en code machine, qui peut être chargé sous l'interprête. Le binaire correspondant sera alors MYDOM2.NRLIB.

Il devra aussi y avoir la définition du domaine :

```
MyDomainSuperMieuxEtToutEtTout2(arguments declares): public == prive
```

Ceci est à rapprocher de la définition de fonction :

```
foo(i:INT):INT == i**2
```

Il n'y a pas de grandes différences avec l'écriture d'une fonction. `public` est une catégorie, i.e. un type de domaine, et privé une implémentation des fonctions de la catégorie, et des objets du domaine.

Un package est un domaine qui ne possède pas d'éléments, i.e. dont aucune fonction ne contient `$` dans sa signature.

8.2 Mise en package des fonctions du ".input" précédent

On va d'abord nommer le package :

```
)abb package FFUTILS FiniteFieldUtilities

FiniteFieldUtilities(K:FFIELC): public ==prive where

        FFIELC  ==> FiniteFieldCategory
        SUP     ==> SparseUnivariatePolynomial

        public ==> with

          conjugues     : K -> List K
```

```
  minPolyBozo   : K -> SUP  K
  minPoly       : K -> SUP  K

prive  ==> add

  K:=PF p

  x:SUP K:=monomial(1,1)

  monom: K -> SUP K
  monom a == x-a

  conjugues a ==
        l:=[a]
        c:=frobenius(a)
        while c ^= a repeat
                l:=cons(c,l)
                c:=frobenius(c)
        l

  minPolyBozo a ==
        reduce(*, map( bozo +-> x-bozo ,conjugues a) )

  minPoly a ==
        reduce(*, map( monom, conjugues a) )
```