# An Implementation of Rational Functions in PCL*

W. Windsteiger

March 16, 1995

**Abstract**

The main aim of our work with PCL, a kind of "object-oriented" expansion of COMMON-LISP, is to test whether it is suitable for the implementation of our Gröbner bases package. Criteria for this are speed and the facility of generic programming. For this purpose we implemented the domain of *Rational Functions*. For arithmetic in this field it is necessary to compute the greatest common divisor (gcd) of two multivariate polynomials, which was the main effort of this work.

## 1 Introduction

In this paper a generic implementation of *Rational Functions* is described. Arithmetic for the occurring polynomials is independent of the domain of coefficients. Special attention was focussed on the *gcd of polynomials* as simplification-algorithm for Rational Functions.

The modular approach goes back to [Knuth 69], [Brown 71] and [Collins 72]. In our implementation we follow the description given in [Davenport 88] and [Winkler 88] for computing the gcd of polynomials. The gcd is computed modulo several *small primes* and is then lifted to a solution of the original problem. In section 2 the algebraic foundations are described. Properties of the gcd and a first sketch of the algorithm are given in sections 2.1 and 2.2, the Chinese Remainder Technique, a crucial part of the algorithms, is stated in section 2.3. In sections 2.4.1 and 2.5 Rational Functions and the principle of modular computation are described.

Furthermore in section 3 some hints on the implementation are given. Examples, timings and the source code can be found in the Appendix but is to be said that work on the implementation is not finished and therefore is not perfected both in style and in speed, see section 4.1.2.

*This research work was done in the frame of the Gröbner Bases project sponsored by the *Bundesministerium für Wissenschaft und Forschung*, Austria.

# 2 Algebraic Background

## 2.1 The GCD of Univariate Polynomials

Let $f, g \in \mathbb{Z}[x]$. If two polynomials $f, g$ have a common factor $h$, they can be written as

$$f = f_1 h \ , \ g = g_1 h.$$

This property also holds if one takes all coefficients modulo $p$, where $p$ is a prime number. The gcd modulo $p$ can be computed much faster because the coefficients are bounded by $p$. So the main disadvantage of the Euclidean algorithm, the intermediate growth of coefficients, can be suppressed. One repeats this procedure for several primes $p_i$ and then "reconstructs" the solution in $\mathbb{Z}[x]$ from these solutions in $\mathbb{Z}_{p_i}[x]$. The algorithm is roughly as follows:

1. Compute a bound $b$ for the absolute values of the coefficients in the gcd.
2. Choose primes $p_1, \ldots, p_n < b$ s.t. $\prod_{i=1}^n p_i > 2b$ and compute the gcd's modulo $p_1, \ldots, p_n$.
3. Obtain a solution modulo $\prod_{i=1}^n p_i$ by applying the Chinese Remainder Theorem (see section 2.3). This solution can then be viewed as a solution over $\mathbb{Z}$ because all coefficients in the gcd are bounded by $b < 1/2 \prod_{i=1}^n p_i$.

Of course a lot of details have to be proven.

**Lemma 1** *Each coefficient of the gcd of $f = \sum_{i=0}^m f_i x^i$ and $g = \sum_{i=0}^n g_i x^i$ with coefficients in $\mathbb{Z}$ is bounded by the "Landau-Mignotte-bound"*

$$2^{\min(m,n)} \gcd(f_m, g_n) \min\left(\frac{1}{f_m}\sqrt{\sum_{i=0}^m f_i^2}, \frac{1}{g_n}\sqrt{\sum_{i=0}^n g_i^2}\right) .$$

Notation: $f = \sum_{i=0}^m f_i x^i \in \mathbb{Z}[x]$, $p$ a prime number.
If one takes all the $f_i$'s modulo $p$ the resulting polynomial is denoted by $f_p$. $d$ denotes the gcd of $f$ and $g$.

**Lemma 2** *If $p$ does not divide both leading coefficients of $f$ and $g$ then the degree of* $\gcd(f_p, g_p)$ *is not less than the degree of $d$.*

**Lemma 3** *$p$ as in Lemma 2. If $p$ does not divide the resultant of $f/d$ and $g/d$ then* $\gcd(f_p, g_p) = d_p$.

Proofs of these lemmata: see [Winkler 88].

There are only finitely many primes, which do not divide the leading coefficients but for which the degree of $\gcd(f_p, g_p)$ is greater than the degree of $d_p$. Such primes are called "unlucky primes". The lemmata make sure that the algorithm as sketched above ends after finitely many steps.

## 2.2 The GCD of Multivariate Polynomials

If $f$ and $g$ are elements of the polynomial ring $\mathbb{Z}[x_1, \ldots, x_n]$ the whole process runs similarly: view the polynomials as elements of $\mathbb{Z}[x_1, \ldots, x_{n-1}][x_n]$. Taking the coefficients (polynomials in $n-1$ variables) modulo the prime ideal generated by $x_{n-1} - r$ (these are the remainders from dividing the coefficient polynomials by $x_{n-1} - r$) is equivalent to evaluating the coefficient polynomials for $x_{n-1} = r$. Repeating this evaluation recursively reduces the number of variables and leads to the univariate gcd-computation. Recursive application of the Chinese Remainder Theorem (see section 2.3.1) leeds back to the solution for the multivariate case.

Whereas computing a bound for the coefficients is rather difficult in the univariate case, this is easy in the multivariate case.
Notation: Let $f \in I[x_1, \ldots, x_n]$ then, for every $i$, $\deg_{x_i} f$ denotes the degree of $f$ in the variable $x_i$.

**Lemma 4** *Let $I$ be an arbitrary Integral Domain. If $h$ is a factor of $f$ in $I[y,x]$, then* $\deg_y h \leq \deg_y f$.

There are also lemmata equivalent to those mentioned in the univariate case that guarantee that the algorithm stops after finitely many steps. For further details see [Davenport 88].

## 2.3 The Chinese Remainder Theorem

**Theorem 1** *(Chinese Remainder Theorem) Let $D$ be a Euclidean Domain, $r_1, \ldots, r_n \in D$, $m_1, \ldots, m_n \in D$ pairwise relatively prime. Then there exists an $r \in D$ such that for every $i \in \{1, \ldots, n\}$ $r \equiv_{m_i} r_i$, where $\equiv_{m_i}$ denotes congruency modulo $m_i$. This solution is unique modulo $\prod_{i=1}^{n} m_i$.*

For $n = 2$ the algorithm proceeds as follows:
**Algorithm CRA2:**

1. $c \leftarrow m_1^{-1} \bmod m_2$.
2. $r_1\prime \leftarrow r_1 \bmod m_1$.
3. $\sigma \leftarrow ((r_2 - r_1\prime \bmod m_2) \cdot c) \bmod m2$.
4. $r \leftarrow r_1\prime + \sigma m_1$.

For $n > 2$ one first computes the solution for the first two congruencies. The solution $R_1$ is then combined with the third congruency and so on.

### 2.3.1 The Chinese Remainder Theorem for Polynomials

The case of polynomials over the integers must be treated separately because $\mathbb{Z}[x]$ and $\mathbb{Z}[x_1, \ldots, x_n]$ are not Euclidean Domains. The crucial operation in the algorithm CRA2 is step 1. The inverse of an integer polynomial modulo another integer polynomial need not have integer coefficients. For example, $f = (x-1)(x-2) = x^2 - 3x + 2$, $g = (x-3)$. By the extended Euclidean algorithm we get $1 = f \cdot 1/2 - g \cdot x/2$ and therefore $f^{-1} = 1/2 \mod g$.

In the situation of gcd-computation this does not matter. The gcd of two integer polynomials is again an integer polynomial. All moduli we consider are monic (i.e. their leading coefficient is one), hence, the image of the gcd modulo such a modulus has again integer coefficients (division of an integer polynomial by a monic integer polynomial yields quotient and remainder both polynomials over the integers!). Therefore the polynomial $r$ that we get in step 4 has to be an integer polynomial, so the polynomial $\sigma$ that we introduce in step 3 must have integer coefficients because $r_1\prime$ and $m_1$ have integer coefficients. Summarizing all this one can switch to computation over the rational numbers in steps 1 and 3 because it is guaranteed that all denominators vanish at the latest after step 3.

## 2.4 Rational Functions (Fractions)

### 2.4.1 What are Rational Functions?

Whenever $I$ is an Integral Domain then the ring of polynomials with coefficients in $I$, denoted by $I[x_1 \ldots x_n]$, is also an Integral Domain. By the following construction every Integral Domain $I$ can be embedded into a field:

1. We define a relation $\sim$ on $I \times I$:

$$(a, b) \sim (c, d) :\Longleftrightarrow ad = bc \ .$$

2. Define: $(a, b) + (c, d) = (ad + bc, bd) \quad , \quad (a, b) \cdot (c, d) = (ac, bd)$.

3. $\sim$ is a congruence relation.

4. $Q(I) := (I \times (I \backslash \{0\}))/_{\sim}$ .

5. $(Q(I), +, \cdot)$ is a field and is called the Quotient Field of $I$. An element $[(a, b)]$ ( = the equivalence class of $(a, b)$ ) is denoted by $\frac{a}{b}$.

In the case $I = R[x]$ (or $I = R[x_1, \ldots, x_n]$) for an arbitrary integral domain $R$, $Q(I)$ is called the field of Rational Functions and is denoted by $R(x)$ ($R(x_1, \ldots, x_n)$ respectively). The most important case in Computer Algebra is the case $I = \mathbb{Z}[x_1, \ldots, x_n]$.

### 2.4.2 Arithmetic for Rational Functions

Since Rational Functions are equivalence classes one desires to have a *canonical simplifier* for these classes, i.e. a function that computes unique representatives for equivalence classes. A canonical simplifier $S$ for the equivalence relation $\sim$ on the class $O$ should fulfill the following two restrictions for all objects $o$ in $O$:

1. $S(o) \sim o$ and
2. $S(o)$ should be "simpler" than $o$, whatever "simpler" means.

In the case of Rational Functions a low degree of the numerator and the denominator is a reasonable property of a "simple" object. Then the following algorithm is a canonical simplifier for Rational Functions: "Divide both the numerator and the denominator by their gcd, then (if necessary) multiply numerator and denominator by -1 to make the leading coefficient of the denominator positive". Hence, the gcd of multivariate polynomials is the crucial operation for the *simplification problem* for Rational Functions, see also section 4.1.2 for timings.

Addition and multiplication were already defined in section 2.4.1 on introducing Rational Functions. The identity for $+$ is given by $\frac{0}{1}$, the identity for $\cdot$ is given by $\frac{1}{1}$. $\frac{-a}{b}$ is the inverse of $\frac{a}{b}$ with respect to $+$, the inverse to $\frac{a}{b}$ with respect to $\cdot$ is $S(\frac{b}{a})$.

A straightforward approach for doing arithmetic in a quotient field would just use the definition of the sum and product from section 2.4.1:

$$\frac{a}{b} + \frac{c}{d} = S\left(\frac{ad+bc}{bd}\right) \quad , \quad \frac{a}{b} \cdot \frac{c}{d} = S\left(\frac{ac}{bd}\right) \quad .$$

However, this is not the best method. Using the algorithms of Henrici for summation and multiplication of fractions results in a drastic save of time for these computations (depending on the number of variables). These algorithms use two gcd-computations with "simple" elements instead of one computation with "non-simple" elements, see [Winkler 88] for some details.

## 2.5 Modular Arithmetic

For the purpose of modular computation of the gcd for polynomials over $\mathbb{Z}$ it is sufficient to do arithmetic modulo a positive integer $m$, called the modulus. We denote by $H_m$ the *canonical simplifier associated with* $m$, namely $H_m(a) = remainder(a, m)$. Summation and multiplication are defined as follows:

$$a +_m b = H_m(a+b) \quad , \quad a \cdot_m b = H_m(a \cdot b) \quad .$$

$H_m$ should be modified a little for summation, because in this case it is preferable just to subtract the modulus instead of computing the remainder.

There are two natural choices for a set of representatives for residue classes:

1. $\{0, \ldots, m-1\}$
2. $\{a | -m/2 < a \leq m/2\}$

Each of these choices has certain advantages and disadvantages. Whatever the choice is, it is denoted by $\mathbb{Z}_m$.

The crucial operation in $\mathbb{Z}_m$ is division. Let us view the quotient of $a$ and $b$ as the product of $a$ and the inverse of $b$ (i.e. $a/b = ab^{-1}$). In $\mathbb{Z}_m$ $b^{-1}$ only exists for those $b$ where $\gcd(b, m) = 1$. In this case we can compute $u, v$ such that $u \cdot b + v \cdot m = 1$ by the extended Euclidean algorithm. $v \cdot m \equiv 0 \pmod{m}$, so $b^{-1} = H_m(u)$. Furthermore, if $p$ is a prime number, every $b$ in $\mathbb{Z}_p$ is invertible and so $\mathbb{Z}_p$ is a field for every prime number $p$.

# 3 Implementation

## 3.1 Basics

The implementation is done in "PCL", a kind of "object-oriented" expansion of "COMMON LISP". It is possible to define "classes" and then implement "methods" for the instances of these classes. Whenever a function is called, a type-check is performed and the suitable method is called. There is also the possibility to establish a hierarchy of classes by building subclasses. Each method for instances of a certain class is also valid for the instances of all subclasses.

Every instance of a class has one or more "slots" containing its values. If one wants a slot to be visible to all instances of a class, one can define this particular slot as a "shared slot", but these instances can have "non-shared slots" as well. An example for this are the modular numbers, where one slot contains the value and the second slot contains the modulus. The first one is non-shared whereas the second can be shared when all elements have the same modulus.

## 3.2 Syntax of the most important constructs

(defclass [*class-name*]
  (*superclass*)
  ( (slot-specifier$_1$)
      ⋮
    (slot-specifier$_n$)))

where each slot-specifier has the form

(*slot-name* [*accessor-function*] [*init-function*] ...) .

With the "defclass-construct" one can define new classes (optionally as subclasses of other classes) where the slot-specifiers define the number of slots and their behavior (shared or non-shared, name of the accessor-function, name of the init-function,...).

(make-instance [*class-name*]
[$\textit{init-function}_1$][$\textit{arg}_1$] ... [$\textit{init-function}_n$][$\textit{arg}_n$])

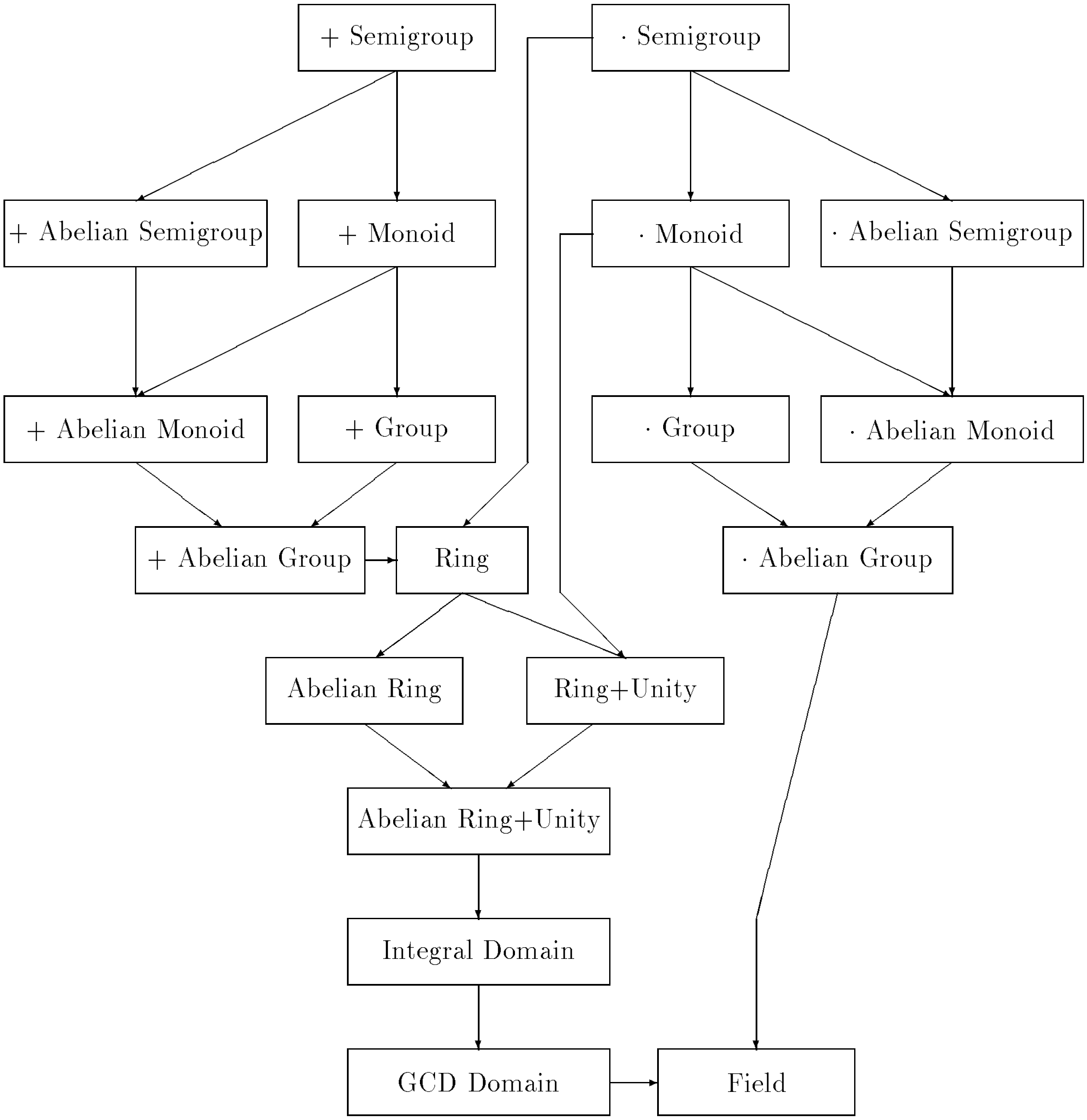
creates an instance of the class and initializes the slots with the given arguments, but not every slot has to be initialized. Such a slot is then called "unbounded".

(defmethod *name* [($\textit{arg}_1$ $\textit{class}_1$)] ... [($\textit{arg}_n$ $\textit{class}_n$)])

defines an n-ary function, where $\textit{arg}_i$ is an instance of $\textit{class}_i$ for $i \in \{1, \ldots, n\}$. For syntactical details see [Bobrow,DeMichiel,Gabriel,Keene,Kiczales,Moon].

## 3.3 Algebraic Structure

The basis for the whole package is the implementation of algebraic structures like groups, rings, fields,... .
Notation: "+ Semigroup" denotes the semigroup for addition, "$\cdot$ Semigroup" the semigroup for multiplication and $\boxed{*} \longrightarrow \boxed{**}$ says that $*$ is implemented as a subclass of $**$. In each of these classes a *characteristic function* for the domain is introduced e.g. a characteristic function in "+ Group" is the difference of two elements, which can of course be defined whenever an inverse with respect to addition is available.



+ Semigroup
· Semigroup
+ Abelian Semigroup
+ Monoid
· Monoid
· Abelian Semigroup
+ Abelian Monoid
+ Group
· Group
· Abelian Monoid
+ Abelian Group
Ring
· Abelian Group
Abelian Ring
Ring+Unity
Abelian Ring+Unity
Integral Domain
GCD Domain
Field

## 3.4 Representation of Objects

- *Fractions* are elements with two slots, first the numerator and second the denominator. Fractions can be built from integral domains, but in our specific implementation the gcd is used for simplification so we can build fractions only from gcd-domains. Arithmetic operations use the Henrici-Algorithms.

- With *polynomials* we mean polynomials with *integer coefficients*. They are in *sparse* representation with two slots both containing a list. The first list is associated with the exponents with non-zero coefficients, the second contains the corresponding coefficients. The exponent-list is assumed to be sorted with the highest exponent as first element and all operations on polynomials are implemented in such a way that this property holds for the result as well. The class of polynomials, "poly-sparse", has two subclasses, namely classes of univariate and multivariate polynomials, "poly-univ-sparse" and "poly-multiv-sparse". Multivariate polynomials are polynomials whose coefficients are again polynomials, univariate polynomials have integers as coefficients. This recursive representation of polynomials was chosen because it fits best to the implemented gcd-algorithm.

- *Modular numbers* are implemented in two different ways: the class "residue" offers in contrary to the class "residue-store" a shared slot for the module. So for computation one would choose the class "residue" whereas for storing values with different moduli one has to use "residue-store".

## 3.5 Comments on the Implementation

This section is meant to give help in studying the source code, which is listed in section 4.2.

### 3.5.1 File *numbers*

In this file the built-in arithmetic operations on the built-in types "integer", "rational", "float",... are renamed in such a way that object-oriented behavior is guaranteed. It also contains an error-detect-function and a function ``show'' for lists needed for output of polynomials.

### 3.5.2 File *ordered-object*

Contains order-predicates based on the predicates ``less?'' and ``equal?'', so that for each ordered domain the only predicates to implement are ``less?'' and ``equal?''.

### 3.5.3 File *algebraic-domains*

In this file the class definitions for basic algebraic domains in the hierarchy as sketched in section 3.3 are given. Furthermore the characteristic functions for each domain can be found in this file.

### 3.5.4 Files *rational, integer, float*

These files would not be necessary in a totally finished version because in such a version the domains *rational, integer* and *float* should be available as self-defined classes so that they can be subclasses of other classes. Then all these operations that are in these files now would be superfluous.

### 3.5.5 File *prime-generation*

Includes functions for generation of prime numbers needed in the univariate gcd-algorithm and functions for finding integers satisfying the properties needed in the multivariate gcd-algorithm.

### 3.5.6 File *residue*

Simplification is provided by computing the remainder of value and modulus out of the set of representatives $\{a / -m/2 < a \leq m/2\}$ so that negative numbers are available, too. In the case of addition and subtraction simplification is done as sketched in section 2.5 on modular arithmetic. The class "residue-store" contains mainly the same as "residue" but offers a non-shared slot for the modulus.

### 3.5.7 File *poly-sparse*

Contains all operations on polynomials that are used for both univariate and multivariate polynomials in the same way. The constructor-function gets an additional argument to determine what kind of polynomial to create. No fast algorithms on polynomials (e.g. Karatsuba-algorithm, Fast Fourier Transformation,...) for multiplication are implemented.

### 3.5.8 File *poly-univ-sparse*

In addition to the description in section 2.1 on the gcd of univariate polynomials the function ``greatest-common-divisor'' uses the formula

$$\gcd(f,g) = \gcd(pp(f), pp(g)) \cdot \gcd(cont(f), cont(g)) \quad ,$$

where $pp(f)$ denotes the primitive part of $f$ and $cont(f)$ denotes the content of $f$. The test whether a common divisor is already found is made after every stage where the result did not change.
The print-function ``show'' is not yet perfectly developed but its major task is just to show a result without respect to style.

#### 3.5.9 File *poly-multiv-sparse*

*poly* $\in K[x_1, \ldots, x_{n-1}][x_n]$. The function ``poly-homomorphism( poly, m)'' computes the homomorphic image of *poly* modulo the ideal generated by $m$. Since $m$ is always of the form $x_{n-1} - c$ this is the same as evaluating *poly* for $x_{n-1} = c$. The function ``greatest-common-divisor'' works as described in section 2.2 on the gcd of multivariate polynomials. For the function ``show'' the same is true as in the univariate case.

#### 3.5.10 File *fraction*

Summation and multiplication use the algorithms of Henrici, the predicate ``less?'' is not jet totally general.

#### 3.5.11 File *constructor*

Provides faster constructors for instances of a class. ``defconstructor'' does not need so much memory and so results in an increase of speed.

### 3.6 Summary

The algorithms are not perfectly developed yet. Especially the gcd-implementation could be improved with respect to style. Since both, the univariate and the multivariate case, are modular algorithms of the *same scheme*, **one generic** version of this algorithm would be sufficient in the final version of this package. This algorithm would then roughly look like the sketch in section 2.1, where the general idea of the modular method was explained. It would then contain subalgorithms like ``choose-prime-ideal-generator'', which has a certain meaning in the univariate and the multivariate case. In the first case it would choose a *prime number* that has the properties of the corresponding lemmata stated in section 2.1 about the univariate gcd-algorithm, in the second case a *monic polynomial* of degree one, for that the equivalent lemmata hold.

Further improvements can be gained by implementing fast multiplication algorithms for polynomials.

# 4 Appendix

## 4.1 Examples

### 4.1.1 Test-data

- $poly1 = x^5 - x^4 - 3x^2 - 3x + 2 = (-2+x)(1+x)(-1+2x+x^3)$
- $poly2 = x^4 - 2x^3 - 3x^2 + 4x + 4 = (-2+x)^2(1+x)^2$
- $poly3 = (poly1)^6(poly2)^8$. $poly3$ has degree 62, all coefficients are non-zero, coefficients range from 1 to 15 digits.
- $poly4 = (poly1)^{10}(poly2)^5$. $poly4$ has degree 70, all coefficients non-zero and in the same range as in *poly3*.
- $poly5 = poly3 poly4$. $poly5$ has degree 132, all coefficients are non-zero, coefficients range from 1 to 23 digits.
- $poly6 = x^5 + 3x^4 + 2x^3 - 2x^2 - 3x - 1$
- $poly7 = x^6 + 3x^5 + 3x^4 + 2x^3 + 3x^2 + 3x + 1$
- $poly12 = (y^2 - y - 1)x^2 - (y^2 - 2)x + (y^2 + y + 1)$
- $poly13 = (y^2 - y + 1)x^2 - (y^2 + 2)x + (y^2 + y + 2)$
- $poly14 = z^4(y-1)(x+1)$
- $poly15 = z^2(y^3 - y^2)(x+1)^2$
- $poly16 = z^6 + z^5 y + zy^3 - zy + y^4 - y^2$
- $poly17 = z^5 y - 2z^5 + z^4 y^2 - 2z^4 y + z^2 y + zy^2$
- $poly18 = z(y^4 + y)(x+1)^2$
- $poly19 = z^2(x+1)^3$
- $f1 = \frac{poly1}{poly6} = \frac{x^4 - 2x^3 + 2x^2 - 5x + 2}{x^4 + 2x^3 - 2x - 1}$
- $f2 = \frac{poly7}{poly2} = \frac{x^4 + x^3 + x + 1}{x^2 - 4x + 4}$
- $s1 = f1 + f2 = \frac{x^8 + 3x^7 + 3x^6 - 7x^5 + 14x^4 - 20x^3 + 28x^2 - 31x + 7}{x^6 - 2x^5 - 4x^4 + 6x^3 + 7x^2 - 4x - 4}$
- $d1 = f1 - f2 = \frac{-x^8 - 3x^7 - x^6 - 5x^5 + 14x^4 - 22x^3 + 32x^2 - 25x + 9}{x^6 - 2x^5 - 4x^4 + 6x^3 + 7x^2 - 4x - 4}$
- $p1 = f1 \cdot f2 = \frac{x^5 - x^4 + 3x^3 - 3x^2 + 3x - 1}{x^3 - 2x^2 - x + 2}$

- $f3 = \frac{poly14}{poly15} = \frac{z^2}{(x+1)y^2}$
- $f4 = \frac{poly18}{poly19} = \frac{y^4+y}{(x+1)z}$
- $s2 = f3 + f4 = \frac{z^3+y^6+y^3}{zy^2(x+1)}$
- $p2 = f3 \cdot f4 = \frac{z(y^3+1)}{y(x+1)^2}$
- $d2 = p2 - s2 = \frac{-(x+1)z^3+z^2(y^4+y)-(x+1)(y^6+y^3)}{zy^2(x+1)^2}$
- $s3 = f3 + (f4)^{-1} = \frac{z^2(y^3+1)+zy(x+1)^2}{(y^5+y^2)(x+1)}$
- $poly30 = f3 \cdot a^2 + f4 \cdot a + s2$
- $poly31 = p2 \cdot a^2 + s3 \cdot a$

### 4.1.2 Time Statistics

Computations were carried out on an *Apollo 3500* workstation with 5 MIPS.

| Operation | Result | Time(sec.) |
|---|---|---|
| gcd($poly1, poly2$) | $x^2 - x - 2$ | 4,17 |
| Compute $poly3$ | $poly3$ | 11,43 |
| Compute $poly4$ | $poly4$ | 16,9 |
| $poly3 + poly4$ | degree: 70 | 0,33 |
| | 1 - 15 digits | |
| $poly3 \cdot poly4$ | $poly5$ | 32,19 |
| $(poly5)^2$ | $(poly5)^2$ | 133,84 |
| gcd($poly12, poly13$) | 1 | 2,04 |
| gcd($poly14, poly15$) | $z^2(y-1)(x+1)$ | 5,86 |
| gcd($poly16, poly17$) | $z + y$ | 34,04 |
| gcd($poly18, poly19$) | $z(x+1)^2$ | 5,58 |
| Compute $f1$ | $f1$ | 8,23 |
| Compute $f2$ | $f2$ | 10,54 |
| $f1 + f2$ | $s1$ | 2,71 |
| $f1 - f2$ | $d1$ | 2,70 |
| $f1 \cdot f2$ | $p1$ | 10,47 |
| Compute $f3$ | $f3$ | 6,20 |
| Compute $f4$ | $f4$ | 5,81 |
| $f3 + f4$ | $s2$ | 4,00 |
| $f3 \cdot f4$ | $p2$ | 4,28 |
| $p2 - s2$ | $d2$ | 6,41 |
| $f3 + (f4)^{-1}$ | $s3$ | 3,06 |
| $poly30 + poly31$ | — | 9,71 |
| $poly30 \cdot poly31$ | — | 41,25 |

## 4.2 Source Code

```
;**********************************************************
;
;  number
;
;  provides "mixed operations" on built-in type "number"
;
;**********************************************************


(defmethod sum ((a number) (b number)) (+ a b))

(defmethod difference ((a number) (b number)) (- a b))

(defmethod negative ((a number)) (- 0 a))

(defmethod product ((a number) (b number)) (* a b))

(defmethod power ((a number) (b number)) (expt a b))

(defmethod equal? ((a number) (b number)) (= a b))

(defmethod zero? ((a number))
  (equal? a (zero a)))

(defmethod one? ((a number))
  (equal? a (one a)))

(defmethod less? ((a number) (b number)) (< a b))

(defmethod less-or-equal? ((a number) (b number)) (<= a b))

(defmethod greater? ((a number) (b number)) (> a b))

(defmethod greater-or-equal? ((a number) (b number)) (>= a b))

(defmethod maximum ((n1 number) (n2 number)) (max n1 n2))

(defmethod minimum ((n1 number) (n2 number)) (min n1 n2))


; ++++++++ Error-function ++++++++

(defun detect-error (function-name &rest arguments)
```

```
  (princ '|Error occured in function |)
  (princ function-name)
  (princ '| when called with arguments:|)
  (terpri)
  (show arguments))



; ++++++++ Print-function ++++++++

(defmethod show ((l list))
  (show-list l 10))

(defun show-list (l n)
  ; print all elements of list "l"
  ; with n elements per line
  (show-list* l n n))

(defun show-list* (l n total)
  (cond ((null l) nil)
        ((zero? n) (terpri) (show-list* l total total))
        (T (show (car l)) (princ '|   | ) (show-list* (cdr l) (- n 1)
                                                      total))))

;*********************************************************
;
;       ordered-object
;
;*********************************************************


(defmethod greater? (x y) (less? y x))

(defmethod less-or-equal? (x y) (or (less? x y) (equal? x y)))

(defmethod greater-or-equal? (x y) (or (greater? x y) (equal? x y)))


; Now we presuppose that the domain is totally ordered

(defmethod maximum (x y)
  (if (greater-or-equal? x y) x y))

;*******************************************************************
;
;                  algebraic domains
```

```
;    (class definitions and characteristic functions)
;
;*******************************************************************



;*******************************************************************

(defclass additive-semi-group-element
  () ())

;*******************************************************************

(defclass additive-monoid-element
  (additive-semi-group-element)())

(defmethod zero? ((m additive-monoid-element))
  (equal? m (zero m)))

;*******************************************************************

(defclass additive-group-element
  (additive-monoid-element)())

(defmethod difference ((a additive-group-element)
                       (b additive-group-element))
  (sum a (negative b)))

;*******************************************************************

(defclass abelian-additive-semi-group-element
  (additive-semi-group-element) ())

(defmethod sum ((a abelian-additive-semi-group-element)
                (b abelian-additive-semi-group-element))
  (sum b a))

;*******************************************************************

(defclass abelian-additive-monoid-element
  (additive-monoid-element abelian-additive-semi-group-element)())

;*******************************************************************

(defclass abelian-additive-group-element
  (abelian-additive-monoid-element additive-group-element)())
```

```
;*******************************************************************

(defclass multiplicative-semi-group-element
  () ())

(defmethod power ((m multiplicative-semi-group-element) (n integer))
  (power*  m (one m) n))

(defmethod power* ((m1 multiplicative-semi-group-element)
                   (m2 multiplicative-semi-group-element)
                   (times integer))
  (cond ((zero? times) m2)
        ( T (power* m1 (product m1 m2) (- times 1)))))

(defmethod square ((m multiplicative-semi-group-element))
  (power m 2))

;*******************************************************************

(defclass multiplicative-monoid-element
  (multiplicative-semi-group-element)())

(defmethod one? ((m multiplicative-monoid-element))
  (equal? m (one m)))

;*******************************************************************

(defclass multiplicative-group-element
  (multiplicative-monoid-element)())

(defmethod quotient ((a multiplicative-group-element)
                     (b multiplicative-group-element))
  (product a (inverse b)))

;*******************************************************************

(defclass abelian-multiplicative-semi-group-element
  (multiplicative-semi-group-element) ())

(defmethod product ((a abelian-multiplicative-semi-group-element)
                    (b abelian-multiplicative-semi-group-element))
  (product b a))

;*******************************************************************
```

```
(defclass abelian-multiplicative-monoid-element
  (multiplicative-monoid-element
   abelian-multiplicative-semi-group-element)())

;*****************************************************************

(defclass abelian-multiplicative-group-element
  (abelian-multiplicative-monoid-element
   multiplicative-group-element)())

;*****************************************************************

(defclass ring-element
  (abelian-additive-group-element
   multiplicative-semi-group-element) ())

;*****************************************************************

(defclass ring-with-identity-element
  (ring-element  multiplicative-monoid-element) ())

;*****************************************************************

(defclass abelian-ring-element
  (ring-element abelian-multiplicative-semi-group-element)())

;*****************************************************************

(defclass abelian-ring-with-identity-element
  (ring-with-identity-element  abelian-ring-element) ())

;*****************************************************************

(defclass integral-domain-element
  (abelian-ring-with-identity-element) ())

;*****************************************************************

(defclass gcd-domain-element
  (integral-domain-element) ())

(defmethod g-c-d-extended ((a gcd-domain-element) (b gcd-domain-element))
  (greatest-common-divisor-ex a b))

(defmethod g-c-d ((a gcd-domain-element) (b gcd-domain-element))
  (greatest-common-divisor a b))
```

```
(defmethod cofactor1 ((a gcd-domain-element) (b gcd-domain-element))
  (second (g-c-d-extended a b)))

(defmethod cofactor2 ((a gcd-domain-element) (b gcd-domain-element))
  (third (g-c-d-extended a b)))

(defmethod g-c-d (a b)
  (terpri)
  (princ '|Arguments are not elements of a "GCD-DOMAIN"|) (terpri)
  (princ '|First argument:|) (terpri)
  (show a) (terpri) (terpri)
  (princ '|Second argument:|) (terpri)
  (show b))

;*******************************************************************

(defclass field-element
  (gcd-domain-element abelian-multiplicative-group-element) ())

(defmethod g-c-d ((f1 field-element) (f2 field-element)) (one f1))

;*******************************************************************

;*********************************************************
;
;   rational
;
;*********************************************************


; The Domain of Rational Numbers is a Field.


(defmethod zero ((r rational)) 0)

(defmethod one ((r rational)) 1)

(defmethod inverse ((r rational)) (/ 1 r))

(defmethod quotient ((r1 rational) (r2 rational)) (/ r1 r2))

(defmethod show ((r rational))
  (cond ((equal? r (zero r)) (princ '0))
        ((equal? (denominator r) (one (denominator r)))
                 (princ (numerator r)))
```

```
        ( T (princ (numerator r))
            (princ '/)
            (princ (denominator r)))))

;*********************************************************
;
;         integer
;
;*********************************************************

; The Domain of Integers is a Ring.
; The Domain of Integers is a Subdomain
;     of the Domain of Rationals.

(defmethod g-c-d ((i1 integer) (i2 integer))
  (gcd i1 i2))


;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;
; g-c-d-extended (x y) -> (g a b)  where g = gcd(x,y)
;                                    and  g = ax + by
;
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defmethod g-c-d-extended ((x integer) (y integer))
  ; gcd of 2 elements + cofactors
  (gcd-euclid-extended x y (one x) (zero x) (zero x) (one x)))

(defun gcd-euclid-extended (z1 z2 a1 a2 b1 b2)
  (cond ((equal? z2 (zero z2)) (let* ((result (list z1 a1 a2))
                                      (nf (normalizing-factor
                                            (first result))))
                                 (mapcar #'(lambda (x) (product x nf))
                                       result)))
         ( T (let* ((r1 (remainder z1 z2))
                    (q  (divide z1 z2))
                    (c1 (difference a1 (product b1 q)))
                    (c2 (difference a2 (product b2 q))))
           (gcd-euclid-extended z2 r1 b1 b2 c1 c2)))))


(defmethod cofactor1 ((i1 integer) (i2 integer))
  (second (g-c-d-extended i1 i2)))
```

```
(defmethod cofactor2 ((i1 integer) (i2 integer))
  (third (g-c-d-extended i1 i2)))

(defmethod least-common-multiple ((i1 integer) (i2 integer))
  (lcm i1 i2))

(defmethod divide ((i1 integer) (i2 integer))
  (values (truncate i1 i2)))

(defmethod divide? ((i1 integer) (i2 integer))
  (zero? (remainder i2 i1)))

(defmethod remainder ((i1 integer) (i2 integer)) (rem i1 i2))

(defmethod normalizing-factor ((i integer))
  (cond ((less? i (zero i)) (negative (one i)))
        ( T (one i))))

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;
; CRA2 (r1 r2 m1 m2) -> r  where r = r1 mod m1  and  r = r2 mod m2
;
; r1 < m1 ,  r2 < m2
; class of computation: residue
;
; Version: case that all modules are numbers !!!
;
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

(defmethod cra2 ((r1 integer) (r2 integer) (m1 integer) (m2 integer))
  (let* ((z1 (cons-residue r1 m2))
         (z2 (cons-residue* r2 m2))
         (i  (inverse (cons-residue m1 m2)))
         (i* (product i (difference z2 z1))))
    (sum r1 (product (number i*) m1))))

; ++++++++ Print-function ++++++++

(defmethod show ((i integer))
  (princ i))

;**********************************************************
;
;   float
;
;**********************************************************
```

```
; The Domain of Floating Point Numbers is a Field.


(setq *epsilon* 1.0e-5)

(defmethod zero ((f float)) 0.0)

(defmethod one ((f float)) 1.0)

(defmethod inverse ((f float)) (/ 1 f))

(defmethod quotient ((f1 float) (f2 float)) (/ f1 f2))

(defmethod equal?  ((f1 float) (f2 float))
  (< (abs (- f1 f2))  *epsilon*))

(defmethod show ((f float))
  (print f))

;**********************************************************
;
;  prime-generation
;
;**********************************************************

(defun next-prime (number)
; generate lowest prime that is greater than "number"
  (cond ((< number 2) 2)
        ((= number 2) 3)
        ( T (next-prime* number 1))))

(defun next-prime* (number n)
  (let ((prime (- (* n 6) 1)))
    (cond ((and (< number prime) (prime? prime))  prime)
          ((and (< number (+ prime 2)) (prime? (+ prime 2)))
             (+ prime 2))
          ( T (next-prime* number (+ n 1))))))

(defun prime? (p)
  (cond ((= p 3) T)
        ((zerop (rem p 3)) nil)
        ( T (prime*? p 1))))

(defun prime*? (p n)
```

```
  (let ((n1 (- (* 6 n) 1)))
    (cond ((< p (expt n1 2)) T)
          ((zerop (rem p n1)) nil)
          ( T (let ((n2 (+ n1 2)))
                (cond ((< p (expt n2 2)) T)
                      ((zerop (rem p n2)) nil)
                      ( T (prime*? p (+ n 1)))))))))

(defun next-prime-no-factor (p d)
  ; finds next prime that is no factor of d
  (let ((prime (next-prime p)))
    (cond ((zerop (rem d prime)) (next-prime-no-factor prime d))
          ( T prime))))

(defun next-integer* (i p1 p2)
  ; "next-integer" returns a list of three elements:
  ; the first element is the next possible integer for evaluation,
  ; the second and third elements are the evaluated polynomials !!!
  (let* ((i* (cond ((plusp i) (- i))
                   ((minusp i) (- (- i 1)))
                   ( T (+ i 1))))
         (p1-hom (evaluation-depth p1 i* 1))
         (p2-hom (evaluation-depth p2 i* 1)))
    (cond ((or (equal? (degree p1-hom) (degree p1))
               (equal? (degree p2-hom) (degree p2)))
                 (list i* p1-hom p2-hom))
          ( T (next-integer i* p1 p2)))))

(defun next-integer (i p1 p2)
  ; "next-integer" returns a list of three elements:
  ; the first element is the next possible integer for evaluation,
  ; the second and third elements are the evaluated polynomials !!!
  (let* ((i* (+ i 1))
         (p1-hom (evaluation-depth p1 i* 1))
         (p2-hom (evaluation-depth p2 i* 1)))
    (cond ((or (equal? (degree p1-hom) (degree p1))
               (equal? (degree p2-hom) (degree p2)))
                 (list i* p1-hom p2-hom))
           ( T (next-integer i* p1 p2)))))

 ;*********************************************************
;
; residue (computation)
;
;*********************************************************
```

```
; For computation of a lot of residues with the same module,
; one desires to store the module only once!
; The class residue supports a shared-slot for the module.

(defclass residue
  (field-element)
  ((number :accessor number :initarg :number)
   (module :accessor module :allocation :class :initarg :module)))

; ++++++++ Constructor ++++++++
;
;  see file "constructor" !!!
;

; ++++++++ Functions ++++++++

(defmethod normalizing-factor ((r residue))
  (inverse r))

(defmethod inverse ((r residue))
  (let* ((result-list (g-c-d-extended  (module r) (number r)))
         (g (first result-list))
         (b (third result-list)))
    (cond ((equal? g (one g)) (residue b (module r)))
          ( T (detect-error 'inverse r)))))

(defmethod one ((r residue))
  (cons-residue* (one (number r)) (module r)))

(defmethod zero ((r residue))
  (cons-residue* (zero (number r)) (module r)))

(defmethod negative ((r residue))
  (residue (negative (number r)) (module r)))

(defmethod sum ((r1 residue) (r2 residue))
  (let* ((r1n (number r1))
         (r2n (number r2))
         (m (module r1))
         (s (sum r1n r2n))
         (rn (cond ((greater? s (quotient m 2)) (difference s m))
                   ((less-or-equal? s (negative (quotient m 2)))
                     (sum s m))
                   ( T s))))
    (cons-residue* rn m)))
```

```
(defmethod difference ((r1 residue) (r2 residue))
  (let* ((r1n (number r1))
         (r2n (number r2))
         (m (module r1))
         (d (difference r1n r2n))
         (rn (cond ((greater? d (quotient m 2)) (difference d m))
                   ((less-or-equal? d (negative (quotient m 2)))
                     (sum d m))
                   ( T d))))
    (cons-residue* rn m)))

(defmethod product ((r1 residue) (r2 residue))
  (let ((r1n (number r1))
        (r2n (number r2)))
    (residue (product r1n r2n) (module r1))))

(defmethod equal? ((r1 residue) (r2 residue))
  (let ((r1n (number r1))
        (r2n (number r2)))
    (equal? r1n r2n)))

(defmethod less?  ((r1 residue) (r2 residue))
  (let ((r1n (number r1))
        (r2n (number r2)))
    (less? r1n r2n)))

; ++++++++ Print-function ++++++++

(defmethod show ((r residue))
  (show (number r))
  (princ '| |)
  (princ 'mod)
  (princ '| |)
  (show (module r)))

;*********************************************************
;
; residue-store (storage)
;
;*********************************************************

(defclass residue-store       ; residue-store
  (field-element)
  ((number :accessor number :initarg :number)
   (module  :accessor module  :initarg :module)))
```

```
; ++++++++ Constructor ++++++++
;
; see file "constructor" !!!


;    *************************************************************
;    *
;    *                       A T T E N T I O N
;    *                       =================
;    *
;    *       Not every element of Z/(m) is invertible.
;    *
;    *       If you call "inverse" with an element that is not
;    *       invertible the element is set to (negative (one b))
;    *       (as an indication).
;    *
;    *************************************************************

; ++++++++ Functions ++++++++

(defmethod inverse ((r residue-store))
  (let* ((result-list (g-c-d-extended  (module r) (number r)))
         (g (first result-list))
         (b (third result-list)))
    (cond ((equal? g (one g)) (cons-residue-store b (module r)))
          ( T (detect-error 'inverse r)))))

(defmethod one ((r residue-store))
  (cons-residue-store* (one (number r)) (module r)))

(defmethod zero ((r residue-store))
  (cons-residue-store* (zero (number r)) (module r)))

(defmethod sum ((r1 residue-store) (r2 residue-store))
  (let ((r1n (number r1))
        (r1m (module r1))
        (r2n (number r2))
        (r2m (module r2)))
    (cond ((equal? r1m r2m) (let* ((s (sum r1n r2n))
                                   (rn (cond ((less? s r1m) s)
                                             ( T (difference s r1m)))))
                              (cons-residue-store* rn r1m)))
          ( T (detect-error 'sum r1 r2)))))

(defmethod difference ((r1 residue-store) (r2 residue-store))
```

```
  (let ((r1n (number r1))
        (r1m (module r1))
        (r2n (number r2))
        (r2m (module r2)))
    (cond ((equal? r1m r2m) (let* ((d (difference r1n r2n))
                                   (rn (cond ((less? d (zero d))
                                              (sum d r1m))
                                             ( T d))))
                              (cons-residue-store* rn r1m)))
          ( T (detect-error 'difference r1 r2)))))

(defmethod product ((r1 residue-store) (r2 residue-store))
  (let ((r1n (number r1))
        (r1m (module r1))
        (r2n (number r2))
        (r2m (module r2)))
    (cond ((equal? r1m r2m) (cons-residue-store (product r1n r2n) r1m))
          ( T (detect-error 'product r1 r2)))))


(defmethod equal? ((r1 residue-store) (r2 residue-store))
  (let ((r1n (number r1))
        (r1m (module r1))
        (r2n (number r2))
        (r2m (module r2)))
    (cond ((equal? r1m r2m) (equal? r1n r2n))
          ( T (detect-error 'equal? r1 r2)))))

(defmethod less?  ((r1 residue-store) (r2 residue-store))
  (let ((r1n (number r1))
        (r1m (module r1))
        (r2n (number r2))
        (r2m (module r2)))
    (cond ((equal? r1m r2m) (less? r1n r2n))
          ( T (detect-error 'less? r1 r2)))))

;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;
; CRA (l) -> r  where l = (z1,...,zn)
;
; class of all zi: residue-store
; class of computation: residue
;
;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

(defmethod  cra (l)
```

```
  (let* ((z1 (first l))
         (z2 (second l))
         (r (cons-residue-store (cra2 (number z1) (number z2)
                                      (module z1) (module z2))
                  (product (module z1) (module z2)))))
    (cond ((null (cddr l)) (number r))
          ( T (cra (cons r (cddr l)))))))

; ++++++++ Print-function ++++++++

(defmethod show ((r residue-store))
  (show (number r))
  (princ '| |)
  (princ 'mod)
  (princ '| |)
  (show (module r)))

;*********************************************************
;
;  poly-sparse
;
;  Representation: sparse
;
;*********************************************************


(defclass poly-sparse
  (gcd-domain-element)
  ((exp-list :accessor exp-list :initarg :exp-list)
   (coef-list :accessor coef-list :initarg :coef-list)))


; ++++++++ Constructor ++++++++
;
;  see file "constructor" !!!
;

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
; SPECIFICATION
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;
; The Domain of polynomials is a Ring
;
; Element domains are lists.
;
; Necessary operations in domains of
```

```
; the elements in these lists:
;
;  1: Operations on lists
;
;     Then the following operations on
;     polynomials are available:
;        zero, zero?, degree, leading-coef,
;        leading-monomial, reductum
;
;  2: Availability of further operations
;        zero, one, negative, equal?, less?, sum, product
;
;     Additional operations on polynomials available then:
;        one, negative, equal?, sum, product
;
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::



(defmethod zero ((p poly-sparse))
  (cons-poly-sparse p '() '()))

(defmethod zero? ((p poly-sparse))
  (null (exp-list p)))

(defmethod degree ((p poly-sparse))
; "degree" returns the degree in the main-variable !!!
; if p is a multivariate polynomial: p in K(x1,...,xn-1)(xn) then
; "degree" returns the degree in xn  !!!
  (cond ((zero? p) -1)
        ( T (first (exp-list p)))))

(defmethod degree-depth ((p poly-sparse) d)
  ; p is represented in K(x1,...,xn-1)(xn)
  ; "degree-depth" returns the degree of p in xn-d !!!
  (cond ((zero? d) (degree p))
        ( T (let ((cl* (mapcar #'(lambda (x) (degree-depth x (- d 1)))
                       (coef-list p))))
              (reduce #'maximum cl*)))))

(defmethod leading-coef ((p poly-sparse))
; if p is a multivariate polynomial: p in K(x1,...,xn-1)(xn) then
; "leading-coef" returns a polynomial
; in K(x1,...,xn-1)
  (first (coef-list p)))
```

```
(defmethod leading-term ((p poly-sparse))
  (cons-poly-sparse p (list (degree p)) (list (leading-coef p))))

(defmethod reductum ((p poly-sparse))
  (cons-poly-sparse p (rest (exp-list p)) (rest (coef-list p))))

(defmethod content ((p poly-sparse))
  (cond ((zero? p) 0)
        ( T (reduce #'g-c-d (coef-list p)))))

(defmethod primitive-part ((p poly-sparse))
  (cond ((zero? p) p)
        ( T (let* ((c (content p))
                   (coef-list* (mapcar #'(lambda (x) (divide x c))
                                       (coef-list p))))
              (cons-poly-sparse p (exp-list p) coef-list*)))))

(defmethod negative ((p poly-sparse))
  (cons-poly-sparse p (exp-list p)
                    (mapcar 'negative (coef-list p))))

(defmethod one ((p poly-sparse))
  (cons-poly-sparse p (list (zero (degree p)))
                      (list (one (leading-coef p)))))

(defmethod equal? ((p1 poly-sparse) (p2 poly-sparse))
  (cond ((or (zero? p1) (zero? p2))  (and (zero? p1) (zero? p2)))
        ((and (equal? (degree p1) (degree p2))
              (equal? (leading-coef p1) (leading-coef p2)))
        (equal? (reductum p1) (reductum p2)))
        ( T NIL)))

(defmethod evaluation ((p poly-sparse) c)
  ; Evaluation of p at "main-variable=c"
  (let ((el (exp-list p))
        (cl (coef-list p)))
    (horner-evaluation el cl c (first cl))))

(defun horner-evaluation (el cl c pc)
  (cond ((zero? (first el)) pc)
        ( T (let* ((next-coef (cond ((and (null (rest el))
                                          (> (first el) 0))
                                       (zero (first cl)))
                                    ((= (second el) (- (first el) 1))
                                       (second cl))
                                    ( T (zero (first cl)))))
```

```
                   (el* (cond ((zero? next-coef) (cons (first el)
                                                 (cons (- (first el) 1)
                                                       (rest el))))
                        ( T el)))
                   (cl* (cond ((zero? next-coef) (cons (first cl)
                                          (cons next-coef (rest cl))))
                              ( T cl)))
                   (pc* (sum (product c pc) (second cl*))))
              (horner-evaluation (rest el*) (rest cl*) c pc*)))))

(defmethod evaluation-depth ((p poly-sparse) c d)
  ; p is represented in K(x1,...,xn-1)(xn)
  ; "evaluation-depth" returns p evaluated for xn-d=c !!!
  (cond ((zero? d) (evaluation p c))
        ( T (let ((cl* (mapcar #'(lambda (x) (evaluation-depth x c
                                                (- d 1)))
                             (coef-list p))))
              (modular-poly (leading-coef p) (exp-list p) cl*
                   NIL NIL)))))

(defmethod insert-var ((p poly-sparse))
  ; inserts a new variable as main-variable
  (cons-poly-multiv-sparse '(0) (list p)))

(defmethod insert-var ((n number))
  (cons-poly-univ-sparse '(0) (list n)))

(defmethod insert-var-depth ((p poly-sparse) d)
  (cond ((zero? d) (insert-var p))
        ( T (let ((cl* (mapcar #'(lambda (x) (insert-var-depth x
                                                (- d 1)))
                              (coef-list p))))
              (cons-poly-multiv-sparse (exp-list p) cl*)))))

(defmethod insert-var-depth ((n number) d)
  (insert-var n))

(defmethod convert-poly ((p poly-sparse) n)
  ; converts p to a polynomial in n variables !!!
  (convert-poly* p n (depth p)))

(defmethod convert-poly* ((p poly-sparse) n d)
  (cond ((equal? n d) p)
        ( T (convert-poly* (insert-var-depth p 1) n (+ d 1)))))

(defmethod monic-poly ((i integer))
```

```
  ; constructs a monic linear polynomial with constant i.
  (cons-poly-univ-sparse '(1 0) (list 1 (negative i))))

(defmethod normalizing-factor ((p poly-sparse))
  (normalizing-factor (leading-coef p)))

; A constant can also be viewed as a polynomial. Therefore
; the sum and the multiplication of a polynomial and a constant
; is commutative.
; If you call "sum" with first argument a "self-defined" constant and
; second a polynomial the function "sum" of
; "abelian-additive-semi-group" will be called !!!
; Analog with "product" !!!

(defmethod sum ((p poly-sparse) (n number))
  (sum-polynomial-constant p n))

(defmethod sum ((n number) (p poly-sparse))   ; could be omitted if
                                              ; number was
                                              ; a self-defined class
  (sum-polynomial-constant p n))

(defmethod sum ((p poly-sparse) (r residue))
  (sum-polynomial-constant p r))

(defmethod sum ((p poly-sparse) (r residue-store))
  (sum-polynomial-constant p r))

(defmethod sum-polynomial-constant ((p poly-sparse) c)
  (let* ((coef-list-r (reverse (coef-list p)))
         (exp-list-r (reverse (exp-list p)))
         (min-deg (first exp-list-r)))
    (cond ((zero? min-deg) (let* ((s (sum (first coef-list-r) c))
                                  (coef-list~ (cond ((zero? s)
                                                     (rest coef-list-r))
                                                        ( T (cons s
                                                     (rest coef-list-r)))))
                                  (exp-list~ (cond ((zero? s)
                                                     (rest exp-list-r))
                                                       ( T exp-list-r))))
                              (cons-poly-sparse p (reverse exp-list~)
                                                      (reverse coef-list~))))
           ( T (let ((exp-list~ (cons (zero (degree p)) exp-list-r))
                     (coef-list~ (cons c coef-list-r)))
                 (cons-poly-sparse p (reverse exp-list~)
                                     (reverse coef-list~)))))))
```

```
(defmethod sum ((p1 poly-sparse) (p2 poly-sparse))
  (cond ((zero? p1) p2)
        ((zero? p2) p1)
        ( T (sum* p1 p2 (zero p1)))))   ; tail-recursive

(defmethod sum* ((p1 poly-sparse) (p2 poly-sparse) (p3 poly-sparse))
  (cond ((and (zero? p1) (zero? p2))
           (cons-poly-sparse p1 (reverse (exp-list p3))
                                (reverse (coef-list p3))))
        ((zero? p1)
           (cons-poly-sparse p2 (reverse (append (reverse (exp-list p2))
                                                 (exp-list p3)))
                                (reverse (append (reverse (coef-list p2))
                                                 (coef-list p3)))))
        ((zero? p2)
           (cons-poly-sparse p1 (reverse (append (reverse (exp-list p1))
                                                 (exp-list p3)))
                                (reverse (append (reverse (coef-list p1))
                                                 (coef-list p3)))))
        ( T
           (let* ((p1* (cond ((less? (degree p1) (degree p2)) p2)
                             ( T p1)))
                  (p2* (cond ((less? (degree p1) (degree p2)) p1)
                             ( T p2)))
                  (c (find-coef-p (degree p1*) p2*))
                  (sum-of-coef (sum (leading-coef p1*) c))
                  (p3* (cond ((zero? sum-of-coef) p3)
                             ( T (cons-poly-sparse p1 (cons (degree p1*)
                                                             (exp-list p3))
                                                      (cons sum-of-coef
                                                         (coef-list p3)))))))
              (cond ((equal? (degree p1*) (degree p2*))
                        (sum* (reductum p1*) (reductum p2*) p3*))
                    ( T (sum* (reductum p1*) p2* p3*)))))))

(defmethod find-coef-p ((d integer) (p poly-sparse))

  ; c <- find-coef-p (d p)   where  "c" is the coefficient of the
  ;                                  variable with degree "d" in
  ;                                  the polynomial p.
  ;
  ; Special facts for application of "find-coef-p" in function "sum":
  ; =============
  ;
  ;     1:  p is not equal to zero because of the "cond-construction"
```

```
  ;         in function "sum*" !!
  ;     2:  d = degree p1   (see call of function "find-coef-p" !!)
  ;     3:  c = corresponding coefficient in p2
  ;     4:  degree p1 >= degree p2 !!
  ;
  ;
  ; Conclusion: "c" can only be the leading coefficient of p2 !!!!!
  ; ===========
  ;
  ; So:   If degree p1 = degree p2  ==>  c = leading coefficient p2
  ; ===      else  c = 0  !!!!!

  (cond ((less? (degree p) d) (zero (leading-coef p)))
        ( T (leading-coef p))))

(defmethod difference ((p1 poly-sparse) (n number))
  (sum p1 (negative n)))

(defmethod difference ((n number) (p1 poly-sparse))
  (sum n (negative p1)))

(defmethod product ((p1 poly-sparse) (p2 poly-sparse))
  (cond ((or (zero? p1) (zero? p2)) (zero p1))
        ((one? p1) p2)
        ((one? p2) p1)
        ( T (product* p1 p2 (zero p1)))))

(defmethod product* ((p1 poly-sparse) (p2 poly-sparse) (p3 poly-sparse))
  (cond ((zero? p1) p3)
        ( T (let ((p3* (sum p3 (product-monomial-polynomial
                                 (leading-term p1) p2))))
              (product* (reductum p1) p2 p3*)))))

(defmethod product-monomial-polynomial ((m poly-sparse) (p poly-sparse))
  (cond ((or (zero? m) (zero? p)) (zero p))
        ((one? m) p)
        ((one? p) m)
        ( T (cons-poly-sparse p (mapcar #'(lambda (x) (sum x (degree m)))
                                      (exp-list p))
                                (mapcar #'(lambda (x) (product x
                                                           (leading-coef m)))
                                      (coef-list p))))))

(defmethod product ((p poly-sparse) (n number))
  (product-polynomial-constant p n))
```

```
(defmethod product ((n number) (p poly-sparse))
  ; could be omitted if number was a self-defined class
  (product-polynomial-constant p n))

(defmethod product ((p poly-sparse) (r residue))
  (product-polynomial-constant p r))

(defmethod product ((p poly-sparse) (r residue-store))
  (product-polynomial-constant p r))

(defmethod product-polynomial-constant ((p poly-sparse) c)
  (cond ((or (zero? c) (zero? p)) (zero p))
        ((one? c) p)
        ((one? p) (cons-poly-sparse p '(0) (list c)))
        ( T (cons-poly-sparse p (exp-list p)
                                (mapcar #'(lambda (x) (product c x))
                                    (coef-list p))))))

(defmethod quotient ((p1 poly-sparse) (p2 poly-sparse))
  (first (quotient-remainder* p1 p2 (zero p1))))

(defmethod remainder ((p1 poly-sparse) (p2 poly-sparse))
  (second (quotient-remainder* p1 p2 (zero p1))))

(defmethod quotient-remainder ((p1 poly-sparse) (p2 poly-sparse))
  (quotient-remainder* p1 p2 (zero p1)))

(defmethod quotient-remainder* ((p1 poly-sparse) (p2 poly-sparse)
                                (result poly-sparse))

  ; tail-recursive
  ; the result of "quotient-remainder*" is a list with the quotient of p1
  ; and p2 as its first element and the remainder
  ; of p1 and p2 as its second element !!

  (cond ((less? (degree p1) (degree p2)) (list result p1))
        ( T (let* ((c (quotient (leading-coef p1) (leading-coef p2)))
                   (d (difference (degree p1) (degree p2)))
                   (new (cons-poly-sparse p1 (list d) (list c)))
                   (quotient~ (sum result new))
                   (rem (difference p1 (product p2 new))))
                   (quotient-remainder* rem p2 quotient~)))))

(defmethod cra2 ((p1 poly-sparse) (p2 poly-sparse)
 (m1 list) (m2 integer))
  ; where m1 is a list of numbers (r1,...rn) to be interpreted as the
```

```
  ; polynomial (x-r1)*...*(x-rn)
  ; and m2 is to be interpreted as the polynomial (x-m2) !!!
  ; See (Davenport,1988)
  (let* ((d (+ (depth p2) 1))
         (p1* (convert-poly p1 d))
         (poly-list-univ (mapcar #'monic-poly m1))
         (poly-list-mod (mapcar #'(lambda (x) (evaluation x m2))
                           poly-list-univ))
         (inv (reduce #'product (mapcar #'inverse poly-list-mod)))
         (poly-list-multiv~ (mapcar #'(lambda (x) (convert-poly x (- d 1)))
                                    poly-list-univ))
         (poly-list-multiv (mapcar #'insert-var poly-list-multiv~))
         (m (reduce #'product poly-list-multiv))
         (p1~ (convert-poly (difference p2 (evaluation-depth p1* m2 1)) d)))
     (sum p1* (product m (product inv p1~)))))

;  GCD-extended

(defmethod gcd-rs ((p1 poly-sparse) (p2 poly-sparse))
  ; computes gcd of polynomials over a field !!!
  (gcd-rs* p1 p2))

(defmethod gcd-rs* ((p1 poly-sparse) (p2 poly-sparse))
  (let ((rem (remainder p1 p2)))
    (cond ((zero? rem) p2)
          ( T (gcd-rs* p2 rem)))))

(defmethod gcd-rs-extended ((p1 poly-sparse) (p2 poly-sparse))
  ; gcd by remainder-sequence
  ; p1 and p2 are polynomials with coefficients over a field !!!
  (let* ((g (gcd-rs-extended* p1 p2 (one p1) (zero p1) (zero p2) (one p2)))
         (nf (normalizing-factor (first g))))
    (mapcar #'(lambda (x) (product nf x)) g)))

(defmethod gcd-rs-extended* ((p1 poly-sparse) (p2 poly-sparse) a1 a2 b1 b2)
  (cond ((zero? p2) (list p1 a1 a2))
        ( T (let* ((r1 (remainder p1 p2))
                   (q (quotient p1 p2))
                   (c1 (difference a1 (product b1 q)))
                   (c2 (difference a2 (product b2 q))))
              (gcd-rs-extended* p2 r1 b1 b2 c1 c2)))))

;*********************************************************
;
;  poly-univ-sparse
;
```

```
;  Univariate Polynomial
;
;  Representation: sparse
;
;*****************************************************


(defclass poly-univ-sparse
  (poly-sparse) () )

; Suggestion: New slot "domain" that stores the domain of the coefficients
;             e.g  "F" ... field
;            "R" ... ring
;                    .
;                    .
;                    .

; ++++++++ Constructor ++++++++
;
;  see file "constructor" !!!
;


; ++++++++ Functions ++++++++

(defmethod depth ((p poly-univ-sparse))      ; returns the "depth of a
  (cond ((zero? p) 0)                        ; polynomial (number of var.)
        ( T 1)))

(defmethod divide ((p1 poly-univ-sparse) (p2 poly-univ-sparse))
  ; pseudo-division
  (let* ((m (degree p1))
         (n (degree p2))
         (factor (power (leading-coef p2) (+ (- m n) 1)))
         (p1* (product p1 factor)))
     (quotient p1* p2)))

(defmethod pseudo-remainder ((p1 poly-univ-sparse) (p2 poly-univ-sparse))
  ; pseudo-remainder
  (let* ((m (degree p1))
         (n (degree p2))
         (factor (power (leading-coef p2) (+ (- m n) 1)))
         (p1* (product p1 factor)))
    (remainder p1* p2)))

(defmethod divide-over-z? ((p1 poly-univ-sparse) (p2 poly-univ-sparse))
```

```
  ; Checks whether p1 divides p2 over Z !!!
  (cond ((less? (degree p2) (degree p1)) (zero? p2))
        ((divide? (leading-coef p1) (leading-coef p2))
           (let* ((c (quotient (leading-coef p2) (leading-coef p1)))
                  (d (difference (degree p2) (degree p1)))
                  (new (cons-poly-univ-sparse (list d) (list c)))
                  (rem (difference p2 (product p1 new))))
             (divide-over-z? p1 rem)))
        (T NIL)))

(defmethod divide? ((p1 poly-univ-sparse) (p2 poly-univ-sparse))
  ; Checks whether the remainder of the division of "a" and "b" is zero !!!
  (zero? (remainder p2 p1)))

(defmethod cra2 ((p1 poly-univ-sparse) (p2 poly-univ-sparse) (m1 integer)
                 (m2 integer))
  ; p1 p2 are polynomials with coefficients over the class "residue" !!!
  (let* ((i (inverse (cons-residue m1 m2)))
         (p1* (poly-homomorphism (natural-number p1) m2))
         (i* (product i (difference p2 p1*))))
     (sum (natural-number p1) (product  (natural-number i*) m1))))

;------------ Greatest Common Divisor ----------

(defmethod poly-homomorphism ((poly poly-univ-sparse) (p integer))
  ; transforms a polynomial over Z into a polynomial over Zp !!
  (let* ((el (exp-list poly))
         (cl (mapcar #'(lambda (x) (cons-residue x p)) (coef-list poly))))
    (modular-poly poly el cl NIL NIL)))

(defun modular-poly (poly el cl el-mod cl-mod)
  ; all monomials with coefficient "0 mod p" vanish !!
  (cond ((null cl) (cons-poly-sparse poly (reverse el-mod)
                                          (reverse cl-mod)))
        ((zero? (first cl)) (modular-poly poly (rest el) (rest cl) el-mod
                                          cl-mod))
        ( T (modular-poly poly (rest el) (rest cl)
                          (cons (first el) el-mod)
                          (cons (first cl) cl-mod)))))

(defmethod natural-number ((p poly-univ-sparse))
  ; transforms a polynomial over Zp into a polynomial over Z !!
  (cons-poly-univ-sparse (exp-list p)
                         (mapcar 'number (coef-list p))))

(defmethod landau-mignotte ((p1 poly-univ-sparse) (p2 poly-univ-sparse)
```

```
                            (g integer))
  ; computes the Landau-Mignotte-bound
  (let ((d1 (degree p1))
        (d2 (degree p2))
        (n1 (/ (2-norm (coef-list p1)) (abs (leading-coef p1))))
        (n2 (/ (2-norm (coef-list p2)) (abs (leading-coef p2)))))
    (* (power 2 (min d1 d2)) g (min n1 n2))))

(defun 2-norm (l)
  ; euclidean norm
  (sqrt (apply #'+ (mapcar #'(lambda (x) (power x 2)) l))))

(defmethod greatest-common-divisor ((p1 poly-univ-sparse)
                                    (p2 poly-univ-sparse))
  (cond ((equal? p1 p2) p1)   ; preprocessing
        ((or (zero? p1) (zero? p2)) (zero p1))
        ((or (one? p1) (one? p2)) (one p1))
        ( T
          (let* ((c1 (content p1))
                 (c2 (content p2))
                 (gcd-cont (g-c-d c1 c2))
                 (pp1 (primitive-part p1))
                 (pp2 (primitive-part p2))
                 (lc1 (leading-coef pp1))
                 (lc2 (leading-coef pp2))
                 (g (g-c-d lc1 lc2))
                 (l-m (* 2 (landau-mignotte pp1 pp2 g)))
                 (prime (next-prime-no-factor 1 g)))
            (product gcd-cont (new-prime pp1 pp2 l-m prime g))))))

(defmethod new-prime ((p1 poly-univ-sparse) (p2 poly-univ-sparse)
                      l-m prime g)
  (let* ((p (next-prime-no-factor prime g))  ;next prime not dividing g !!
         (p1-mod-p (poly-homomorphism p1 p)) ;polynomial over Z is trans-
 (p2-mod-p (poly-homomorphism p2 p)) ;formed to a polynomial over Zp
         (gcd-mod-p (gcd-rs   p1-mod-p   p2-mod-p))) ;gcd mod p is computed
                                         ;by remainder-sequences
    (find-lucky-prime  p1  p2  gcd-mod-p  l-m  p  g)))

(defmethod find-lucky-prime ((p1 poly-univ-sparse) (p2 poly-univ-sparse)
                             (gcd-mod-p1 poly-univ-sparse) l-m p g)
  (setq *prime* p)
  (cond ((zero? (degree gcd-mod-p1)) (one p1))
   ( T (let ((test (prog (p* gcd-mod-p*)
                         (setq p* p)             ; initialize
                         (setq gcd-mod-p* (natural-number gcd-mod-p1))
```

```
                  loop
                  (cond ((< l-m p*) (return (list T gcd-mod-p*
                                       *prime*)))) ; check + exit
                    (let* ((p (next-prime-no-factor  *prime*  g))
                           (p1-mod-p (poly-homomorphism p1 p))
                           (p2-mod-p (poly-homomorphism p2 p))
                           (gcd-mod-p~
                           (gcd-rs   p1-mod-p   p2-mod-p)))
                      (cond ((less? (degree gcd-mod-p~)
                                    (degree gcd-mod-p*))
                                (return (list NIL gcd-mod-p~ p)))
                            ( T
                                (setq *prime* p)
                                (setq *previous* gcd-mod-p*)
                                (setq gcd-mod-p*
                                  (primitive-part
                                    (cra2
                                      (poly-homomorphism gcd-mod-p*
                                                         p*)
                                      gcd-mod-p~ p* p)))
                                (cond ((or (equal?
                                              gcd-mod-p*
                                              *previous*)
                                           (equal?
                                              gcd-mod-p*
                                              (natural-number
                                                 gcd-mod-p~)))
                                     (cond ((and (divide-over-z?
                                                    gcd-mod-p* p1)
                                                 (divide-over-z?
                                                    gcd-mod-p* p2))
                                              (return (list 'found
                                                    gcd-mod-p*))))))
                             (setq p* (product p p*)))))
                  (go loop))))   ; end of "let-initialization"
      (cond ((null (first test))
               (find-lucky-prime p1 p2 (second test) l-m
                                (third test) g))
            ((equal (first test) 'found) (second test))
            ((and (divide-over-z? (second test) p1)
                  (divide-over-z? (second test) p2))
               (second test))
            ( T (new-prime p1 p2 l-m (third test) g)))))))

; ++++++++ Print-function ++++++++
```

```
(defmethod show ((p poly-univ-sparse))
  (print 'Exponent-list)
  (terpri)
  (show (exp-list p))
  (print 'Coefficient-list)
  (terpri)
  (show (coef-list p))
  (terpri))

;**********************************************************
;
;  poly-multiv-sparse
;
;  Multivariate Polynomial
;
;  Representation: sparse
;
;**********************************************************

(defclass poly-multiv-sparse
  (poly-sparse) () )

; ++++++++ Constructor ++++++++
;
;  see file "constructor" !!!
;

(defmethod depth ((p poly-multiv-sparse))
  (cond ((zero? p) 0)
        ( T (+ 1 (depth (leading-coef p))))))

(defmethod depth ((n number)) 0)

(defmethod content ((p poly-multiv-sparse))  ;integer-content
  (reduce #'g-c-d (mapcar #'content (coef-list p))))

(defmethod primitive-part ((p poly-multiv-sparse))
;primitive-part w.r.t. integer-content
  (let ((cl* (mapcar #'primitive-part (coef-list p))))
    (cons-poly-multiv-sparse (exp-list p) cl*)))

(defmethod divide ((p1 poly-multiv-sparse) (p2 poly-multiv-sparse))
  ; pseudo-division
  (let* ((m (degree p1))
         (n (degree p2))
         (factor (power (cons-poly-sparse p2 '(0) (list (leading-coef p2)))
```

```
                        (+ (- m n) 1)))
         (p1* (product p1 factor)))
    (quotient p1* p2)))

(defmethod pseudo-remainder ((p1 poly-multiv-sparse)
                             (p2 poly-multiv-sparse))
  ; pseudo-remainder
  (let* ((m (degree p1))
         (n (degree p2))
         (factor (power (cons-poly-sparse p2 '(0) (list (leading-coef p2)))
                        (+ (- m n) 1)))
         (p1* (product p1 factor)))
    (remainder p1* p2)))

(defmethod divide-over-z? ((p1 poly-multiv-sparse) (p2 poly-multiv-sparse))
  ; Checks whether p1 divides p2 over Z !!!
  (cond ((less? (degree p2) (degree p1)) (zero? p2))
        ((divide-over-z? (leading-coef p1) (leading-coef p2))
          (let* ((c (quotient (leading-coef p2) (leading-coef p1)))
                 (d (difference (degree p2) (degree p1)))
                 (new (cons-poly-sparse p2 (list d) (list c)))
                 (rem (difference p2 (product p1 new))))
            (divide-over-z? p1 rem)))
        (T NIL)))

;------------ Greatest Common Divisor ----------

(defmethod poly-homomorphism ((poly poly-sparse) (m poly-univ-sparse))
  ; P mod m
  ; P(x1,...,xn) mod ((xn-1)-r) = P (xn-1)=r (x1,...,xn-2,xn)
  (cond ((and (one? (degree m)) (one? (leading-coef m)))
           (evaluation-depth poly (negative (reductum m)) 1))
        ( T (princ '|Function not defined|))))

(defmethod greatest-common-divisor ((p1 poly-multiv-sparse)
                                    (p2 poly-multiv-sparse))
  (cond ((and (zero? (degree p1)) (zero? (degree p2)))
           (insert-var (greatest-common-divisor1 (leading-coef p1)
                                                 (leading-coef p2))))
        ( T (greatest-common-divisor1 p1 p2))))

(defmethod greatest-common-divisor1 ((p1 poly-multiv-sparse)
                                     (p2 poly-multiv-sparse))
  (cond ((equal? p1 p2) p1)
        ((zero? p1) p2)
        ((zero? p2) p1)
```

```
        ((or (one? p1) (one? p2)) (one p1))
        ( T
          (let* ((m (sum 1 (minimum (degree-depth p1 1)
                                    (degree-depth p2 1))))
                 (c1 (content p1))
                 (c2 (content p2))
                 (gcd-cont (g-c-d c1 c2)))
            (product gcd-cont (new-number (primitive-part p1)
                                          (primitive-part p2) m 0))))))

(defmethod new-number ((p1 poly-multiv-sparse) (p2 poly-multiv-sparse) m n)
  (let* ((eval (next-integer n p1 p2))
         (r (first eval))         ;next number for evaluation !!!
         (p1-mod (second eval))
         (p2-mod (third eval))
         (gcd-mod (g-c-d  p1-mod  p2-mod)))
    (find-lucky-number  p1  p2  gcd-mod  m  r)))

(defmethod find-lucky-number ((q1 poly-multiv-sparse)
                              (q2 poly-multiv-sparse)
                              (gcd-mod-r poly-sparse)
                              (m number) (r number))
  (setq *last* r)
  (let ((test (prog (m*  r-poly  gcd-mod*)
               (setq m* 1)                   ; initialize
               (setq r-poly (list r)) ; "r-poly" is a polynomial of the
                                      ; form (x-r1)* ... *(x-rn)
                                      ; "r-poly" is stored as (r1,...rn)
               (setq gcd-mod* gcd-mod-r)
               loop
               (cond ((< m m*) (return (list T (convert-poly gcd-mod*
                                             (depth q1)) *last*))))
               (let* ((eval (next-integer *last* q1 q2))
                      (r* (first eval))
                      (q1-mod (second eval))
                      (q2-mod (third eval))
                      (gcd-mod (g-c-d  q1-mod  q2-mod)))
                 (cond ((less? (degree gcd-mod)
                               (degree gcd-mod*))
                           (return (list NIL gcd-mod r*)))
                       ( T
                         (setq *last* r*)
                         (setq *previous* gcd-mod*)
                         (setq gcd-mod* (cra2 gcd-mod* gcd-mod r-poly r*))
                         (cond ((not (equal? (depth *previous*)
                                             (depth gcd-mod*)))
```

```
                                   (setq *previous* (insert-var-depth
                                                      *previous* 1))))
                         (cond ((equal? gcd-mod* *previous*)
                                  (cond ((and (divide-over-z? gcd-mod* q1)
                                              (divide-over-z? gcd-mod* q2))
                                     (return (list 'found gcd-mod*))))))
                         (setq r-poly (cons r* r-poly))
                         (setq m* (sum m* 1)))))
               (go loop))))   ; end of "let-initialization"
    (cond ((null (first test))
             (find-lucky-number q1 q2 (second test) m (third test)))
          ((equal (first test) 'found) (second test))
          ((and (divide-over-z? (second test) q1)
                (divide-over-z? (second test) q2))
             (second test))
          ( T (new-number q1 q2 m (third test))))))

; ++++++++ Print-function ++++++++

(defmethod show ((p poly-multiv-sparse))
  (print 'Exponent-list)
  (terpri)
  (show-list (exp-list p) 10) (terpri) (terpri)
  (print 'Coefficient-list)
  (terpri)
  (show-list (coef-list p) 10)
  (terpri))

;***********************************************
;
;  Fraction
;
;***********************************************

(defclass fraction
  (field-element)
  ((numer :accessor numer :initarg :numer)
   (denom :accessor denom :initarg :denom)))

; ++++++++ Constructor ++++++++
;
; see file "constructor"
;

; ++++++++ Functions ++++++++
```

```
(defmethod inverse ((f fraction))
  (cons-fraction* (denom f)
            (numer f)))

(defmethod one ((f fraction))
  (cons-fraction* (one (numer f))
            (one (denom f))))

(defmethod zero  ((f fraction))
  (cons-fraction* (zero (numer f))
            (one (denom f))))

(defmethod negative ((f fraction))
  (cons-fraction* (negative (numer f))
            (denom f)))

(defmethod sum ((f1 fraction) (f2 fraction))
     (cond ((zero? f1) f2)
           ((zero? f2) f1)
           ( T (let* ((f1n (numer  f1))
                      (f1d (denom  f1))
                      (f2n (numer  f2))
                      (f2d (denom  f2))
                      (g (g-c-d f1d f2d))
                      (f1d* (quotient f1d g))
                      (f2d* (quotient f2d g))
                      (fn~ (sum (product f1n f2d*) (product f2n f1d*)))
                      (fd~ (product f1d f2d*)))
                  (cond ((zero? fn~) (zero f1))
                        ( T (let* ((g* (g-c-d fn~ g))
                                   (fn (quotient fn~ g*))
                                   (fd (quotient fd~ g*)))
                              (cons-fraction* fn fd))))))))

(defmethod sum ((f fraction) (n number))
  (let* ((fn (numer f))
         (fd (denom f))
         (n* (product n fd)))
    (cons-fraction (sum fn n*) fd)))

(defmethod sum ((n number) (f fraction))
  (let* ((fn (numer f))
         (fd (denom f))
         (n* (product n fd)))
    (cons-fraction (sum fn n*) fd)))
```

```
(defmethod product ((f1 fraction) (f2 fraction))
  (cond ((equal? f1 (one f1)) f2)
        ((equal? f2 (one f2)) f1)
        ((or (zero? f1) (zero? f2)) (zero f1))
        ( T (let* ((f1n (numer f1))
                   (f1d (denom f1))
                   (f2n (numer f2))
                   (f2d (denom f2))
                   (g1 (g-c-d f1n f2d))
                   (g2 (g-c-d f1d f2n))
                   (fn (product (quotient f1n g1) (quotient f2n g2)))
                   (fd (product (quotient f1d g2) (quotient f2d g1))))
               (cons-fraction* fn fd)))))

(defmethod product ((f fraction) (n number))
  (cons-fraction (product (numer f) n) (denom f)))

(defmethod product ((n number) (f fraction))
  (cons-fraction (product (numer f) n) (denom f)))

(defmethod equal? ((f1 fraction) (f2 fraction))
  (and (equal? (numer f1) (numer f2))
       (equal? (denom f1) (denom f2))))

; Additional Assumptions: availability of less? in integral domain of slot
;     numer.
; Fact:
;  New function less?
;
; The following function is not yet totally general. It assumes very
; special properties of the normalizing-factor function.
;

(defmethod less? ((f1 fraction) (f2 fraction))
  (let ((f1n (numer f1))
        (f1d (denom f1))
        (f2n (numer f2))
        (f2d (denom f2)))
    (less? (product f1n f2d) (product f2n f1d))))

; ++++++++ Print-function ++++++++

(defmethod show ((f fraction))
  (print 'Numerator)
  (terpri)
  (show (numer f))
```

```
  (terpri)
  (terpri)
  (print 'Denominator)
  (terpri)
  (show (denom f)))

;**********************************************************
;
;  constructor
;
;  provides "fast constructors" for user-defined classes
;
;**********************************************************

; As considered in (Buchberger, Loos, 1982) there are
; two natural choices of representatives for Z/(m):
;   a) {0,1,...,m-1}
;   b) { a / -m/2 < a <= m/2 }
; The functions "cons-residue-store" "cons-residue-store*" "cons-residue"
; and "cons-residue*" create representatives of type b).

(pcl::defconstructor cons-residue-store residue-store (n m)
                     :number (simplified-mod n m)
                     :module m)

; The input for the function cons-residue-store* is supposed to be
; simplified, so that n is included in the set shown in b) above !!!

(pcl::defconstructor cons-residue-store* residue-store (n m)
                     :number n
                     :module m)

(defmethod simplified-mod ((n integer) (m integer))
  (let ((n* (remainder n m)))
    (cond ((greater? n* (quotient m 2)) (difference n* m))
          ((less-or-equal? n* (negative (quotient m 2)))
             (sum n* m))
          ( T n*))))

(defmethod residue ((p1 poly-sparse) (p2 poly-sparse))
  (cons-residue* (pseudo-remainder p1 p2) p2))

(defmethod residue ((i1 integer) (i2 integer))
  (cons-residue i1 i2))

(pcl::defconstructor cons-residue residue (n m)
```

```
                     :number (simplified-mod n m)
                     :module m)

(pcl::defconstructor cons-residue* residue (n m)
                     :number n
                     :module m)

(pcl::defconstructor cons-poly-univ-sparse poly-univ-sparse (el cl)
                     :exp-list el
                     :coef-list cl)

(pcl::defconstructor cons-poly-multiv-sparse poly-multiv-sparse (el cl)
                     :exp-list el
                     :coef-list cl)

(defmethod cons-poly-sparse ((p poly-univ-sparse) el cl)
  (cons-poly-univ-sparse el cl))

(defmethod cons-poly-sparse ((p poly-multiv-sparse) el cl)
  (cons-poly-multiv-sparse el cl))

(pcl::defconstructor cons-fraction* fraction (n d)
                     :numer (product (normalizing-factor d) n)
                     :denom (product (normalizing-factor d) d))

(defmethod cons-fraction ((n gcd-domain-element) (d gcd-domain-element))
  (let* ((g (g-c-d n d))
         (n* (quotient n g))
         (d* (quotient d g)))
     (cons-fraction* n* d*)))

(defmethod cons-fraction (n d)
  (terpri)
  (princ '|Arguments are not elements of an "INTEGRAL-DOMAIN"|) (terpri)
  (princ '|First argument:|) (terpri)
  (show n) (terpri) (terpri)
  (princ '|Second argument:|) (terpri)
  (show d))
```

# References

[Bobrow,DeMichiel,Gabriel,Keene,Kiczales,Moon] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, D.A. Moon: *COMMON LISP object system specification X3J13 Document 88-002R*, SIGPLAN Notices, Volume 23, 9/88.

[Brown 71] W.S. Brown: *On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors*, JACM 18, 467-504, 1971.

[Buchberger 83] B. Buchberger, G.E. Collins, R. Loos: *Computer-Algebra – Symbolic and Algebraic Computation*, 2nd ed., Springer, 1983.

[Collins 72] G.E. Collins: *The SAC-1 Polynomial GCD and Resultant System*, Univ. of Wisconsin Comp. Sci. Dept. Tech. Report No.145, 1-93, 1972.

[Davenport 88] J.H. Davenport, Y. Siret, E. Tournier: *Computer Algebra – Systems and Algorithms for Algebraic Computation*, Academic Press/London, 1988.

[Knuth 69] D.E. Knuth: *The Art of Computer Programming*, Vol.II, Semi-numerical Algorithms, Addison-Wesley, 1969.

[Winkler 88] F. Winkler: *Computer-Algebra I*, Lecture script WS 88/89.