MAS
Modula–2 Algebra System
Interactive Usage

Computer Algebra Group
University of Passau

MAS Version 1.0[1]

[1]Document revision October 13, 1996

Abstract

MAS is an experimental computer algebra system combining imperative programming facilities with algebraic specification capabilities for design and study of algebraic algorithms. MAS contains a large library of implemented Gröbner base algorithms for nearly all algebraic structures where such methods exist. This document describes the interactive usage of MAS, the MAS language, the specification component, basic arithmetic and polynomial system libraries.

Preface

MAS (Modula–2 Algebra System) is an experimental computer algebra system combining imperative programming facilities with algebraic specification capabilities for design and study of algebraic algorithms. MAS contains a large library of implemented Gröbner base algorithms for nearly all algebraic structures where such methods exist, together with many applications which make use of these algorithms. The interactive part of MAS views mathematics in the sense of universal algebra and model theory and is in some parts influenced by category theory.

MAS combines Modula–2 program development, a LISP interpreter with a Modula–2 like language and an algebraic specification component. MAS can be used interactively, but includes access to to the comprehensive ALDES/SAC–2 and DIP algebraic algorithm libraries. MAS can also be used as ordinary Modula–2 program library. Despite of its design it can directly access numerical Modula–2 libraries.

The current implementations (version 1.0) run on IBM RS6000 / AIX, IBM–AT 386/ DOS (or compatible), IBM–AT 386/ OS2 2.x, 3.x (or compatible) and further workstations with unix tools and C compiler. The previous implementations (version 0.7 and 0.6) were running on IBM RS6000 / AIX, IBM–PC, IBM–AT / MS–DOS (or compatible), Atari 1040ST / GEM–TOS and on Commodore Amiga / Amiga–DOS. It is mostly written in the programming language Modula-2.

Major mathematical library changes of the current version 1.0 are:

- added a package for counting real roots based on Hermites method by F. Lippold,
- added a package on permutation invariant polynomials by M. Göbel,
- added an optimized Gröbner base package (including the "sugar"-method) by C. Rose,
- added a package to compute factorized Gröbner bases by J. Pfeil,
- added a logic formula representation with simplification package and real quantifier elimination package by A. Dolzmann,
- a package for involutive bases by R. Grosse-Gehling,
- Improved comprehensive Gröbner Base algorithms using factorization, condition evaluation and case elimination by M. Pesch,
- arbitrary domain polynomial system extended to a generic Gröbner base package.

Major system changes of the current version 1.0 are:

- MAS language accepts small letter key words and braces {} to denote list expressions.
- Improved error handling and user signal processing to examine long running computations.
- GNU readline for easier command line editing.
- Using Kpathsearch Library from K. Berry.
- Improved batch processing capabilities.
- Distribution now uses GNU autoconfig for easy compilation.
- Generic garbage collection support for most architectures.

Major mathematical library changes between version 0.7 and version 0.6 were:

- arbitrary domain polynomial system implemented,
- added comprehensive Gröbner base package by E. Schönfeld,
- added several new basic arithmetic packages: complex numbers, quaternion numbers, octonion numbers, finite fields,
- added package for computation in non-commutative polynomial rings of solvable type: $*$-product, left Gröbner base, two-sided Gröbner base, elements in the center,
- added a package for the computation of generators for the module of syzygies of systems of homogeneous polynomial equations and Gröbner bases for modules over polynomial rings (also available for solvable polynomial rings) by J. Phillip.
- added universal Gröbner base package by T. Belkahia.
- added d-Gröbner base and e-Gröbner base packages for Gröbner bases over the integers and univariate rational polynomial rings by W. Mark.

Major system changes between the version 0.6 and 0.7 were:

- Distribution based on Modula-2 to C translator and a C distribution which will work on 'most' workstations.
- New support for PC 386 and higher (OS2 2.0 and higher) with emx dll runtime libraries.
- New support for PC 386 and higher (DOS 5.0) with emx DOS extender (10 times faster).
- Dropped support for the Atari, Amiga and PC XT up to 286 versions. That means, that we do no more distribute executables for these systems, but if you have the maskern(el) you can get the new source code (except maskern) and compile it on your system.

- The HELP and help command has been changed to provide name ranges and more information from the procedure comments.

Major system changes between the version 0.3 and 0.6 were:

- added language extensions for specification capabilities,
- added a parser for the ALDES language and possibility for interpretation of ALDES programs,
- added an interface between the MAS language and the distributive polynomial system,
- improved symbol handling by hash tables combined with balanced trees,
- EMS support for IBM PC implementations,
- C code version of the Modula-2 programms, generated be the Modula-2 to C translator from the GMD.

Organization

This document is a description of the front–end part of the system: the interpreter. It is intended as a manual for the MAS interactive part. The syntax of the MAS language together with informal semantics and examples is discussed. Further basic arithmetic and polynomial system data structures and libraries are discussed. A document for implementors is already available under the title 'MAS Library description'. The definition modules and indexes are contained in a separate document 'MAS Specification, Definition Modules, Indexes'.

Some discussions are currently rather short but will be improved in later revisions of the document. The first chapters (up to chapter 7) are already self contained and can serve as a tutorial introduction into MAS and the algorithm libraries and data structures. The later chapters are complete, but very condensed. So you may need some knowledge of LISP, the ALDES / SAC–2 computer algebra system and Modula-2 in the more advanced chapters.

The organization of this document is as follows:

Chapter 1 gives a short introduction of MAS and shows how to use MAS for the first time.

Chapter 2 gives an tutorial overview of MAS, and discusses several important aspects of MAS, such as arithmetic, help facilities, specifications and function overloading, talking LISP and handling errors.

Chapter 3 describes the MAS language and chapter 4 describes the MAS specification component.

Chapter 5 gives an introduction into list processing and algorithm complexity.

In chapter 6 the basic arithmetic algorithms and in chapter 7 the polynomial systems are described.

Chapter 8 contains description of the available packages like Gröbner bases, ideal dimension, primary ideal decomposition, real roots of zero dimensional ideals, real root counting, real quantifier elimination, polynomial invariants, involutive bases and more.

In Chapter 9 the parser for the ALDES language and the syntax of the ALDES language are presented.

Chapter 10 summarizes system commands: display commands, pragmas, LISP commands and command line parameters.

Chapter 11 gives some information on the system components, on the internal structure, on implementation issues and on the configuration of MAS. It also provides some background information on the underlying LISP.

The appendices contain notes on the distribution and the current release of MAS and an index of this document.

Acknowledgments

Many thanks to all who made contributions or influenced this project: R. Loos, G. E. Collins and co–workers for the ALDES / SAC–2 system; I. Giese, W. Kynast and H. Czytrek for discussions in the early stages of the MAS project; M. Pesch, B. Haible, V. Weispfenning and T. Becker for feedback during the later stages of the development of MAS. In the current version also contributions from students are incorporated:

- a package on permutation invariant polynomials by M. Göbel,
- a package for counting real roots by F. Lippold,
- a "sugar"-optimized Gröbner base package by C. Rose,
- a Gröbner factorizer package by J. Pfeil,
- a logic package and extensions to the interpreter by A. Dolzmann,
- a real quantifier elimination package by A. Dolzmann,
- a package for involutive bases by R. Grosse-Gehling,
- a non-noetherian ring package by I. Bader,
- a comprehensive Gröbner base package by E. Schönfeld,
- a syzygy package by J. Phillip,
- a linear algebra library by J. Müller,
- an universal Gröbner base package by T. Belkahia,
- an d- and e-Gröbner base package by W. Mark,
- port to IBM PC, M2SDS by B. Deyle,
- port to IBM PC, Topspeed Modula–2 by H. Freibel,
- port to Atari ST, SPC by E. Reisinger,
- port to Amiga, M2Amiga by M. Rothmeier,
- a hash table symbol handling by T. Wollersberger,

- and an ALDES parser by K. Rieger.

If you find a bug in MAS, please let us know (email to `mas@alice.fmi.uni-passau.de`). Also any suggestions will be welcome. Most of the MAS system is available via anonymous ftp from internet node `alice.fmi.uni-passau.de` or on world wide web `http://alice.fmi.uni-passau.de/mas.html`.

Passau, October 13, 1996. H. Kredel[1], M. Pesch[2]

[1]University of Mannheim, L 15, 16, D-68131 Mannheim, FRG, Tel: (49,0) 621/ 292 5673, E-mail: kredel @ rz.uni-mannheim.de

[2]University of Passau, Innstraße 33, D-94030 Passau, FRG, Tel: (49,0) 851/ 509-3123, E-mail: pesch @ alice.fmi.uni-passau.de

Contents

List of Tables

Chapter 1

Introduction

MAS is a computer algebra system in contrast to symbol manipulation systems. The MAS data objects are not considered as symbolic expressions, but they are considered to be elements of algebraic structures. So working with MAS usually requires that you have to decide in which algebraic structure you want to compute. Data objects can then be converted from their external representation (character sequences) to their internal representation (lists over numbers). These internal objects can then be used as inputs to programs which are implementations of algebraic functions.

In contrast, other symbol manipulation programs such as Derive [Rich *et al.* 1988], Reduce [Hearn 1987], Maple [Geddes *et al.* 1986], Mathematica [Wolfram 1988], or Macsyma [Pavelle, Wang 1985] view their input as untyped expressions built from numbers, symbols, operators and functions. The user has then a wide variety of tools to manipulate expressions (like expanding expressions, collecting or extracting subexpressions or applying substitutions or rewrite rules to the expressions). The interpreter of the Scratchpad II [Jenks *et al.* 1984, Jenks *et al.* 1985] computer algebra system is further equipped with an algebraic 'knowledge' and tries to determine the algebraic domain to which the input expression belongs.

In MAS algebraic objects are constants for the interpreter and can only be modified using appropriate methods from the respective algebraic structures. The mathematical knowledge of MAS is only contained in the algorithm libraries (either Modula–2 programs or specification definitions).

1.1 Getting Started

In this section we give a first look on the usage of the MAS system.

To start the program, select the directory containing the MAS program and data files.

```
start:           mas or mas.exe
banner:          MAS Version r.xu
system prompt:   MAS:
system answer:   ANS:
leave with:      EXIT.
```

When MAS is started, it first initializes all available storage for list processing. Then it reads the data set 'mas.ini' from the current directory. It next issues the following prompt:

```
MAS:
```

Now MAS is waiting for a program to be typed in. The program has to be in a Modula–2 like syntax (see below). The parser reads the parts of the program: declarations and statements. A simple assignment like `a:=1.` suffices as program. From this program the parser generates a LISP S–expression which is then fed into the evaluator. The result is displayed after

```
ANS:
```

Then the read-eval-print loop is repeated.

Example 1:

```
MAS: a:=2*3.   (* this is a one statement program *)
ANS: 6.
```

By this one statement program `2*3` is evaluated and the result `6` is bound to the symbol (or variable) 'a'. Comments are enclosed in `(*` and `*)`.

The next example shows the usage of the WHILE statement.

Example 2:

```
MAS: d:=0. i:=0.    (* two one statement programs *)
ANS: 0
ANS: 0
MAS: WHILE i < 17 DO
           i:=i+1; d:=d+i*i
           END.
ANS: 1785
```

First the symbol 'd' is bound to 0 and also the symbol 'i' is bound to 0. Then comes a WHILE–loop in Modula–2 style. In its body the squares of 'i' are computed and summed to 'd'. Observe that statements have values as in LISP. However constructions like `d:=IF cond THEN ... END` are **syntactically** invalid in the MAS language.

It is possible to access compiled functions and procedures. However you **must** use the corresponding write procedures to display the object not as list.

Example 3:

```
MAS: p:=APPI().
ANS: (2 (221327762 71487876 210828714))
MAS: APWRIT(p).
0.31415926535897932384 6E1
ANS: ()
```

In this example `APPI` denotes a function from the arbitrary precision floating point library which returns the value of π in the actual precision (initially 20 decimal digits). The

computed value is in internal lisp representation and to display the value in readable form the procedure `APWRIT` must be called.

To find out which compiled procedures are accessible there is a 'help' command. With '`help(all).`' you get a list of all available functions together with their signature. During the display screen scrolling can be stopped by pressing any key (in particular <CNTRL>-S will work), output can be resumed by again pressing any key (e.g. <CNTRL>-Q). If more help support is loaded then with '`help(name).`' you get the procedure comments of all functions which have a name starting with '`name`'.

If the specification support is loaded it is possible to declare variables and to use generic assignments. If further generic parse is enabled (with `PRAGMA(GENPARSE)`) then also generic expressions are possible.

Example 4:

```
MAS: VAR f: FLOAT.          (* declare FLOAT variable *)
ANS: FLOAT.
MAS: f:="1.0E99".
ANS: "0.10000000000000000000E100"
MAS: f:= "2.0": FLOAT * f - f.
ANS: "0.10000000000000000000E100"
```

In this example MAS uses the type information of the variable 'f' to determine the correct conversion function from the string "1.0E99" to the internal representation of floating point numbers. During the evaluation the data object is tagged, so that the output routine can call the correct printing function. If `PRAGMA(GENPARSE)` was executed, then the arithmetical operators (like `*` and `-` in line 5) are associated with generic functions (in this case with `PROD` and `DIF`). The constant `2.0` from the FLOAT structure can be entered in the form `"2.0": FLOAT`. The generic operations can also be used within algorithms.

In this section we have seen that **control statements** can be entered directly from the keyboard, that **compiled functions** can be accessed and that **variable declarations** and **generic expressions** are available in MAS.

Chapter 2

Elementary Concepts

In this section we will discuss some of the key features and concepts of MAS.

Starting point for the development of MAS was the requirement for a computer algebra system with an up to date language and design which makes the existing ALDES / SAC–2 algorithm libraries available. At this time there have been about 650 algorithms in ALDES / SAC–2 and in addition I had 450 algorithms developed on top of ALDES / SAC–2. The tension of reusing existing software in an interactive environment with specification capabilities contributes most to the evolution of MAS.

The resulting view of the software has many similarities with the model theoretic view of algebra. The abstract specification capabilities are realized in a way that an interpretation in an example structure (a model) can be denoted. This means that it is not only possible to compute in term models modulo some congruence relation, but it is moreover possible to exploit an fast interpretation in some optimized (or just existing) piece of software.

The main **design concepts** are: MAS replaces the ALDES language [Loos 1976] and the FORTRAN implementation system of SAC-2 by the Modula-2 language [Wirth 1985a]. Modula-2 is well suited for the development of large program libraries; the language is powerful enough to implement all parts of a computer algebra system and the Modula–2 compilers have easy to use program development environments.

To provide an interactive calculation system a LISP interpreter is implemented in Modula–2 with full access to the library modules. LISP was chosen because it seemed to be most flexible and well understood to provide a basis for an interactive system and experimenting with generic functions. Using Modula–2 procedure types, the compiled procedures are made accessible from the LISP interpreter. To add a new compiled procedure one has to compile an interface module coded in Modula–2 with the appropriate import lists and declaration procedures and then one re–links the MAS main program. This guarantees maximum efficiency of the developed algorithms.

As mentioned above, the mathematical knowledge of MAS is only contained in the Modula–2 algorithm libraries. Thus the Modula–2 compiler with its linker serves as a knowledge acquisition system without requiring any further effort to setup knowledge–based computer algebra systems as suggested by[Calmet, Lugiez 1987].

For better usability a Modula-2 like imperative (interaction) language was defined, including a type system and function overloading capabilities. The MAS parser generates LISP

S–expressions, which are then evaluated by the LISP interpreter.

To increase expressiveness high–level specification language constructs have been included together with conditional term rewriting capabilities. They resemble facilities known from algebraic specification languages like ASL [Wirsing 1986].

The MAS language and its interpreter has no *knowledge of mathematics* and mathematical objects; however it is capable to describe (specify) and implement mathematical objects and to use libraries of implemented mathematical methods. Further the imperative programming, the conditional rewriting and function overloading concepts are separated in a clean way.

The type declaration concept equips the interpreter with the knowledge that some variables and objects have types. Applying generic functions to these typed variables or objects causes the interpreter to look for an executable function, which can work with the specific object types.

The system design also assures high portability of the software. All storage and input/output management is handled by only two library modules MASSTOR and MASBIOS. They have been first implemented for an Atari 1040 computer running TOS and GEM and are also running on other computers.

In the following sections we explain how to make efficient use of MAS: editing files, reading files, browsing definition modules. Further some basic arithmetic functions are explained and the error handling of MAS is discussed.

2.1 Edit–run–debug Cycle

In general you will not type every statement directly to the MAS terminal. It is much more convenient to use your favourable editor to prepare a sequence of statements for MAS. Then you can feed this data set into MAS, save some output and determine your next action.

Although editors are computer and operating system dependent, any implementation of MAS should provide some similar statements as discussed now for the Atari implementation.

On the Atari there are two ways to call an editor: `EDIT("data-set-name")` or via the operating system `DOS("editor-name data-set-name")`. After return from the editor the data set can be read in with `IN("data-set-name")`. Output can be saved in a file with `OUT("data-set-name")` and then the file can be closed using `SHUT("data-set-name")`.

To speedup this cycle and minimize typing you can use the following two procedures

```
VAR what: STRING.

PROCEDURE doit;
BEGIN what:="data set name";
      EDIT(what) END doit.

PROCEDURE run;
IN(what) run.
```

So typing `doit.` will call the editor and after return typing `run.` will execute the file. You can also place several procedures like 'doit' in the startup file 'MAS.INI', so that they are ready to be used. `run` will then always execute the last edited file.

The editor can also be used to browse the appropriate definition modules during editing. If the editor is programmable (like microEMACS) further improvements of this procedure are possible and are described in the section on 'getting help'.

2.2 Elementary Arithmetic

As mentioned above, the interpreter only knows about atoms and lists, that means the arithmetic operators '+, −, ∗, /' and relations '<, =, <=, >, >=' are only valid with atoms as arguments. But this is not checked, and may lead to unpredictable results, so you are *self* responsible to supply the correct arguments to the compiled subroutines.

In the next example we will discuss the computation of n factorial for different values of n. If you know that you need factorial(n) only for small values of n, such that factorial(n) < beta $= 2^{29}$, that is for $n <= 12$ you may write:

```
MAS: PROCEDURE fac(n);
     VAR   d, i: ANY;
     BEGIN d:=1; i:=n;
           WHILE i > 1 DO
                 d:=d*i; i:=i-1
                 END;
           RETURN(d)
           END fac;
     fac(6).
ANS: 720
```

For bigger values of n you must use arbitrary precision integers as provided with the ALDES/SAC-2 subroutine libraries. Here we use IPROD for multiplication (∗) and IWRITE to display the result, otherwise the internal SAC-2 list representation of arbitrary precision integers is displayed.

```
MAS: PROCEDURE fac(n);
     VAR   d, i: ANY;
     BEGIN d:=1; i:=n;
           WHILE i > 1 DO
                 d:=IPROD(d,i); i:=i-1
                 END;
           RETURN(d)
           END fac;
     j:=fac(50).
ANS: (0 373030912 375187572 439570590 300896040
        154982421 452089979 2365)
MAS: IWRITE(j).
     30414093201713378043612608166064768844377641
     568960512000000000000
```

Note that since i contains only small integers, the built-in operations can be used. The computing time for 'fac(50)' is 4 seconds on an Atari 1040 ST and 2 seconds are needed to print the result.

Question: how are the arbitrary precision integers internally represented if you know that 2^{29} is represented by the list $(0,1)$ and -2^{29} by $(0,-1)$.

It is also possible to use recursion as with LISP or Modula–2:

```
PROCEDURE fac(n);
IF n <= 1 THEN RETURN(1)
          ELSE RETURN(IPROD(n,fac(n-1))) END fac.
```

Now let's try a more difficult example: compute 50 digits of e using arbitrary precision rational numbers.

The function 'Exp' takes x and the desired precision *eps* as input parameter. It computes the taylor series of the exponential function: $\sum_{i=1,\ldots} \frac{x^i}{\mathrm{fac}(i)}$. The summation runs until $\frac{x^i}{\mathrm{fac}(i)} < \frac{eps}{2}$, that is the rest is smaller than the desired precision. Integers are converted to rational numbers by the function 'RNINT' or 'RNRED'. 'RNPROD', 'RNSUM', 'RNABS' and 'RNCOMP' denote the product, sum, absolute value and comparison of rational numbers.

```
dig:=50.
Eps:=RNRED(1,IEXP(10,dig)).

PROCEDURE Exp(x,eps);
(*Exponential function. eps is the
desired precision. *)
VAR   s, xp, i, y, p: ANY;
BEGIN
(*1*) y:=RNINT(1); s:=RNINT(1); i:=0;
      p:=RNPROD(eps,RNRED(1,2));
(*2*) REPEAT i:=i+1; xp:=RNRED(1,i);
             y:=RNPROD(y,x); y:=RNPROD(y,xp);
             s:=RNSUM(s,y)
             UNTIL RNCOMP(RNABS(y),p) <= 0;
      RETURN(s)
(*9*) END Exp.
```

The results of the computation can be displayed by the functions 'RNDWR' and 'RNWRIT'. RNWRIT writes out a rational number as 'nominator/denominator' and RNDWR writes a rational number as decimal number with specified number of digits after the decimal point.

```
one:=RNINT(1).
e:=Exp(one,Eps).

BEGIN CLOUT("AbsErr = "); RNDWR(Eps,dig); BLINES(0) END.
BEGIN CLOUT("Result = "); RNDWR(e,dig); BLINES(0) END.
BEGIN CLOUT("Result = "); RNWRIT(e); BLINES(0) END.
```

The following output is produced:

```
{0 sec} ANS: (1 1)

{20 sec} ANS: ((119769761 450433631 444044040 360650700
               406458113 17125896) (0338539520 159342123
               356614372 176392732 6300265))

AbsErr = 0.00000000000000000000000000000000
          00000000000000001

Result = 2.71828182845904523536028747135266 2
          49775724709369996-

Result = 76384051975228859737656454938414688
         9815927382313633/281001223550575979
         70862852124890231398727680000000
```

The computing time for 'Exp(1)' is 20 seconds on an Atari 1040 ST and 4 seconds are needed to print the result with 'RNDWR'.

2.3 Specification Component

As already mentioned MAS views mathematics in the sense of universal algebra and model theory and is in some parts influenced by category theory. In contrast to other computer algebra systems (like Scratchpad II [Jenks *et al.* 1985]), the MAS concept provides a clean separation of computer science and mathematical concepts. The MAS language and its interpreter has no *knowledge of mathematics* and mathematical objects; however it is capable to describe (specify) and implement mathematical objects and to use libraries of implemented mathematical methods. Further the imperative programming, the conditional rewriting and function overloading concepts are separated in a clean way. The informal semantics of the MAS language is also discussed in [Kredel 1991].

MAS includes the capability to join *specifications* and to rename sorts and operations during import of specifications. This allows both the specification of abstract objects (rings, fields), concrete objects (integers, rational numbers) and concrete objects in terms of abstract objects (integers as a model of rings). Specifications can be parameterized in the sense of λ abstraction.

The specification language constructs are discussed in detail in chapter 4. In this overview we will just describe the advantages in the interactive usage which are gained when using the specification component.

2.3.1 Example

We will only discuss some features of the interactive use of the specification component by means of the following example:

```
VAR r, s: RAT.              ANS: RAT().

r:="2222222222.777777777777777".
ANS: "22222222227777777777777777/1000000000000000".

s:=r/r.                     ANS: "1".

s:=r^0 + s - "1": RAT.      ANS: "1".
```

The first line declares the variables `r` and `s` to be of type `RAT`, that is to be rational numbers.

The second line is a so called generic assignment. Depending on the type of `r` the character string on the right hand side is read (or converted to internal form). Internally an object with type, value and descriptor information is created. This information is then used by the generic write function `WRITE(RAT)` for displaying the result in the next line.

The fourth line shows the computation of `r/r`. According to the type information of `r` the corresponding generic function is determined. Then the interpretation of the generic function in the rational numbers is determined and executed.

Finally the information on the output parameters is used to create a new typed object. This object is then bound to the variable `s` and finally it is displayed.

The last line shows the computation of the expression `r^0 + s - "1": RAT`. The term `"1": RAT` denotes a constant from the rational numbers, namely `1`. The contents of the character string are read by the generic function `READ(RAT)` and a new typed object is created. The expression `r^0` is computed by an abstract function (namely `EXP`) of the abstract `RING` implementation.

We have seen, that the usage of the specification component is easy when the specification libraries are available. However a detailed explanation of the complete truth is somewhat complex and cannot be discussed here.

2.4 Getting Help

One problem in using MAS is that information is required on what functions are available. Further the specifications of the functions are often required. MAS provides several help facilities which are discussed in this section.

There is a help command showing all accessible compiled functions, all interpreter functions and all signatures, a EXTPROCS command showing accessible external compiled functions, an SIGS command showing defined signatures of functions, a VARS command which displays defined variables, a TYPES command which displays defined types, a GENERICS command listing all generic functions, a LISTENV command to display values of variables, and a system browse facility for Modula–2 definition modules.

2.4.1 HELP and help Command

Use the `HELP.` or `help.` command to display the following list

```
Enter 'help(name[,mod])' or 'help(start,end[,mod])' to get
```

```
more help.
'Name' means the first characters of a range of names,
'start,end' means a range of names.
'[,mod]' is optional and 'mod' can be
'ModulName' = list module names of the procedures,
'all' = list all loaded procedures,
'Loaded' = list loaded procedures,
'Comment' = list procedure comments (default).
```

Use the '`help(all).`' command to display a list of all compiled functions accessible from the interpreter. The help command output is like the following:

```
List of functions and procedures:

PROCEDURE ADV(LIST; LIST,LIST)
FUNCTION  APFINT(LIST): LIST
FUNCTION  APFRN(LIST): LIST
...
PROCEDURE TIME
PROCEDURE TRACE
FUNCTION  TSGBASE(LIST,LIST,LIST): LIST

0 signatures,
10 interpreter procedures,
340 compiled functions,
83 compiled procedures
accessible.
```

From this list you can deduce the name of a function and its 'arity', that is the number of input or output parameters. For example the procedure named 'ADV' has one input and two output parameters, or the function 'APFRN' has one input parameter.

For the specification of these functions you must refer to the respective ALDES / SAC–2 or MAS documentation or to the corresponding Modula-2 definition modules. On some systems (e.g. UNIX and OS2) you can also load more help support by typing '`helpup.`'. Then the command '`help(DIRPGB).`' will display

```
Comments:

PROCEDURE DIRPGB(P,TF: LIST): LIST;
(*Distributive rational polynomials Groebner basis.
P is a list of rational polynomials in distributive
representation in r variables. PP is the Groebner basis of P.
t is the trace flag.*)
```

Use the '`help(DIRP,ModulName).`' command to display a list of all Modula-2 definition modules, which contain a procedure with name starting with `DIRP`. The help command output is like the following:

```
Module Names:

DIRPAB is in: DIPRN.md.
DIRPDA is in: DIPDEC0.md.
DIRPDF is in: DIPRN.md.
DIRPDM is in: DIPRN.md.
DIRPDR is in: DIPRN.md.
DIRPEM is in: DIPRN.md.
DIRPES is in: SYMMFU.md.
DIRPEV is in: DIPRN.md.
...
```

2.4.2 System Browser

For better online help it would be nice if the Modula–2 program development system has some browser facilities as in the Smalltalk-80 system with its class hierarchy browser [Goldberg 1981].

We have added macros and data sets for the microEMACS editor to mimic some system browsing facilities. Therefore if you use the microEMACS editor, you can browse a system file containing a procedure to module cross reference. When pressing the function key 'F3', an editor window is opened with the following contents:

```
Procedure to module cross-reference:

saci.def 10:     AADV(L: LIST; VAR AL,LP: LIST);
sacsym.def 18:   ACOMP(A,B: LIST): LIST;
sacsym.def 23:   ACOMP1(A,B: LIST): LIST;
masstor.def 44:  ADV(L: LIST; VAR a, LP: LIST);
saclist.def 10:  ADV2(L: LIST; VAR AL,BL,LP: LIST);
saclist.def 15:  ADV3(L: LIST; VAR AL1,AL2,AL3,LP: LIST);
saclist.def 20:  ADV4(L: LIST; VAR AL1,AL2,AL3,AL4,LP: LIST);
sacanf.def 10:   AFDIF(AL,BL: LIST): LIST;
sacanf.def 15:   AFINV(M,AL: LIST): LIST;
sacanf.def 21:   AFNEG(AL: LIST): LIST;
sacanf.def 26:   AFPROD(M,AL,BL: LIST): LIST;
sacanf.def 32:   AFQ(M,AL,BL: LIST): LIST;
sacanf.def 38:   AFSIGN(M,I,AL: LIST): LIST;
sacanf.def 44:   AFSUM(AL,BL: LIST): LIST;
masapf.def 67:   APABS(A: LIST): LIST;
masapf.def 73:   APCMPR(A,B: LIST): LIST;
masapf.def 12:   APCOMP(ML,EL: LIST): LIST;
masapf.def 103:  APDIFF(A,B: LIST): LIST;
masapf.def 115:  APEXP(A,NL: LIST): LIST;
masapf.def 23:   APEXPT(A: LIST): LIST;
masapf.def 38:   APFINT(N: LIST): LIST;
masapf.def 121:  APFRN(A: LIST): LIST;
...
```

The three columns have the following meaning: 1) name of the definition module containing the procedure, 2) the line number of the procedure within the definition module, 3) the procedure header with formal parameter declarations.

From the variable names it is sometimes possible to remember the meaning of the variables. If this information is not sufficient the corresponding definition module can be browsed. Place the cursor in the line with the procedure you are interested in and press function key 'F4'. Then the system browser window is replaced by a window containing the corresponding definition module. The cursor is moved to the comment of the function.

```
...
PROCEDURE APSPRE(N: LIST);
(*Arbitrary precision floating point set precision.
N is the desired precision of the floating point numbers. *)

PROCEDURE APFINT(N: LIST): LIST;
(*Arbitrary precision floating point from integer.
The integer N is converted to an arbitrary precision
floating point number. *)
...
```

So browsing the definition modules you can determine which programs you want to call. Then using the HELP function you can determine if the program is accessible from the interpreter.

2.4.3 Signatures

The signature shows more verbose and mnemonic names of the data types of the formal parameters of the procedures. Signature declarations for the most often used procedures are contained in separate files and can be loaded upon request (or automatically during startup). This information can then be displayed using the SIGS command.

The specification declarations are discussed in more detail in chapter 4.

```
List of all signatures:

APSUM:(FLOAT FLOAT) -> FLOAT.
APWRIT:(FLOAT) -> ().
GCD:(ER ER) -> ER.
IGCD:(INT INT) -> INT.
NEG:(RING) -> RING.
PROD:(RING RING) -> RING.
Q:(FIELD FIELD) -> FIELD.
QR:(ER ER ER ER) -> ().

30 signatures.
```

For example 'GCD' is a function taking two elements of an euclidean ring ('ER') as input and returns an element of an euclidean ring. 'IGCD', integer greatest common divisor, takes two integers as input and returns an integer.

2.5 Handling Errors

There is a wide variety of sources for errors:

If the **parser** detects any syntax errors he tries to skip invalid program parts, at least until he finds a period (= end of program mark) in the input source. Then he returns a quoted expression of what he was able to generate so far. So only syntactically correct programs become evaluated.

Errors during **evaluation** are reported, but execution can usually be continued if the errors are not too severe.

Errors occurring at runtime in **compiled code** are also reported. This includes errors trapped by the processor (p.e. division by zero). Whether execution is continued depends on the severity of the error. Such a case may look like:

```
5 Division by Zero
** fatal error: processor interrupt
(a)bort, (b)reak, (c)ontinue (d)ebug or <ENTER> ? <ENTER>
Trying to restart processor ...

MAS:
```

In general you should hit <ENTER> or <RETURN> and the system will take the appropriate action:

error: continue, that means take some corrective action and try to continue

fatal error: break, that means return to the top level command loop

confusion: abort, that means execute processor HALT instruction

You should not respond with a 'softer' reaction than the system would do unless you know what you are doing. But you can freely break to the top level command loop or abort the run completely.

The error handler executes the command loop as a coroutine to the main MAS program. It further installs itself into the error handler provided by the Modula–2 runtime support. By these mechanisms the error handler can catch runtime errors and most software errors. However as with any LISP system incorrect input data may cause infinite loops, which cannot be handled. Only if MAS is still producing output you may press the ESC key, which will cause a fatal error. In other cases you must rely on the reset button (or some software monitors providing hotkeys).

As you may have noticed in the above example, there is also a **debug** option. Typing 'd' will take you to the command loop of a debug processor. It is usually a restricted form of the main MAS command loop. The commands entered must be in LISP syntax to emphasize its role as emergency aid. New errors produced during debugging count higher and may very quickly lead to total program abort. Beside these considerations you can use almost all LISP functions, especially you can list variable contents or modify variables. Finally typing 'EXIT' takes you back to the error prompt.

2.6 Talking LISP

Not all LISP constructs have an equivalent in the MAS language, e.g. the 'DM' define macro is not available in the MAS language. So it can be necessary to switch to LISP by calling the `PRAGMA(LISP)` procedure which switches the parser to LISP syntax. To return to Modula style input you must use the `(PRAGMA MODULA)` procedure in LISP syntax to switch the MAS parser back on.

```
MAS:  PRAGMA(LISP).          (* switch to LISP *)
ANS:  ()
LISP: (SETQ a 5)             (* use LISP syntax now *)
ANS: 5
LISP: (PRAGMA MODULA)                (* switch to MAS *)
ANS: ()
```

A macro example for those who miss a FOR–statement:

```
PRAGMA(LISP).

(DM FOR (X)
    (LIST (QUOTE PROGN)
          (CAR  (CDR X))
          (LIST (QUOTE WHILE)
                (CAR (CDR (CDR X)))
                (LIST (QUOTE PROGN)
                      (CAR (CDR (CDR (CDR (CDR X)))))
                      (CAR (CDR (CDR (CDR X))))
                      ) ) ) )

(PRAGMA MODULA)

a:=0.
FOR( SETQ(i,0), LEQ(i,10), SETQ(i,i+1), SETQ(a,a+i*i) ).

ANS: 11
```

The syntax is similar to the syntax of the FOR–statement in C. Only the executed statement must be included in the parameter list of the FOR macro:

```
FOR(init,cond,step,stat)  ==  (FOR init cond step stat)
```

The semantics of the FOR macro is defined by the following WHILE statement:

```
init; WHILE cond DO stat; step END  ==
(PROGN init (WHILE cond (PROGN stat step)))
```

The MAS language (like Modula–2) does not allow assignments as expressions, so all parameters of the FOR macro must be calls of the SETQ function or expressions. In C the FOR statement would be:

```
FOR( i:=0, i <= 10, i:=i+1) a:=a+i*i
```

The evaluation can be traced using the `PRAGMA(TRACE)` procedure which turns the trace flag in the evaluator ON and OFF.

Chapter 3

The MAS Language

This chapter contains the MAS language description. Besides the lexical and syntactical constructs it gives information on conventions, program interpretation and input / output facilities. The specification component is discussed separately in chapter 4. It does not contain a description of the algebraic data structures and functions.

The MAS language is only a 'cover' for the MAS LISP. The language parser can be switched of and programs in pure LISP syntax can be entered as well. Moreover some MAS facilities are only accessible in LISP mode. Therefore also some LISP constructs are discussed in this chapter. The reader not interested in them may skip such parts during a first reading. Such parts are indicated by small letter type style.

The initial language definition was the PL/0 language as described in N. Wirth's book "Compilerbau" [Wirth 1985b]. This definition was extended to allow procedures with parameters and several new declarations where provided. Some polishing was done to allow the interactive use of the parser, e.g. empty programs are accepted.

In the following sections we discuss first the lexical conventions and then the language syntax.

3.1 Lexical Conventions

The 'atomic' constituents of the language are characters and tokens (character sequences with special meaning).

3.1.1 Character Set

The character set of MAS is a subset of the ASCII character set. It consists of the

digits `0123456789`

letters `aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ`

others `.,=+-*/$()␣!"#%&':;<>?@[\]^_‘{}|~`

The number of characters is denoted by χ (= 95 here).

3.1.2 Tokens

Lexical tokens of the language are:

```
#  <  >  =  <=  >=  <>
+  -  *  /  %
(  )  ,  .  ;  : { }
-> => :=
keyword  number  identifier
string  comment
```

Characters not contained in this list of tokens may only appear in strings and comments.
The keywords are:

NOT, AND, OR,
IF, THEN, ELSE, END,
WHILE, DO, TO, REPEAT, UNTIL,
PROCEDURE, BEGIN, EXPOSE,
SPECIFICATION, IMPLEMENTATION, MODEL, AXIOMS,
SORT, VAR, IMPORT, SIGNATURE, MAP, RULE, WHEN,
not, and, or,
if, then, else, end,
while, do, to, repeat, until,
procedure, begin, expose,
specification, implementation, model, axioms,
sort, var, import, signature, map, rule, when.

The meanings of most of the tokens and keywords should be 'as expected' and are discussed later. At this place we will only say some words on numbers, identifiers, strings and comments.

3.1.3 Numbers

Numbers may be only so called β–integers. The set of β–integers is defined as:

$$\mathcal{B} = \{x \in \mathbf{Z} | -\beta < x < \beta\},$$

where $\mathbf{Z}$ denotes the set of integral numbers.
The magnitude of β depends on the implementation of MAS on the actual computer. Precisely β is defined as a power of 2 such that:

$$\chi < \beta \text{ and } 4\beta \leq \gamma,$$

where χ denotes the number of characters and γ is the least number such that for all, representable integral numbers y: $|y| \leq \gamma$. On 32 bit computers $\gamma = 2^{31}$ (32 minus one sign bit). Some basic constants and ranges are summarized in table 3.1.3.
Commonly $\beta = 2^{29} = 536870912$. β–integers are also called **atoms**.
Other kinds of numbers such as floating point numbers, arbitrary precision integral numbers, rational numbers etc. are provided by library modules. These modules also supply read / parse and write / pretty print routines for the respective data types.

Name	Definition	Value
γ	integer size	$2^{31} = 2147483648$
β	$4\beta \leq \gamma$	$2^{29} = 536870912$
γ–integer	$\{n \in \mathbf{Z} \mid -\gamma < n < \gamma\}$	
β–integer	$\{n \in \mathbf{Z} \mid -\beta < n < \beta\}$	
ν	number of cells	run time dependent
atoms	β–integers	
pointers	$\{n \in \mathbf{Z} \mid \beta \leq n \leq \beta + (8\nu)\}$	
empty list	β	
SIL	β	
NIL	β	

Table 3.1: Constants and Ranges

3.1.4 Identifiers

Identifiers are used as names of variables and names of procedures. The character sequence of an identifier must start with a letter and may be followed by digits and letters. Identifiers are case sensitive, that means upper case and lower case letters are distinct. The length of identifiers is restricted by the requirement that they must fit on one input line.

Example: NIL, p123, AL9, XSH, AlongName.

Naming Conventions

Identifier names from the ALDES / SAC–2 system follow the so called 'Implementation ALDES' naming scheme. However in comments we have already introduced upper case and lower case letters according to the 'Publication ALDES' naming scheme. So it is useful to know the transliteration between both conventions. The transliterations are summarized in table 3.1.4. Several ornaments are transliterated in clockwise order.

Publication ALDES	Implementation ALDES	Meaning
a, b	AL, BL	lower case
A', B'	AP, BP	prime
A^*, B^*	AS, BS	star
$\bar{A}, \bar{B}$	AB, BB	bar
$\hat{A}, \hat{B}$	AH, BH	hat
A_0, B_1	A0, B1	subscript
$\bar{a}'_0$	ALBP0	clockwise

Table 3.2: Transliteration Scheme

Procedure and variable names from the ALDES / SAC–2 system are at most 6 characters long and follow certain further conventions. The first letters identify the data structure and the following letters are chosen according to the meaning of the identifier.

Example:

IPROD Integer Product

MPPROD Modular Polynomial Product

DIRPPR Distributive Rational Polynomial Product

3.1.5 Strings

Character sequences enclosed in double or single quotes are called strings. Within the quotes any character from the character set may appear. Quotes itself can be part of a string, if they are written twice.

Example:

`"this is a string"` denotes the string `this is a string`,
`"'"` denotes the string ',
`'"'` denotes the string ",
`"x'7""'"` denotes the string `x'7"'`

Strings are internally represented as lists of numbers (β–integers). So all list operations are applicable to strings, like concatenating, reversing.

3.1.6 Comments

Comments are sequences of characters enclosed in `(*` and `*)`. Comments may be nested, that means the comment character sequence may contain *pairs* of `(*`, `*)`.

Comments can appear everywhere except in other tokens.

3.1.7 Blanks

Blanks can appear everywhere except in numbers, identifiers, keywords or two letter tokens. Blanks must be used in some cases to separate keywords. So `ENDEND` would mean the identifier ENDEND and not two END keywords.

Characters in input lines which do not belong to the MAS character set are converted to blanks. ASCII characters like CR (return), LF (line–feed), EOL (end–of–line) are *ignored* during input form data sets.

3.2 Syntax

In this section we discuss the MAS language syntax and the meaning of the language constructs. First we give the complete syntax diagram and the list of syntax errors. Then we will concentrate the description on the procedural aspects of the MAS language. The MAS type system and generic function support will be discussed later in a separate section.

3.2.1 Syntax Diagram

The syntax definition is given in extended BNF notation. That means `name` denotes syntactic entities, `{}` denotes (possibly empty) sequences, `()` denotes required entities, `|` denotes case selection and `[]` denotes optional cases. Terminal symbols are enclosed in double quotes and productions are denoted by `=`. The syntax diagram is listed in table 3.2.1.

Observe, that a program is a (possibly empty) sequence of declarations, followed by a (possibly empty) statement followed by a period. A statement can be an assignment, a procedure call or a IF–, WHILE–, REPEAT– or BEGIN–statement. Declarations are VAR and PROCEDURE. The syntax of the language constructs for specifications is discussed in chapter 4.

The semantics of the statements are discussed next, but let us first mention the syntax errors and syntax warnings detected by the parser (see tables 3.2.1 and 3.5).

If a syntax error is detected one of the error messages is displayed followed by the actual input line where the last character read is underscored. However this last character is one character and one lexical token to far. That means the syntax error is caused by one token behind.

Error repair is limited to skipping tokens until something meaningful is found.

In case syntax errors are detected, the execution of the program is totally suppressed, that means no executable code is generated. If a syntax warning is given execution proceeds.

3.2.2 Expressions

Expressions are built from numbers, strings, identifiers and function calls. Mathematical operators defined in the MAS language are only valid on numbers (β–integers), variables having a number as value or function calls with a number as return value or on generic items (defined in chapter 4. Variables are bound to values by means of the assignment statement, which is discussed later.

So expressions are sums or differences of terms:

```
expression = ["+"|"-"] term {("+"|"-") term}
```

The operators `+` and `-` correspond to the LISP functions `ADD` and `SUB` or to the generic functions `SUM`, `NEG` and `DIF`.

Terms are themselves products, quotients or remainders of powers:

```
term = power {("*"|"/"|"%") power}
```

The operators `*`, `/` and `%` correspond to the LISP functions `MUL`, `QUOT` and `REM` or to the generic functions `PROD` and `Q`.

Note: From the syntax definition follows, that '`*`', '`/`' and '`%`' have precedence over '`+`' and '`-`'.

Powers are factors with an optional number as exponent.

```
power = factor [ ("^"|"**") number ]
```

```
program      = block"."
block        = { "VAR" identlist  ":" ident [string] ";"
               | "PROCEDURE" ident ["( "[identlist]
                                    [";" "VAR" identlist] ")"]
                                [":" ident]";" block ident ";"
               }
               statement
statement    = [ident ":=" listexpr |
               ident [ "(" [actualparms] ")" ] |
               "RETURN" [ "(" [ expression ] ")" ]
               "BEGIN" statementseq "END" |
               "IF" condition "THEN" statementseq
                             ["ELSE" statementseq] "END" |
               "WHILE" condition "DO" statementseq "END" |
               "REPEAT" statementseq "UNTIL" condition ]
listexpr     = "{" expression {"," expression} "}" | expression
identlist    = ident {"," ident}
actualparms  = listexpr {"," listexpr}
statementseq = statement {";" statement}
condition    = "NOT" condition |
               "(" condition ")" ("AND"|"OR") "(" condition ")"
               expression ("="|"#"|"<"|"<="|">"|">=") expression
expression   = ["+"|"-"] term {("+"|"-") term}
term         = power {("*"|"/"|"%") power}
power        = factor [ ("^"|"**") number ]
factor       = ident ["("[actualparms]")"] |
               number | string [ ":" typeexpr ] |
               "(" expression ")"
string       = ('"' {character} '"' | "'" {character} "'")
ident        = letter {letter|digit}
number       = digit {digit}
```

Table 3.3: MAS Syntax Diagram

1	= expected
2	type declaration expected
4	identifier expected
5	; or , expected
6	expression expected
7	) expected
8	factor expected
9	. expected
10	assignment expected
13	:= or (expected
14	statement expected
15	: expected
16	THEN expected
17	; or END expected
18	DO expected
20	relation expected
21	, or) expected
22	-> expected
23	(or identifier expected
24	(expected
25	condition expected
26	number expected
27	, or) or ; expected
28	; expected
29	END or ; expected
30	, or] expected
31	/ expected
32	== expected
33	END or ; or BEGIN expected
34	, or { expected
35	TO expected

Table 3.4: Syntax Error Messages

1	identifier expected
2	algorithm name expected

Table 3.5: Syntax Warning Messages

The operators + and - correspond to the LISP function POW or to the generic function EXP.

Factors are identifiers, function calls, numbers, strings or expressions enclosed in parenthesis.

```
factor = ident ["("[actualparms]")"] |
         number |
         string [ ":" typeexpr ] |
         "(" expression ")"
```

Strings can be accompanied by type information. This is discussed in chapter 4.

List expressions are sequences of expressions enclosed by curly braces or expressions.

```
listexpr = "{" expression {"," expression} "}" | expression
```

Function calls can be supplied by actual parameters which are sequences of list expressions:

```
actualparms = listexpr {"," listexpr}
```

Note, that function calls are distinguished from variables (identifiers) only by syntax. Function names **must** be specified with parenthesis, even if no arguments are supplied. Names without parenthesis are considered to be variable names.

Example:

`name` = variable,
`name()` = function call without argument,
`name(1)` = function call with argument '1',
`name1(name2)` = function call with variable as argument.

S–expressions

We include some information on LISP. This may be skipped during a first reading.

The main LISP language construct is the so called S–expression or symbolic expression. S–expressions are lists of objects, where the first object evaluates to a function symbol. For example the S–expression corresponding to the MAS expression `2+3` is the list `(ADD 2 3)`. The translation scheme between variables and functions in MAS and in LISP is the following:

MAS expression	S-expression
name	name
name()	(name)
name(1)	(name 1)
name1(name2)	(name1 name2)

3.2.3 Conditions

Two expressions combined by an relational operator are a condition. Further two conditions can be combined by AND or OR, or a condition can be negated by NOT:

```
condition = "NOT" condition |
            "(" condition ")" ("AND"|"OR") "(" condition ")" |
            expression ("="|"#"|"<>"|"<"|"<="|">"|">=") expression
```

The hash character (#) denotes 'not equal'. The LISP functions corresponding to the relational operators (in the same sequence as above are): `EQ`, `NE`, `NE`, `LT`, `LEQ`, `GT` and `GEQ`. Conditions evaluate to **true** or **false** depending on the expressions and the relational operators.

How are **true** and **false** defined ? The are at least two possibilities:

1. LISP like:
   In LISP systems usually **false** is defined to be NIL, and every thing which is not NIL is considered to be **true**. There is additionally a variable `T` to denote **true**.

2. ALDES / SAC–2 like:
   false is defined to be `0` (zero) and **true** is defined to be `1` (one). In the implementation of ALDES it is however also assumed, that everything $\neq$ `0` is considered to be **true**.

In MAS LISP we have implemented the first variant. But the ALDES / SAC–2 boolean routines return `0` or `1`, so some care is needed to obtain the right result from the evaluation of the condition.

However the syntax of the MAS language does not allow a function call as condition. So one is forced to write a condition with relational operator. Then one can decide to compare the function value against `0` or `NIL` or what else.

3.3 Statements

Statements are assignment, procedure call, return, begin, if, while and repeat:

```
statement = [ assignment | procedurecall |
              beginend | return |
              if | while | repeat ]
```

Observe that a statement can be empty.

3.3.1 Assignment

The assignment statement has the following syntax:

```
ident ":=" listexpr
```

The expression respectively the list expression is evaluated a nd the result is bound to the variable named 'ident'. It is not required to declare variables before they are used in an assignment statement. However variables with type information must be defined with a VAR statement.

Example:

`x:=4*7+3-29*5.` evaluates to `-114`.

The corresponding S–expression is:

```
(ASSIGN x (SUB (ADD (MUL 4 7) 3) (MUL 29 5)))
```

where ASSIGN denotes the LISP assignment function. In this case ASSIGN is equivalent to the LISP SETQ function which might be more familiar.

y:=x*x. evaluates to 12996.

The corresponding S–expression is:

```
(ASSIGN y (MUL x x))
```

3.3.2 Procedure Call

A procedure call has the following syntax:

```
ident [ "(" [actualparms] ")" ]
```

The actual parameters can be expressions as described in the section on MAS expressions. If no parameters are required, then the parenthesis can be omitted. Example:

ADD(1,3). evaluates to 4.

The corresponding S–expression is:

```
(ADD 1 3)
```

Contrary to other LISP systems, but closer to Modula–2 syntax, identifiers as statements are supposed to be procedure calls and not variable names. Schematically we have:

MAS statement	S-expression
name.	(name)
name().	(name)
name(2).	(name 2)

By this translation scheme we obtain several equivalent formulations of MAS statements. For example the S–expression

```
(ASSIGN x (ADD 1 3))
```

is generated from any of the following MAS statements:

```
ASSIGN(x,ADD(1,3)).
x:=ADD(1,3).
x:=1+3.
```

So all 3 statements are equivalent.

Further examples:

x:=IEXP(2,100). evaluates to (0 0 0 8192).

The function IEXP stands for integer exponentiation and delivers the internal representation of 2^{100}. Then x is bound to the list (0 0 0 8192).

IWRITE(x). evaluates to () and as a side effect produces 12...376.

This statement is a procedure call. First x is evaluated to the value to which it is bound (0 0 0 8192). This list is then supplied as input to the procedure IWRITE, which means integer write. Procedures always evaluate to NIL (= ()). IWRITE writes the external (decimal) representation 12...376 of (0 0 0 8192) to the terminal.

External and Internal Functions

We have to distinguish between *internal*, *external* and *user defined* functions or procedures. User defined functions are those declared with the PROCEDURE declaration in MAS.

Internal functions are the functions which are builtin LISP functions, that means functions implemented in the LISP evaluator. Especially all functions which correspond to MAS operators are builtin.

External functions are **compiled** Modula–2 procedures which are accessible from the interpreter. A list of all external functions can be displayed with the `help(all)` procedure. The output looks like the following:

```
List of all compiled functions and procedures:

PROCEDURE ADV(LIST; LIST,LIST)
FUNCTION  APFINT(LIST): LIST
FUNCTION  APFRN(LIST): LIST
...
PROCEDURE TIME
PROCEDURE TRACE
FUNCTION  TSGBASE(LIST,LIST,LIST): LIST

65 functions and 27 procedures accessible.
```

The first column contains the type `FUNCTION` or `PROCEDURE`, which indicates if a value is returned or not. In the second column follows the name of the function. Finally the type of input and output parameters is shown. Currently all displayed parameters types are `LIST`, so this gives only a hint on the number of input and output parameters.

The description of the procedures and the meaning of the parameters must be taken from the Modula–2 library definition modules. A procedure to module cross reference is contained in the data set `browse.rc`. The definition modules are contained in a folder named `HELP`. See the section on help facilities for more information on the system browser.

Bindings

The assignment statement is one way to bind a value to a variable. In MAS a variable and a procedure cannot both have the same name, that means the identifier names are unique. Caution: Other LISP systems allow variables and procedures to have the same name but to mean different objects.

MAS allows procedure names to be used as variable names. So procedure names can be bound to variables and the variable can then be used as synonym for the procedure.

Consider the example:

```
say:=IWRITE.
```

Note that no parenthesis are following the `IWRITE` name. The variable `say` is now bound to the procedure `IWRITE`. So it is valid to use it in a procedure call:

```
say(7).
```

which writes 7 to the terminal.

But don't bind a value to a procedure name, since this overwrites the procedure definition (unless you want to bind it to a different procedure body).

Example:

```
IWRITE:=foo. now
IWRITE(7). leads to a run time error:
** error: invalid function object in APPLY: foo.
```

Such a problem can be 'resolved' by the following assignment.

```
IWRITE:=QUOTE(QUOTE(IWRITE)).
```

This assures that `IWRITE` is bound to `QUOTE(IWRITE)` which evaluates to `IWRITE` when used.

One further remark on bindings is that you can produce infinite loops with them. Consider the example:

```
x:=y. assume x and y are undefined,
y:=x. y is bound to itself,
a:=y. evaluates infinitely.
```

The actual (top level) bindings can be displayed by the `DUMPV` and `LISTV` procedures.

`LISTV` lists the variable values in the syntax:

```
name:=value.
```

`value` is listed in MAS syntax as far as possible, but it is not always guaranteed that such listings can be read by the MAS parser again.

The output of `LISTV` may look like:

```
exit:=EXIT.

PROCEDURE tuwas();
EDIT(was) tuwas.

PROCEDURE run();
IN(was) run.

PROCEDURE masini();
BEGIN was:="MAS.INI"; EDIT(was) END masini.

NIL:=().
```

`DUMPV` lists the variable values in the syntax:

```
(SETQ name value)
```

`value` is listed in LISP syntax, so it is always guaranteed, that such listings can be read in LISP mode again.

The output of `DUMPV` may look like:

```
(SETQ exit EXIT)
(SETQ was (25 21 39 21 49 29 17 68 29 39))
(SETQ tuwas (LAMBDA () (EDIT was)))
(SETQ run (LAMBDA () (IN was)))
(SETQ masini (LAMBDA () (PROGN (ASSIGN was (STRING 37
            12 51 68 29 39 29)) (EDIT was))))
(SETQ NIL ())
```

VAR parameters

Usually function and procedure parameters are evaluated from left to right upon invocation. Exceptions to this rule are so called FEXPR functions or *external* Modula–2 functions with VAR parameters. VAR parameters mean that only a reference to a variable is used as parameter and not the value bound to a variable.

FEXPR functions do not evaluate their arguments at all. They are described in the section on LISP and not discussed here.

The ALDES / SAC–2 libraries contain many procedures with VAR parameters. The VAR and non VAR parameters are organized in the scheme: first all non VAR parameters, then all VAR parameters. This is respected by the MAS interpreter when external procedures are called. The user may therefore supply only variable names in places were VAR parameters appear. Upon exit from the procedure the variables are bound to the computed values.

Example:

```
PROCEDURE ADV(L: LIST; VAR a, LP: LIST);
```

This procedure expects one value input parameter `L` and two VAR parameters `a` and `LP`. `ADV` selects the first element of a list and the rest of a list:

`ADV(LIST(1,2,3),a,B).` binds `a` to `1` and `B` to `(2,3)`.

The integer quotient and remainder function is another example:

`IQR(44,7,a,b).` binds `a` to `6` and `b` to `2`.

3.3.3 Statement Sequence

In certain situations it is allowed to write sequences of statements separated by semicolons (";"). Note that since statements may be empty one can view the semicolon also as statement terminator character. The syntax of statement sequences is:

```
statement { ";" statement }
```

3.3.4 BEGIN–END Statement

In cases where no statement sequences are allowed, but only a statement, the BEGIN–END statement can be used to enclose a statement sequence. Its syntax is:

```
"BEGIN" statement-sequence "END"
```

Note: BEGIN–END does not create a new block with new local variables !

The LISP code generated from BEGIN–END is the PROGN S–expression. This implies that the value of a BEGIN–END statement is the value of the last executed statement.

Example:

```
BEGIN a:=IEXP(2,49); IWRITE(a) END.
```

The corresponding LISP expression is

```
(PROGN (ASSIGN a (IEXP 2 49) (IWRITE a))
```

On the interpreter top level the BEGIN–END statement is often useful to suppress unused output: in the above example the internal representation of 2^{49} is not written, only the external representation appears in the output stream.

3.3.5 IF Statement

The syntax of the IF statement is:

```
"IF" condition "THEN" statementseq1
              ["ELSE" statementseq2] "END"
```

When the evaluation of `condition` yields **true** then `statementseq1` is executed, otherwise `statementseq2` is executed (if present).

Example:

```
a:=1. b:=2.
IF a = b THEN CLOUT("equal")
         ELSE CLOUT("not equal") END.
```

The LISP equivalent of the IF statement is:

```
(IF (EQ a b) (CLOUT "equal")
             (CLOUT "not equal"))
```

The expression produces `not equal` as side effect.

Note: The IF statement returns the value of the statement sequence which gets evaluated.

Note: With the Modula–2 like IF statements all ambiguities of nested IF statements in Pascal are resolved.

3.3.6 WHILE Statement

The syntax of the WHILE statement is:

```
"WHILE" condition "DO" statementseq "END"
```

The condition is evaluated. When the result is **true**, the statement sequence is executed, when the result is **false** then the evaluation of the WHILE statement is finished. After execution of the statement sequence the WHILE statement is again evaluated.

Note: The WHILE statement returns the value of the last executed statement sequence or NIL if the first evaluation of condition is **false**.

Example:

```
i:=0. a:=0.
WHILE i < 17 DO i:=i+1; a:=a+i*i END.
```

The WHILE statement evaluates to `1785`, the sum of the squares of the numbers from 0 to 17.

3.3.7 REPEAT Statement

The syntax of the REPEAT statement is:

```
"REPEAT" statementseq "UNTIL" condition
```

First the statement sequence is executed, then the condition is evaluated. When the result is **true**, the execution of the REPEAT statement is finished. When the result is **false**, the REPEAT statement is again executed.

Note: The REPEAT statement returns the value of the last executed statement sequence.

Example:

```
i:=0. a:=1.
REPEAT i:=i+1; a:=a*2 UNTIL i > 5.
```

The REPEAT statement evaluates to `64`, the 6–th power of 2.

3.4 Declarations

Elementary declarations can be

```
{ variabledecl | proceduredecl }
```

In this place we will only discuss elementary declarations, the specification declarations will be described in a later section.

3.4.1 VAR Declaration

The syntax of the VAR declaration is:

```
"VAR" identifierlist ":" identifier [ string ]
```

The VAR declaration is used to define global or local variables. In general it is not required to declare variables in MAS. But certain usages (support type information, generic function arguments) require defined variables. The `identifier` in the above definition denotes an arbitrary name. The meaning of `string` is explained in chapter 4.

The `identifierlist` has the following syntax:

```
identifier { "," identifier }
```

This is a sequence of identifiers separated by commas.

3.4.2 PROCEDURE Declaration

The syntax of the PROCEDURE declaration is:

```
"PROCEDURE" ident1
            [ "(" [identlist] [";" "VAR" identlist] ")" ]
            [ ":" ident2 ] ";"
            block ident1 ";"
```

With this declaration it is possible to define a new procedure, which can be used afterwards in the same way as the builtin procedures. `ident1` denotes the name of the procedure, it must be repeated at the end of the procedure declaration.

`[ "(" [identlist] [";" "VAR" identlist] ")" ]` denotes the so called **formal parameter list**. The formal parameter list is a list of identifiers separated by commas, followed optionaly by a VAR parameter list.

When the procedure is called (used, invoked) the so called **actual parameters** are evaluated and bound to the formal parameters and are then accessible within the procedure body. The numbers of actual and formal parameters must be equal at runtime, otherwise an error occurs. The formal parameters should be pair wise disjoint, otherwise only the last (that is the right most) actual parameter is bound to the formal parameter. The VAR parameters are not evaluated and must be identifiers. Uppon return from a procedure with VAR parameters, the local values of these parameters are bound to the actual parameter symbols.

The `[ ":" ident2 ]` construct specifies the type of the return value. However this information is not used further by the interpreter. `block` denotes a sequence of declarations followed by a statement.

Results computed within the body of a procedure can be returned to the caller by function values. The return values can be specified *explicitly* be the RETURN statement or *implicitly* as value of the last evaluated statement. Several return values can only be returned as a single list.

The syntax of the RETURN statement is:

```
"RETURN" [ "(" [ expression ] ")" ]
```

The expression is evaluated and the result is returned to the caller. The execution of statement sequences or iteration statements is suspended after evaluation of a RETURN statement.

The generated code for the procedure declaration is

```
(DE ident1 (identifierlist) block)
```

`DE` stands for 'Define–Expr–function', and means that the actual parameters are evaluated before they are bound to the formal parameters. 'fexpr' functions, which do not evaluate their arguments, and 'macros' are discussed in the section on LISP. They cannot be defined in the MAS language, but only in LISP.

The following examples define a function, that squares its argument, in several ways:

```
PROCEDURE sqr(a);
RETURN(a*a) sqr.
```

The square of 'a' is explicit returned with a RETURN statement.

```
PROCEDURE sqr(a);
VAR b: ANY;
b:=a*a sqr.
```

A local variable 'b' is declared to be of ANY type. The square of 'a' is bound to 'b'. Since the assignment statement returns a value, this value is then returned by the function.

```
PROCEDURE sqr(a);
VAR b: ANY;
BEGIN b:=a*a END sqr.
```

The assignment statement can also be enclosed in a BEGIN–END statement.

```
PROCEDURE sqr(a): LIST;
BEGIN RETURN(a*a) END sqr.
```

This is almost in Modula–2 syntax except the missing type specification of the formal parameter 'a'.

Scoping Rules

The scope of a variable is the 'area' within the variable is 'visible'. The visibility can be determined *statically* from the program text, or *dynamically* during execution of the program. The MAS parser uses statical scoping and the MAS interpreter uses dynamical scoping.

Variables which are defined on the top level interpreter are *global* variables. *Local* variables are those which are only visible in a procedure body. But there is a case, when a variable is defined within a procedure, but used within a further procedure called from the first procedure. These variables are called *fluid* variables. The values of fluid variables depend on the actual (run–time) environment of the procedure.

For MAS the following scoping rules apply:

1. Variables which are defined in the procedure header, (formal parameters) are *local* variables.
2. Variables defined with the VAR declaration after the procedure header are *local* variables.
3. For *undeclared* variables the following cases apply:
   (a) If there does not exist a global variable or a local variable in a calling procedure in the textual scope with the same name, then the variable is declared as *local*. In this case a VAR declaration is generated by the parser and a warning message is issued.
   (b) If there exists a global variable or a local variable in a calling procedure in the textual scope with the same name, then the variable is *fluid*. Care has to be taken in the case when a procedure containing fluid variables is transfered (assigned to a global variable or returned as value) outside of its lexical block. In this case the dynamical scoping of the interpreter is used.

Note: Declare variables to ensure the correct usage of them.

Note: Mutual recursive procedures will need a dummy forward reference to be in the textual scope when they are used.

Procedure Variables

In a procedure declaration the name of the procedure becomes a variable, which is bound to the S–expression corresponding to the procedure body. In other words

```
(DE name (formals) body)
```

is equivalent to

```
(ASSIGN name (LAMBDA (formals) body))
```

where LAMBDA is a tag to denote procedure bodies.

By this definition it is possible to use procedures as input to other procedures or to assign procedures to variables. Compiled procedures are not defined in this way and have no associated LAMBDA expressions. However the evaluation mechanism takes care that also compiled procedures can be used as parameters.

Example: Definition of a function `apply`, which applies its first argument to its second argument.

```
PROCEDURE apply(f,x); RETURN(f(x)) apply.

apply(INEG,4).          --> -4
x:=INEG. apply(x,4).    --> -4
```

A definition of a function `mapcar`, which applies its first argument to each list element of its second argument is discussed in the section on list processing.

This completes the discussion of the basic MAS language. In the next section we discuss input and output. Although it is not defined in the language it is included here since it's understanding is required to write MAS programs.

3.5 Input and Output

The facilities for reading and writing data are described in this section.

The concept for input and output in MAS is based on streams. Streams are continuous sequences of characters. A stream is called *open*, when it is connected to a physical device. MAS allows maximal 25 streams to be open at a time.

From these streams at any time two are the *current* streams. That means all read operations go to the current (or actual) input stream and all write operations go to the current output stream. It is possible to switch between open streams and a switch to a non open stream implies a open operation for that stream.

From the beginning of the execution the current input stream is connected to the terminal (the keyboard) and the current output stream is also connected to the terminal (the screen). These two streams are always open. Further the terminal never becomes full during a write operation and never becomes empty during a read operation.

One exception to the concept of current streams is the error stream. It is always connected to the terminal and can not be switched. So all error messages appear on the terminal and response to them is expected from the terminal.

The streams are moreover managed like a stack. If an open input stream becomes empty, then the next open input stream becomes automatically the current input stream. If an open output stream becomes full, then the next open output stream is used. 'Last stream' means the stream which was current before the last stream switch operation.

The stream switching functions are:

IN("stream") the current input stream is switched to the stream 'stream',

OUT("stream") the current output stream is switched to the stream 'stream',

SHUT("stream") the specified stream is closed.

The 'stream' name may be prefixed by a 'device name' to specify non–disk data sets:

CON: is the terminal

WIN: is a window (not yet implemented)

RAM: is an internal memory stream, 'RAM–disk'

GRA: is a graphic window (not yet implemented)

NUL: is a dummy stream to suppress output, always empty on input, never full on output,

Other 'device names' are passed to the operating system and are usually interpreted as disk data sets.

Notes on the usage of the IN function. Consider the following two MAS inputs:

```
a)  IN("x.y"). statement.
```

and

```
b)  BEGIN IN("x.y"); statement END.
```

In a) the IN function switches the actual stream to 'x.y'. The contents of this stream are evaluated, then the statement is executed.

In b) the BEGIN–END statement is parsed and evaluated. The IN function switches the actual stream to 'x.y'. But now the statement is executed and afterwards the contents of the stream 'x.y' are executed.

In other words the IN function does only a switch to the next stream. If MAS is already executing some statements it finishes first and afterwards takes the next input from the new stream.

The following two functions are for output:

BLINES(i)	writes i blank lines to the output stream
CLOUT("string")	writes this string to the output stream

Example:

We will write a function, which puts a character string to the input stream.

```
PROCEDURE cltis(S);
(*Character list to input stream. S is a character list.
S is transferred to the input stream. *)
BEGIN
(*1*) SHUT("RAM:help"); OUT("RAM:help");
      CLOUT(S);
      SHUT("RAM:help"); IN("RAM:help");
(*9*) END cltis.
```

An utility stream 'RAM:help' is opened for output. Then the string is written to this stream and finally the current input stream is switched to 'RAM:help'.

As an application we discuss a function to convert a string to an integer. (A detailed description of integers will be given later.)

```
PROCEDURE s2i(S);
(*Character list to integer. S is a character list. S is
converted to an integer and the result is returned.*)
BEGIN
(*1*) cltis(CCONC(S," "));
      RETURN(IREAD());
(*9*) END s2i.
```

The string 'S' is put to the input stream, then 'IREAD' reads an integer from the current input stream. Note that a blank must be appended with the CCONC function to the string to stop 'IREAD' from requesting more digits from the terminal, or whatever stream is open. With this function it is now more convenient to write

```
a:=s2i("12345678901234567890").
```

instead of

```
a:=IREAD().  12345678901234567890
```

3.5.1 Stream Summary

A list of all streams can be obtained by the `BIOS` function. Its output look like the following:

```
Summary of stream IO

Name     temp.out,
Output,  Byte-IO 11, Line-IO 14, Lmarg 0, Rmarg 79, Size 79.
Name     MAS.INI,
Closed,  Byte-IO 11, Line-IO 14, Lmarg 0, Rmarg 79, Size 79.
Name     CON:,
Output,  Byte-IO 25, Line-IO 13, Lmarg 0, Rmarg 79, Size 79.
Name     CON:,
Input,   Byte-IO 3, Line-IO 0, Lmarg 0, Rmarg 79, Size 79.

4 Files used.
```

The first line contains the name of the stream. The second line contains information on the status of the stream `Closed` etc. Next the number of bytes and lines transferred to or received from the respective stream are displayed. Finally the left margin `Lmarg`, the right margin `Rmarg` and line length `Size` are shown.

3.5.2 Operating System

On some computers the MAS system provides access to the operating system. Most important is the possibility to call an editor from within MAS. The two functions `DOS` and `EDIT` are summarized as follows:

DOS("prog parms")	Calls the program 'prog' with the parameters 'parms'. The meaning of the string depends on the operating system.
EDIT("data set name")	Edits the specified data set. The editor is expected to be 'EDITOR.PRG' on the current directory. The editor on disk is 'microEMACS 3.9'. The data set name string is prefixed by the string " @MAS.RC " which specifies the startup file for EMACS to be 'MAS.RC'.

Examples:

List the directory on an IBM PC:

```
PROCEDURE dir;
DOS("c:\command.com /c dir/p") dir.
```

Call the PC command interpreter:

```
PROCEDURE command;
DOS("c:\command.com") command.
```

Chapter 4

Specification Component

In this chapter we discuss the specification capabilities of MAS. In the first section we give an overview over the design considerations. Then we define the syntax of the respective language constructs and then we discuss the semantics of the constructs.

4.1 Overview

MAS views mathematics in the sense of universal algebra and model theory and is in some parts influenced by category theory. In contrast to other computer algebra systems (like Scratchpad II [Jenks *et al.* 1985]), the MAS concept provides a clean separation of computer science and mathematical concepts. The MAS language and its interpreter has no *knowledge of mathematics* and mathematical objects; however it is capable to describe (specify) and implement mathematical objects and to use libraries of implemented mathematical methods. Further the imperative programming, the conditional rewriting and function overloading concepts are separated in a clean way. The denotational semantics of the MAS language is discussed in [Kredel 1991].

MAS includes the capability to join *specifications* and to rename sorts and operations during import of specifications. This allows both the specification of abstract objects (rings, fields), concrete objects (integers, rational numbers) and concrete objects in terms of abstract objects (integers as a model of rings). Specifications can be parameterized in the sense of λ abstraction.

The *semantics* of a specification can be described either by implementations, axioms or models. The *implementation* part describes (imperative) procedures and data representations.

The *axioms* part describes conditional rewrite rules which define a reduction relation on the term algebra generated by the sorts and operations of the specification. The semantics is therefor the class of models of the term algebra modulo the (congruence) relation. Currently there are no facilities to solve conditional equations.

The *model* part describes the association between abstract specifications (like rings) and concrete specifications (like integers). The semantics is the interpretation of the (abstract) function in the model. Operations in models can be compiled functions, user defined imperative functions or term rewrite rules. The function overloading capabilities are realized

by this concept. Dynamic abstract objects like finite fields can be handled by a descriptor concept.

Evaluation of functional terms is as follows: If there is a model in which the function has an interpretation and a condition on the parameters is fulfilled, then the interpretation of the function in this model is applied to the interpretation (values) of the arguments. If there is an imperative procedure, then the procedure body is evaluated in the procedure context. If the unification with the left hand side of a rewrite rule is possible and the associated condition evaluates to true, then the right hand side of the rewrite rule is evaluated. Otherwise the functional term is left unchanged.

In contrast to functional programming languages (like SML [Appel *et al.* 1988]) which implement typed lambda calculus the types of operations are not deduced from the program text but must be explicitly defined in the specification of an operation, in a variable declaration or in a typed string expression.

A weak point in the current MAS design is that the language is only interpreted. This is actualy not a handicap in execution speed since compiled libraries can be used, but in a too weak semantic analysis of the specifications. This means that certain errors in the specifications are only detected during actual evaluation of an expression.

4.2 Syntax

To precisely define the syntax we first specify the syntactic domains and then give the EBNF definition of the language. Note that we use the terms 'function' and 'procedure' interchangeably throughout the rest of the text.

4.2.1 Syntax Diagram

The syntax definition is given in extended BNF notation. That means `name` denotes non–termial symbols, `{}` denotes (possibly empty) sequences, `()` denotes required entities, `|` denotes case selection and `[]` denotes optional cases. Terminal symbols are enclosed in double quotes and productions are denoted by `=`. The syntax diagramms are listed in table 4.1.

Observe that a program is a (possibly empty) sequence of declarations, followed by a (possibly empty) statement followed by a period. A statement can be an assignment, a procedure call or a IF–, WHILE–, REPEAT–, BEGIN–statement or EXPOSE–statement. Declarations are VAR and PROCEDURE; Unit specifications are SPECIFICATION, IMPLEMENTATION, MODEL, AXIOMS, IMPORT, SORT, SIGNATURE, MAP and RULE. The syntax of statements and expressions has already been discussed in chapter 3. Context conditions have also been discussed there.

4.3 Unit Declarations

A collection of denotations which belong to the same algebraic structure is called a **unit**. A unit consists of at most one SPECIFICATION construct which defines the (algebraic) language of the structure. Optionally several constructs may accompany a specification

```
program      = topblock "."
topblock     = { ( unitspec | var | proc | expose ) ";" }
               statement
block        = { ( var | proc ) ";" } statement
unitspec     = { spec | implement | model | axioms }
spec         = "SPECIFICATION" header ";"
                { ( sort | import | sig ) ";" } "END" ident
implement    = "IMPLEMENTATION" header ";"
                { ( sort | import | var | proc ) ";" }
                statement "END" ident
model        = "MODEL" header ";"
                { ( sort | import | map ) ";" } "END" ident
axioms       = "AXIOMS" header ";"
                { ( sort | import | rule ) ";" }  "END" ident
sort         = "SORT" identlist
import       = "IMPORT" header [ renamings ]
sig          = "SIGNATURE" ident [ "(" [identlist] ")" ]
                                 [ ":" "(" [identlist]")" ]
var          = "VAR" identlist  ":" typeexpr
proc         = "PROCEDURE" ident ["("[identlist ]")"]
                           [":" ident]";" block ident
map          = "MAP" header "->" header [ "WHEN" header ]
rule         = "RULE" expression "=>" expression
                       [ "WHEN" condition ]
typeexpr     = header [ string ]
header       = ident [ "(" [identlist] ")" ]
renamings    = "[" { ident "/" ident ";" } "]"
expose       = "EXPOSE" ident [ "(" [actualparms] ")" ]
```

Table 4.1: Specification Syntax Diagram

which define the semantics of the algebraic object: IMPLEMENTATION, MODEL and AXIOMS.

The pair SPECIFICATION, IMPLEMENTATION is similar to the pair DEFINITION MODULE, IMPLEMENTATION MODULE in Modula–2. The semantics for functions is fixpoint semantics of λ–terms which are given by (imperative) procedures contained in the IMPLEMENTATION construct.

The pair SPECIFICATION, AXIOMS is similar to constructs from algebraic specification languages. The semantics of functions is given by a term model modulo a congruence relation defined by the rewrite rules defined in the `AXIOMS` construct.

The pair SPECIFICATION, MODEL is as far as I know unique to MAS. The semantics of functions is given by mappings which associate an interpretation function according to certain types of arguments.

When all constituents of an unit have been defined, the unit must be exposed (with the EXPOSE construct) to make the functions available for use in expressions.

We turn now to a more detailed discussion of these constructs.

4.4 Specifications

The specification part defines the (algebraic) language of an algebraic structure.

The syntax of the SPECIFICATION declaration is:

```
    spec = "SPECIFICATION" header ";"
           { ( sort | import | sig ) ";" } "END" ident1
  header = ident1 [ "(" [identlist] ")" ]
```

The identifier `ident1` defines the name of the specification and of the unit. The specifications can be parameterized with formal parameters given by `identlist`. The semantics is λ–abstraction of a specification.

4.4.1 SORT Declaration

The syntax of the SORT declaration is:

```
    sort = "SORT" identlist
```

The `SORT` declaration reserves the names in `identlist` for use as sorts.

4.4.2 IMPORT Declaration

The syntax of the IMPORT declaration is:

```
    import = "IMPORT" header [ renamings ]
  renamings = "[" { ident "/" ident ";" } "]"
```

The IMPORT declaration includes an already defined specification, named by the identifier in the header, into the actual specification. Since several specifications can be imported it is possible to join specifications.

The actual specification is therefore extended by the (old) specification. During import it is possible to rename functions and sorts in the imported specification. The ident after the '/' must be defined in the imported specification and is given the new name before the '/'.

4.4.3 SIGNATURE Declaration

The syntax of the SIGNATURE declaration is:

```
sig = "SIGNATURE" ident [ "(" [identlist] ")" ]
                        [ ":" "(" [identlist]")" ]
```

The SIGNATURE declaration defines new function names (ident). Together with the input and output parameter sorts named by the identifieres in identlist.

4.4.4 Example Specification

These constructs allow both the specification of abstract objects (rings, fields), concrete objects (integers, rational numbers) and concrete objects in terms of abstract objects (integers as a model of rings).

The specification of a concrete item like the rational numbers could be as follows:

```
SPECIFICATION RATIONAL;
(*Rational numbers specification. *)
(*1*) SORT RAT, INT, atom;
(*2*) SIGNATURE RNWRITE (RAT)     ;
      SIGNATURE RNDRD   (RAT)     : RAT;
(*3*) SIGNATURE RNone   ()        : RAT;
      SIGNATURE RNzero  ()        : RAT;
(*4*) SIGNATURE RNPROD  (RAT,RAT) : RAT;
      SIGNATURE RNSUM   (RAT,RAT) : RAT;
      SIGNATURE RNDIF   (RAT,RAT) : RAT;
      SIGNATURE RNNEG   (RAT)     : RAT;
      SIGNATURE RNINV   (RAT)     : RAT;
      SIGNATURE RNQ     (RAT,RAT) : RAT;
(*5*) SIGNATURE RNINT   (INT)     : RAT;
      SIGNATURE RNprec  (atom)    ;
(*9*) END RATIONAL.
```

In this specification the sorts RAT, INT and atom are defined. Then the input and output parameter sorts of various functions are defined.

The most general unit is an object which specifies the communication (input / output) operations of objects.

```
SPECIFICATION OBJECT;
(*Object specification. *)
(*1*) SORT obj;
(*2*) SIGNATURE READ     (obj) : obj;
      SIGNATURE WRITE    (obj) ;
(*3*) SIGNATURE DECREAD  (obj) : obj;
      SIGNATURE DECWRITE (obj) ;
(*4*) SIGNATURE DEFAULT  (obj) : obj;
      SIGNATURE COERCE   (obj) : obj;
(*9*) END OBJECT.
```

Abstract specifications can be build from smaller pieces. For example (commutative) fields can be defined in terms of two abelian groups, which are themselves build from abelian monoids (which extend objects).

```
SPECIFICATION AMONO;
(*Abelian monoid specification. *)
(*1*) IMPORT OBJECT[ amono/obj ];
(*2*) SIGNATURE ZERO (amono)       : amono;
(*3*) SIGNATURE SUM  (amono,amono) : amono;
(*9*) END AMONO.

SPECIFICATION AGROUP;
(*Abelian group specification. *)
(*1*) IMPORT AMONO[ ag/amono ];
(*2*) SIGNATURE DIF  (ag,ag) : ag;
      SIGNATURE NEG  (ag)    : ag;
(*9*) END AGROUP.

SPECIFICATION FIELD;
(*Field specification joining two abelian groups. *)
(*1*) IMPORT AGROUP[ field/ag ];
      IMPORT AGROUP[ field/ag, ONE/ZERO, PROD/SUM,
                     REZIP/NEG, Q/DIF ];
(*9*) END FIELD.
```

The renamings are used to write one abelian group 'multiplicatively', like `PROD` for `SUM`.

Using the field specification one could derive an alternative definition of the rational number specification.

```
SPECIFICATION RATIONAL;
(*Rational numbers specification using the abstract
field specification. *)
(*1*) SORT INT, atom;
(*2*) IMPORT FIELD[ RAT/field,
                    RNDRD/READ, RNWRITE/WRITE,
                    RNone/ONE, RNzero/ZERO,
                    RNSUM/SUM, RNNEG/NEG, RNDIF/DIF,
                    RNPROD/PROD, RNQ/RECIP, RNQ/Q ];
```

```
(*3*) SIGNATURE RNINT  (INT): RAT;
      SIGNATURE RNprec (atom);
(*9*) END RATIONAL.
```

Note that some unique functions for rational numbers must be specified separately.

4.5 Implementations

The *implementation* part describes (imperative) procedures and data representations.

The syntax of the IMPLEMENTATION declaration is:

```
implement = "IMPLEMENTATION" header ";"
            { ( sort | import | var | proc ) ";" }
            statement "END" ident1
header = ident1 [ "(" [identlist] ")" ]
```

The identifier `ident1` defines the name of the specification and of the unit. The specifications can be parameterized with formal parameters given by `identlist`.

A statement can be a BEGIN–statement and is executed during the exposition of the unit. The Modula–2 library functions exist a priori and can be accessed without further implementation definitions.

An implementation defines a closed environment for the contained variable and procedure declarations (so called closures).

4.5.1 SORT Declaration

The syntax of the SORT declaration is:

```
sort = "SORT" identlist
```

The `SORT` declaration reserves the names in `identlist` for use as sorts.

4.5.2 IMPORT Declaration

The syntax of the IMPORT declaration is:

```
import = "IMPORT" header [ renamings ]
renamings = "[" { ident "/" ident ";" } "]"
```

The `IMPORT` declaration makes the sorts and operations of a specification locally available. Its semantics correspond to the `EXPOSE` statement.

4.5.3 VAR Declaration

The syntax of the VAR declaration is the same as already discussed in chapter 3:

```
var          = "VAR" identlist  ":" typeexpr
typeexpr     = header [ string ]
```

The `VAR` declaration reserves names for local variables and associates type information with them.

4.5.4 PROCEDURE Declaration

The syntax of the PROCEDURE declaration is the same as already discussed in chapter 3:

```
proc         = "PROCEDURE" ident ["("[identlist ]")"]
                           [":" ident]";" block ident
block        = { ( var | proc ) ";" } statement
```

The `PROCEDURE` declaration defines the imperative (or functional) implementation of a procedure. A `block` can contain further declarations and a statement.

4.5.5 Example Implementation

The implementations can be used to define concrete procedures, abstract procedures or as extension to some existing library functions. The imperative language constructs (like assignments and loops) are fairly standard and are not discussed here.

In case of the rational number unit just some gaps left by the library functions need to be filled.

```
IMPLEMENTATION RATIONAL;
      VAR s: atom;
(*1*) PROCEDURE RNone();
      BEGIN RETURN(RNINT(1)) END RNone;
(*2*) PROCEDURE RNzero();
      BEGIN RETURN(RNINT(0)) END RNzero;
(*3*) PROCEDURE RNWRITE(a);
      BEGIN IF s < 0 THEN RNWRIT(a) ELSE RNDWR(a,s) END;
            END RNWRITE;
(*4*) PROCEDURE RNprec(a);
      BEGIN s:=a END RNprec;
(*8*) BEGIN
            s:=-1;
(*9*) END RATIONAL.
```

Here `RNWRITE` is defined for convenience and internally switches between the two rational number write functions `RNWRIT` and `RNDWR` according to the local precision variable `s`.

Abstract functions are those which use function names of abstract specifications to implement something. For example in an ring one could have an abstract exponentiation function `EXP`.

```
IMPLEMENTATION RING;
(*1*) PROCEDURE EXP(X,n);
      VAR   x: ring; VAR   i: atom;
      BEGIN
      (*1*) IF n <= 0 THEN x:=ONE(X); RETURN(x) END;
      (*3*) i:=n; x:=X;
            WHILE i > 1 DO i:=i-1;
                  x:=PROD(x,X) END;
            RETURN(x)
      (*9*) END EXP;
(*9*) END RING.
```

Here `ONE` and `PROD` denote (abstract) functions from the ring. The operators `<=`, `>` and `-` are used on atoms (integers k in the range $-2^{29} = \beta < k < \beta = 2^{29}$).

4.6 Models

The *model* part describes the association between abstract specifications (like rings) and concrete specifications (like integers).

The syntax of the IMPLEMENTATION declaration is:

```
 model = "MODEL" header ";"
         { ( sort | import | map ) ";" } "END" ident1
header = ident1 [ "(" [identlist] ")" ]
```

The identifier `ident1` defines the name of the specification and of the unit. The specifications can be parametrized with formal parameters given by `identlist`.

Operations / functions in models can be compiled functions, user defined imperative functions or term rewrite rules. Dynamic abstract objects like finite fields can be handled by a descriptor concept. The descriptor can then specify the characteristic of the field.

4.6.1 SORT Declaration

The syntax of the SORT declaration is:

```
sort = "SORT" identlist
```

The `SORT` declaration reserves the names in `identlist` for use as sorts.

4.6.2 IMPORT Declaration

The syntax of the IMPORT declaration is:

```
   import = "IMPORT" header [ renamings ]
renamings = "[" { ident "/" ident ";" } "]"
```

The `IMPORT` declaration makes the sorts and operations of a specification locally available. Its semantics correspond to the `EXPOSE` statement.

4.6.3 MAP Declaration

The syntax of the MAP declaration is:

```
map = "MAP" header1 "->" header2 [ "WHEN" header3 ]
```

The `MAP` declaration defines the interpretation of an abstract operation (named by the identifier in `header1`) In the parameter list of an abstract function the sort names of a model are specified.

In the parameter list of a concrete function (named by the identifier in `header2`) the two selectors `VAL` and `DESC` can appear. The i–th `VAL` selects the value of the i–th abstract function parameter. The i–th `DESC` selects the descriptor of the i–th abstract function parameter. Descriptors are only sketched in the sequel.

Conditional interpretation can be expressed by a `WHEN` clause following the real function. `header3` defines the name and parameters of a LISP condition. In the condition the `VAL` and `DESC` selectors can be used.

Observe that the model interpretation can be viewed as function overloading. The abstract functions are sometimes also called generic functions.

4.6.4 Example Model

On the left hand side in a `MAP` clause appears the abstract function name with sort names as parameters. On the right hand side after the '`->`' stands the concrete function name with `VAL` and `DESC` selector parameters.

```
MODEL FIELD;
(*Rational numbers are a model for fields. *)
(*1*) IMPORT RATIONAL;
(*2*) MAP READ(RAT)       -> RNDRD();
      MAP WRITE(RAT)      -> RNWRITE(VAL);
(*3*) MAP ONE(RAT)        -> RNone();
      MAP ZERO(RAT)       -> RNzero();
(*4*) MAP PROD(RAT,RAT)   -> RNPROD(VAL,VAL);
      MAP SUM(RAT,RAT)    -> RNSUM(VAL,VAL);
      MAP DIF(RAT,RAT)    -> RNDIF(VAL,VAL);
      MAP NEG(RAT)        -> RNNEG(VAL);
      MAP Q(RAT,RAT)      -> RNQ(VAL,VAL);
      MAP REZIP(RAT)      -> RNINV(VAL);
(*9*) END FIELD.
```

This reads as follows: the product function `PROD` is interpreted in the model of rational numbers (two rational numbers as parameters `RAT`) as the concrete function `RNPROD` (from the abstract parameters the values are to be taken according to the `VAL` selectors).

An example using descriptors and conditional interpretation is as follows.

```
MODEL FIELD;
(*Modular integers are a model for fields. *)
      ...
```

```
(*4*) MAP PROD(MI,MI)   -> MIPROD(DESC,VAL,VAL)
                        WHEN EQ(DESC,DESC);
      ...
(*9*) END FIELD.
```

`MI` denotes the modular integer $\mathbf{Z}/_{(p)}$ sort. `MIPROD` denotes the modular integer product where the first parameter is the modulus p selected by `DESC`. The `WHEN` clause specifies that only numbers from the same finite field are to be multiplied (that is their descriptors must be equal (`EQ`)). Descriptors can be specified in `VAR` declarations provided the specifications have defined them.

4.7 Axioms

The *axioms* part describes conditional rewrite rules.

The syntax of the AXIOM declaration is:

```
axioms = "AXIOMS" header ";"
         { ( sort | import | rule ) ";" }  "END" ident1
header = ident1 [ "(" [identlist] ")" ]
```

The identifier `ident1` defines the name of the specification and of the unit. The specifications can be parametrized with formal parameters given by `identlist`.

4.7.1 SORT Declaration

The syntax of the SORT declaration is:

```
sort = "SORT" identlist
```

The `SORT` declaration reserves the names in `identlist` for use as sorts.

4.7.2 IMPORT Declaration

The syntax of the IMPORT declaration is:

```
   import = "IMPORT" header [ renamings ]
renamings = "[" { ident "/" ident ";" } "]"
```

The `IMPORT` declaration makes the sorts and operations of a specification locally available. Its semantics correspond to the `EXPOSE` statement.

4.7.3 RULE Declaration

The syntax of the MAP declaration is:

```
rule = "RULE" expression1 "=>" expression2
       [ "WHEN" condition ]
```

The `RULE` declaration defines a rewrite rule. The meaning is as follows: if the left hand side of a rule (`expression1` above) can be unified with the expression under consideration, then the variables in the right hand side (`expression2` above) are substituted according to the unification. Then the right hand side replaces the actual expression.

The rules define a reduction relation on the term algebra generated by the sorts and operations of the specification.

Variables need not be declared and are assumed to be universally quantified and unbound.

Conditional rewriting can be expressed by a `WHEN` clause following the right hand side of the rewrite rule (`condition` above). The condition is evaluated with the variables substituted according to the actual unification of the left hand side.

Currently there are no facilities to solve conditional equations since there is no back tracking of unsuccessful rewritings.

There are also no provisions to check if the rewrite system is confluent or Noetherian.

4.7.4 Example Axioms

The Peano structure can be specified as follows:

```
SPECIFICATION PEANO;
(*Peano structure specification. *)
(*1*) SORT nat, bool;
(*2*) SIGNATURE null  ()        : nat;
      SIGNATURE one   ()        : nat;
   (* SIGNATURE succ  (nat)     : nat; *)
      SIGNATURE add   (nat,nat) : nat;
      SIGNATURE prod  (nat,nat) : nat;
(*3*) SIGNATURE equal (nat,nat) : bool;
(*9*) END PEANO.
```

Observe that the `succ` (successor) constructor need not be defined in the specification, since no rule has `succ` on the first functional term in the right hand side expression. The Peano axioms can then be coded as rewrite rules as follows:

```
AXIOMS PEANO;
(*Axioms for Peano system. *)
      IMPORT PROPLOG;
      RULE one()                  => succ(null());
(*1*) RULE equal(X,X)             => TRUE();
      RULE equal(succ(X),null())  => FALSE();
      RULE equal(null(),succ(X))  => FALSE();
(*2*) RULE equal(succ(X),succ(Y)) => equal(X,Y);
(*3*) RULE add(X,null())          => X;
      RULE add(null(),X)          => X;
(*4*) RULE add(X,succ(Y))         => succ(add(X,Y));
(*5*) RULE prod(X,null())         => null();
      RULE prod(null(),X)         => null();
(*6*) RULE prod(X,succ(Y))        => add(prod(X,Y),X);
(*9*) END PEANO.
```

PROPLOG denotes a propositional logic specification not listed here. There the constant functions TRUE() and FALSE() are defined. During unification the variables X and Y are bound in the left hand side and then substituted in the right hand side.

4.8 EXPOSE Statement

Once this specification apparatus has been setup one wants to see how it works and what benefits are obtained. The language constructs discussed so far modify only the declaration data base. To access the defined functions they must first be exposed, that means they must be made visible.

The syntax of the EXPOSE statement is:

```
expose = "EXPOSE" ident [ "(" [actualparms] ")" ]
```

The identifier ident defines the name of the unit. The actual parameters are given by actualparms and can be any MAS expression which evaluate to an item meaningful in the unit.

During the exposition of an implementation part the actual environment is used to define the procedure closures. This implies that the order of the exposition is important. So only models which have already been exposed are visible in an abstract implementation.

For example the earlier defined units can be exposed as follows:

```
EXPOSE RATIONAL.
EXPOSE PEANO.
EXPOSE FIELD.
```

From then on the functions like PROD can be used in expressions or top level statements and procedures.

4.9 Operator Overloading

For convenience of the users the MAS parser can be instructed to generate generic function names for the arithmetic operators. However some care is needed since then also a specification of the atoms structure is required to access the built–in primitive arithmetic.

This feature of the parser is set by the pragma switch GENPARSE. It activates / inactivates the generic code generation of the parser. The operators correspond to the following functions: + to SUM, - to DIF or NEG, * to PROD, / to Q and ^ to EXP. See also the section on PRAGMAS 10.2.

4.10 Expression Evaluation

We turn now to the evaluation of arbitrary expressions. Expressions are transformed to functional terms by the parser. The evaluation of functional terms is defined as follows:

1. If there is a model in which the function has an interpretation and the WHEN–condition on the parameters is fulfilled, then the interpretation of the function in this model is applied to the interpretation (values) of the arguments.
2. If there is an imperative procedure, then the procedure body is evaluated in the procedure context.
3. If the unification with the left hand side of a rewrite rule is possible and the associated condition evaluates to true, then the right hand side of the rewrite rule is evaluated.
4. Otherwise the functional term is left unchanged.

Let us step through the following examples:

```
VAR r, s: RAT.              ANS: RAT().

r:="2222222222.77777777777777".
ANS: "22222222227777777777777777/1000000000000000".

s:=r/r.                     ANS: "1".

s:=r^0 + s - "1": RAT.      ANS: "1".
```

The first line declares the variables `r` and `s` to be of type `RAT`, that is to be rational numbers. The second line is a so called generic assignment. Depending on the type of `r` the character string on the right hand side is read (or converted to internal form). Recall that the interpretation of `READ(RAT)` was defined as `RNDRD()` which reads a rational number in decimal representation.

Internally an object with type, value and descriptor information is created. This information is then used by the generic write function `WRITE(RAT)` for displaying the result in the next line.

The fourth line shows the computation of `r/r`. According to the type information of `r` the corresponding generic function `Q(RAT,RAT)` is determined. Then `RNQ(VAL,VAL)` is computed where the values of the data objects are substituted. Finally the information on the output parameters of `RNQ` namely `RAT` is used to create a new typed object. This object is then bound to the variable `s` and finally it is displayed.

The last line shows the computation of the expression `r^0 + s - "1": RAT`. The term `"1": RAT` denotes a constant from the rational numbers, namely `1`. The contents of the character string are read by the generic function `READ(RAT)` and a new typed object is created. Note further that `r^0` is computed by an abstract function (namely `EXP`) of the abstract `RING` implementation. Then the computation proceeds as expected.

A final example for the use of term algebras which explains itself:

```
x:=one().          ANS: succ(null()).
x:=add(x,x).       ANS: succ(succ(null())).
x:=prod(x,x).      ANS: succ(succ(succ(succ(null())))).
```

This concludes the discussion of the language constructs used in the specification component.

Chapter 5

List Processing

In this section we give an introduction to list processing.

Definition: An **atom** is a β–integer. A **list** is a finite sequence of atoms and / or lists. Atoms and lists are **objects**.

In MAS there is a second kind of lists which can also contain symbols (variables).

Definition: An **S–expression** is a list where the first element is a symbol or the first element is an S–expression. S–expressions may contain further symbols.

Empty lists are denoted by the variable `NIL`. Other lists are denoted by their elements separated by commas and enclosed in parenthesis. E. g. `(1,a)` denotes the list of the two elements `1` and `a`.

The size of lists is only limited by the available computer memory.

5.1 List Construction

One way to construct a list is by the `LIST` function. `LIST` can be called by any number of arguments and returns a list of its evaluated arguments.

Example:

```
x:=LIST(1,2,3,4).          --> (1,2,3,4)
y:=LIST(1,LIST(2,3),4).    --> (1,(2,3),4)
```

A second function to construct a list is the `COMP` function (COMPosition). Its first argument is an object and the second argument is a list. The object is added as first element to the list.

Example:

```
a:=COMP(0,x).              --> (0,1,2,3,4)
b:=COMP(3,NIL).            --> (3) == LIST(3)
```

The length of a list can be determined by the `LENGTH` function.

Two lists can be combined to one list by the `CONC` and `CCONC` functions (CONCatenation, Constructive CONCatenation). CONC and CCONC take two lists as arguments and

return the concatenation of the inputs. CONC modifies the first list to produce the concatenated list and CCONC produces a copy of the first input.

Example:

```
a:=CCONC(x,y).          --> (1,2,3,4,1,(2,3),4)
b:=CONC(x,y).           --> (1,2,3,4,1,(2,3),4), but x=b
```

5.2 List Destruction

The basic parts of lists are their first element and the rest list without the first element. The functions `FIRST` and `RED` (for REDuctum = rest) access these parts.

Example: (with the same lists as before)

```
a:=FIRST(y).            --> 1
b:=RED(y).              --> ((2,3),4)
FIRST(RED(y)).          --> (2,3)
```

The procedure `ADV` combines the FIRST and RED function. ADV takes a list as first argument; it returns in the second argument the first element of the list and in the third argument the rest of the list.

Example:

```
ADV(y,a,b).             --> (), now a=1, b=((2,3),4)
a:=a.                   --> 1
b:=b.                   --> ((2,3),4)
```

When a new list is constructed while an old list is processed, the new list is in most cases in the wrong order. So there is a need to reverse lists. There are two functions to accomplish this: `INV` (INVerse) and `CINV` (Constructive INVerse).

INV modifies the existing list where CINV constructs a new list.

Example: (with the same lists as before)

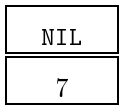
```
a:=CINV(y).             --> a=(4,(2,3),1) y=(1,(2,3),4)
a:=INV(y).              --> a=(4,(2,3),1) y=(1) !!!
```

5.3 List Diagrams

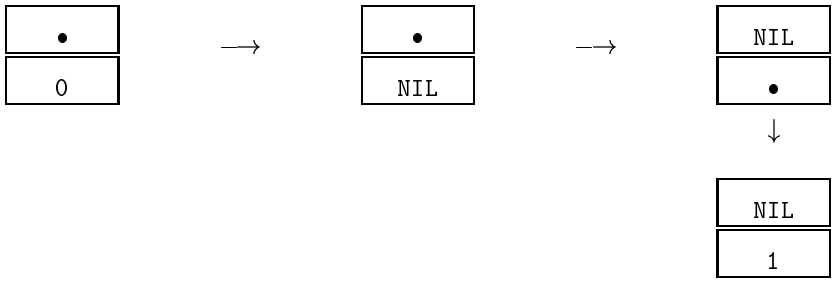
Lists are stored in the computer in memory cells which have a first field and a reductum field. These memory cells are often represented as boxes with two fields:

The FIRST field contains objects. The RED field contains the address (called pointer) of the next cell of the list or `NIL` to denote the end of the list.

Example: `LIST(7).` can be represented as:



`LIST(0,NIL,LIST(1)).` would have the following representation:

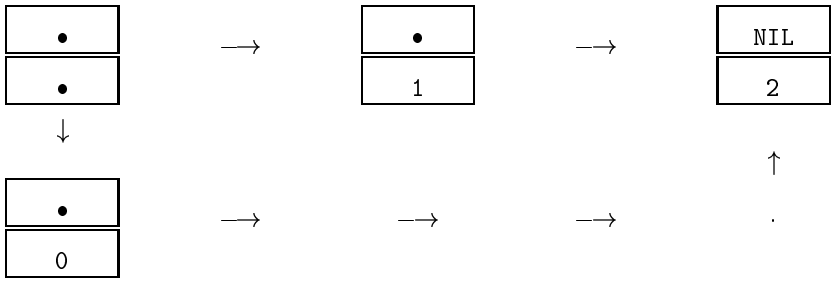


The pointers are shown as '•' and '⟶'. '•' indicates that the field is occupied by a pointer and '⟶' 'points' to the next cell of the list.

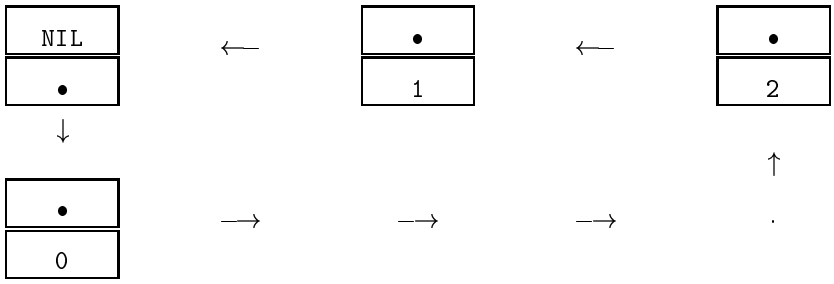
Note that in this representation the cells used to store a list can overlap with other cells:

```
a:=LIST(2).                      --> (2)
b:=COMP(COMP(0,a),COMP(1,a)).    --> ((0,2),1,2)
```

The list `b` contains two pointers to the list `a`:



Some care is needed when such lists are reversed. If it is done with the `INV` function, then a pointer circle may be constructed:



And if this list is printed one would obtain infinite output:

```
INV(b).    -->  (2,1,(0,2,1,(0,2, ... infinitely
```

In such situations it may be required to use the `CINV` function to reverse a list.

5.4 Exercises

1. Construct a list of the numbers $1, \ldots, 100$.
2. Compute the sum of all numbers in this list.
3. Construct an 'infinite' list.
4. Write a function which 'maps' a function to the elements of a list and returns the list of function results.

Solution to exercise 1:

```
i:=1. w:=NIL.
WHILE i <= 100 DO w:=COMP(i,w); i:=i+1 END.
w:=w.   --> (100, 99, ..., 2, 1)
```

In the WHILE loop the list `w` is composed from the numbers. The variable `i` runs from 1 to 100. The list `w` is in the descending order. To get the list in ascending order use `w:=INV(w).` to reverse the list.

Solution to exercise 2:

```
s:=0. u:=w.
WHILE u <> NIL DO ADV(u, j,u); s:=s+j END.
s:=s.   --> 5050
u:=u.   --> ()
```

In the WHILE loop each element of `u` is accessed and summed in the variable `s`. The sum must be below β since `+` is used as sum operator.

Solution to exercise 3:

```
w:=NIL. i:=0.
WHILE i = 0 DO w:=COMP(i,w) END.
-->
** Garbage Collection ... nnn cells, 3 sec.
** ... GC completed.
...
** fatal error: Garbage Collection: too few cells reclaimed.
(a)bort, (b)reak, (c)continue, (d)ebug, <ENTER> ? break
ARG: w:=NIL.
```

In this example the WHILE loop will never terminate since the variable `i` is not incremented within the loop. When the available cell space is consumed, a garbage collection takes place. This means that unused cells (garbage) are searched to become available for list processing again. After a while no more or to few free cells are found and the program encounters a fatal error. Typing `b` for break or `<ENTER>` returns to the top level interpreter loop and the `ARG:` prompt is displayed. Finally empty the list by `w:=NIL.` to free the cells for next use.

Solution to exercise 4:

```
PROCEDURE mapcar(f,x);
(*Map the function f to the elements of
the list x. *)
VAR   y, e, r: ANY;
BEGIN
(*1*) y:=NIL; r:=x;
(*2*) WHILE r # NIL DO
            ADV(r, e, r);
            e:=f(e); y:=COMP(e,y)
            END;
      y:=INV(y); RETURN(y);
(*9*) END mapcar.
a:=LIST(1,2,3,4,5).
ANS: (1 2 3 4 5)
p:=mapcar(INEG,a).
ANS: (-1 -2 -3 -4 -5)
```

Mapcar takes as arguments a function and a list and applies the function to all elements of the list. In the statement `e:=f(e)` the parameter f is used as function. The so constructed new list is returned. In our example the function `INEG`, i.e. integer negation is applied to `(1 2 3 4 5)` producing the list `(-1 -2 -3 -4 -5)` of the negated integers.

5.5 Strings

As already mentioned, character strings are internally represented as lists of β–integers. To make such number lists again visible as character sequence the procedure `CLOUT` can be used (Character List OUT). CLOUT writes the output to the actual output stream.

Example:

```
a:="123".            --> (1,2,3)
CLOUT(a).            --> 123
CLOUT("abc").        --> abc
```

By this list representation of strings it is possible to use all list processing functions also on strings.

Example:

```
CLOUT(INV("abc")).   --> cba
```

The string `"abc"` is first converted to the list `(11,12,13)`. Then this list is inverted and the resulting list is written to the output stream.

5.6 Exercises

1. Write functions `LEFT` and `RIGHT`, which return the left respective right part of a string (list).

`LEFT(A,i)` returns the i left objects of the list A.

`RIGHT(A,i)` returns the objects of the list A starting from $i+1$.

So `CONC( LEFT(S,i), RIGHT(S,i) ) = S` holds.

2. Write a function `SUBSTR` which returns a sub–string of a string.

   `SUBSTR(A,i,j)` returns a list of j objects of the list A starting from $i+1$.

3. Write a function `INDEX` which determines the position of one string within another string.

   `INDEX(a,B)` returns the position of the list a in the list B if a occurs in B, otherwise 0 is returned.

Solution to exercise 1:

```
PROCEDURE left(S,i);
(*Return the left i elements of the list S.*)
VAR   s, SP, T, k: LIST;
BEGIN
(*1*) SP:=S; T:=NIL; k:=0;
      WHILE (k < i) AND (SP # NIL) DO k:=k+1;
            ADV(SP,s,SP); T:=COMP(s,T) END;
      T:=INV(T); RETURN(T);
(*9*) END left.
```

To obtain the left part of a list it is necessary to copy the first i elements.

```
PROCEDURE right(S,i);
(*Return the right elements of the list S,
starting from position i+1.*)
VAR   SP, k: LIST;
BEGIN
(*1*) SP:=S; k:=0;
      WHILE (k < i) AND (SP # NIL) DO k:=k+1;
            SP:=RED(SP) END;
      RETURN(SP);
(*9*) END right.
```

The right part of a list is just the respective reductum of the list. No copying is required.

Solution to exercise 2:

```
PROCEDURE sublist(S,i,j);
(*Return j elements of the list S,
starting from position i+1.*)
RETURN(left(right(S,i),j)) sublist.

a:=LIST(1,2,3,4,5,6,7,8,9,0).

left(a,2).      --> (1,2)
```

```
right(a,7).      --> (8,9,0)

sublist(a,3,4). --> (4,5,6,7)
```

The substring (list) function can be built by combination of the LEFT and RIGHT functions.

5.7 Complexity

In this section we introduce the basic concepts of algorithm complexity. The concepts are applied to the list processing algorithms in the next section.

An important point in the discussion on algorithms is the time and the space used to compute a given problem. Time and space are 'costs' for the usage of an algorithm.

The definition for the computing time cost function is as follows:

Definition: Let I_A denote the set of inputs for an algorithm A and let $\mathbf{R}$ be the real numbers. Then $t_A : I_A \mapsto \mathbf{R}$ denotes the computing time function for algorithm A. I.e. $t_A(i)$ $i \in I_A$ gives the computing time of algorithm A for input i. When A is clear from the context, we will shortly write I for I_A and t for t_A.

The space cost function c_A is defined analog.

Since the costs may vary for different inputs it is convenient to distinguish minimal, average and maximal cost functions:

Definition: Let I_A denote the set of inputs for an algorithm A and let $n = |I_A|$. Then

1. the *minimal* computing time function is $t^- = \min\{t(i) : \ i \in I\}$,
2. the *average* expected computing time function is $t^* = \frac{1}{n} \sum_{i \in I} t(i)$,
3. the *maximal* computing time function is $t^+ = \max\{t(i) : \ i \in I\}$.

The following relation holds by definition:

$$t^- \leq t^* \leq t^+$$

If $t^- = t^+$ we denote the computing time function by t.

Computing time and memory space usage depend on the data representation, the algorithm and on the actual computer and its disks etc. The computer type is of minor interest, since the same algorithm may be implemented on different machines and we can expect that the computing time varies by a constant factor from one machine to another.

How can computing time and space usage be measured independent of a particular computer architecture ?

The elementary data type in computer algebra is the list. Therefore it seems naturally to measure 'time' as function of the list length of an object. Memory usage can be measured in terms of cell usage, which is also a function of list length.

In the following we will assume that operations on atoms cost 1 time unit and that no cells are consumed during such an operation. The time cost for the retrieval of an atom from the FIRST field of a cell will be included in the cost for the operation on the atom. Also

the time to store an atom after an operation will be included in the cost for an operation. List construction with COMP costs one cell. The list length of an object a will be denoted by $L(a)$.

For 'simple' algorithms, like those for list processing or elementary arithmetic, the minimal and maximal computing time coincide. But for other algorithms the times can vary by magnitudes. E.g. polynomials may have many zero coefficients. Then the maximal computing time must take into account that *possibly* all coefficients are non zero whereas the minimal computing time must take into account that only a certain number of coefficients is really non zero.

In complicated cases where the maximum computing time is to bad and the average computing time is not known, also the real computing time on a specific machine is taken to get an idea of the complexity of the algorithm.

When the computing time functions are determined constant factors or lower order terms are usually of minor interest. The big O notation allows to 'forget' about such details. $O(f(n))$ means, that there exists a constant c such that $O(f(n)) < c \cdot f(n)$ for all large n. E.g. $7 \cdot L(a) \sim O(L(a))$ or $L(a) + 3 \sim O(L(a))$. But constant and non constant exponents are of interest $L(a)^3$, $L(a)^n$ or $L(a)^{L(b)}$.

5.8 Algorithms

In this section we give a more formal summary of list processing functions. We include also information on the complexity of the algorithms (see section 5.7). The exposition follows [Loos 1976].

Let $\mathcal{A} = \{x \in \mathbf{Z} : \ |x| < \beta\}$ be the set of atoms,
$\mathcal{L} = \{x \in \mathbf{Z} : \ \beta \leq |x| \leq \beta + \nu\}$ be the set of lists (pointers to cells),
$\mathcal{O} = \mathcal{A} \cup \mathcal{L}$ be the set of objects and $\mathcal{L}_i$ the set of lists of length $\geq i$.

The left column contains the algorithm specification. In the right column the type specification of the algorithm inputs and outputs and the algorithm function are described. Further the algorithm complexity is discussed.

$a \leftarrow FIRST(A)$	$a \in \mathcal{O}, A \in \mathcal{L}_1$. a is the first element of the non empty list A. One cell is accessed, so the computing time is $t = 1$, $c = 0$.
$A' \leftarrow RED(A)$	$A' \in \mathcal{L}, A \in \mathcal{L}_1$. A' is the reductum of the non empty list A, i.e. A without its first element. One cell is accessed, so the computing time is $t = 1$, $c = 0$.
$ADV(A, a, A')$	$a \in \mathcal{O}, A' \in \mathcal{L}, A \in \mathcal{L}_1$. a is the first element of the non empty list A, A' is the reductum of A. One cell is accessed, so $t = 1$, $c = 0$.
$B \leftarrow COMP(a, A)$	$a \in \mathcal{O}, A \in \mathcal{L}, B \in \mathcal{L}_1$. a becomes the first element of the non empty list B, A becomes the reductum of B. One new cell is made available, so $t = 1$, $c = 1$. (Garbage collection is not taken into account.)
$n \leftarrow LENGTH(A)$	$0 \leq n \in \mathcal{A}, A \in \mathcal{L}$. n is the length of A. All cells of the list are traversed, so $t = L(A)$, $c = 0$.

$A \leftarrow LIST(a_1, \ldots, a_n)$	$a_i \in \mathcal{O}$ $(1 \leq i \leq n), n \geq 0, A \in \mathcal{L}$. A is the list of the objects $a_1, \ldots, a_n$. n new cells are made available, so the computing time is $t = n$, $c = n$.
$B \leftarrow INV(A)$	$A, B \in \mathcal{L}$. B is the inverse list of A. The cells of A are used to restructure the list B. The pointer A remains unchanged. All cells are traversed, so $t = L(A)$, $c = 0$.
$B \leftarrow CINV(A)$	$A, B \in \mathcal{L}$. B is the constructive inverse list of A. B is a new list, the cells of A remain unchanged. $L(A)$ new cells are made available, so $t = L(A)$, $c = L(A)$.
$A \leftarrow CONC(A_1, A_2)$	$A, A_1, A_2 \in \mathcal{L}$. A is the concatenation of the lists A_1 and A_2. The cells of A_1 are used to built the list A if A_1 is not empty. The pointers A_1 and A_2 remain unchanged. $L(A_1)$ cells are traversed, so $t = L(A_1)$, $c = 0$.
$A \leftarrow CCONC(A_1, A_2)$	$A, A_1, A_2 \in \mathcal{L}$. A is the constructive concatenation of the lists A_1 and A_2. A is a new list, the cells of A_1 and A_2 are unchanged. $L(A_1)$ new cells are made available, so $t = L(A_1)$, $c = L(A_1)$.
$t \leftarrow EQUAL(A, B)$	$A, B \in \mathcal{O}$, $t \in \{0, 1\}$. If A and B are equal objects, i.e. atoms or lists with the same structure and same atoms, then $t = 1$ otherwise $t = 0$. Maximal the number of cells of the smaller object are traversed, minimal one test is required: $t^+ = \min\{EXTENT(A)+1, EXTENT(B)+1\}$, $t^- = 1$, $c^+ = c^- = 0$.
$n \leftarrow EXTENT(A)$	$A \in \mathcal{O}$, $0 \leq n \in \mathcal{A}$. n is the number of cells of the object A. Overlapping is not counted. For $A \in \mathcal{A}$, $n = 0$. All cells of the object are traversed, so $t = EXTENT(A)$, $c = 0$.
$n \leftarrow ORDER(A)$	$A \in \mathcal{O}$, $0 \leq n \in \mathcal{A}$. n is the maximal nesting level of lists of the object A. For $A \in \mathcal{A}$, $n = 0$. All cells of the object are traversed, so $t = EXTENT(A)$, $c = 0$.

This concludes the summary of list processing functions.

Chapter 6

Basic Arithmetic

The MAS system consists of a kernel with list processing, input / output facilities and an interaction part. The MAS language of the interaction part supports only 'primitive' arithmetic operations i.e. the operators `+`, `-`, `*`, `/`, etc. are only valid on β–integers (atoms). All further arithmetic functions must be imported from Modula–2 libraries.

Available basic arithmetic libraries are:

ALDES/SAC–2 Digit Arithmetic System,

ALDES/SAC–2 Integer Arithmetic System,

ALDES/SAC–2 Rational Number System,

ALDES/SAC–2 Modular Integer Arithmetic System,

ALDES/SAC–2 Integer Factorization System,

ALDES/SAC–2 Set of Integers System,

ALDES/SAC–2 Combinatorical System,

MAS Arbitrary Precision Floating Point System,

MAS Complex Number System,

MAS Quaternion Number System,

MAS Octonion Number System,

MAS Finite Field System,

Which functions are accessible from this libraries can be determined by the MAS `HELP` function. The listings of the definition modules of these libraries is contained in the folder `\HELP` on the MAS distribution disk.

In the following we will discuss the libraries for *arbitrary precision integers*, *arbitrary precision rational numbers* and *arbitrary precision floating point numbers*. Polynomial arithmetic will be discussed later.

For further reading the following documents are recommended:

1. D. E. Knuth: *The Art of Computer Programming*, Vol. II: *Seminumerical Algorithms*, Chap. 4: *Arithmetic*.
   This book contains an in–depth treatment of most basic arithmetic algorithms.

2. G. E. Collins, M. Mignotte, F. Winkler: *Arithmetic in Basic Algebraic Domains.* In B. Buchberger, G. E. Collins, R. Loos: *Computer Algebra*, Computing Supplement 4. This article contains an overview on the computational best algebraic algorithms.

It is further recommended to look into the original ALDES / SAC–2 algorithm documentation or into the translated MAS libraries in Modula–2.

6.1 Integer Arithmetic

The arbitrary precision integers are in the following called **integers**. The small integers are explicitly denoted as β–integers.

We will first discuss the representation of integers by lists. Although it is not required to understand the representation to use the functions on integers, it is required to understand the algorithms. Further the representation has important influence on the computing time, i.e. the complexity of the algorithm.

The base for the number representation of integers is β.

Note: Let $A \in \mathbf{Z}$, $A \geq 0$ then there exist unique numbers n, $a_0, \ldots, a_{n-1} \in \mathbf{Z}$, with $0 \leq a_i < \beta$ $(i = 0, \ldots, n-1)$ and $a_{n-1} \neq 0$ such that:

$$A = \sum_{i=0}^{n-1} a_i \beta^i.$$

For $A < 0$ the same representation holds under the condition $-\beta < a_i \leq 0$ $(i = 0, \ldots, n-1)$ and $a_{n-1} \neq 0$.

Definition: List representation of integers.

$A = 0$ is represented by the atom 0.
$A \neq 0$, $|A| < \beta$, is represented by the atom a_0, and
$A \neq 0$, $|A| \geq \beta$, is represented by the list $(a_0, \ldots, a_{n-1})$.

The 'least significant' β–digit is the first atom in the list representation. The 'most significant' digit appears as last element in the list representation. So e.g. A is even iff a_0 (the first element in the list) is even.

Example:

$\beta = 0 \cdot \beta^0 + 1 \cdot \beta^1$, so $n = 2$ and the representation is $(0, 1)$.
$-\beta = 0 \cdot \beta^0 + (-1) \cdot \beta^1$, the list representation is $(0, -1)$.

6.1.1 Algorithms

The programs of the most important integer algorithms and their complexity are summarized next. Therefore let $\mathcal{A}$ be the set of atoms, $\mathcal{L}$ be the set of lists, $\mathcal{O} = \mathcal{A} \cup \mathcal{L}$ be the set of objects, $\mathcal{I} = \{x \in \mathcal{O} : \ x \text{ represents an element of } \mathbf{Z}\}$ be the set of integers and $\mathcal{L}(\mathcal{I})$ be the set of lists over integers. Further let $L(a)$ denote the number of β–digits of a (if $a \geq \beta$ then this is equal to the list length of the representation of a). We will also write $L(a)$ for $O(L(a))$, i.e. we will not count for constant factors. The computing time

functions t, t^+, t^-, t^* are defined as before in section 5.7. The time analyses are due to G. E. Collins.

$b \leftarrow INEG(a)$ $a, b \in \mathcal{I}$. $b = -a$ is the negative of a. Every β–digit of a must be processed to change the sign, so $t = L(a)$, $c = L(a)$.

$s \leftarrow ISIGN(a)$ $a \in \mathcal{I}$, $s \in \{-1, 0, 1\}$. s is the sign of a. As long as β–digits of a are zero, the list must be processed, so $t^+ = L(a)$, $t^- = 1$, $t^* = 1$, $c = 0$.

$b \leftarrow IABS(a)$ $a, b \in \mathcal{I}$. $b = |a|$ is the absolute value of a. If $\text{sign}(a) = -1$ then a must change the sign, so $t^+ = L(a)$, $t^- = 1$, $c^+ = L(a)$, $c^- = 0$.

$s \leftarrow ICOMP(a, b)$ $a, b \in \mathcal{I}$, $s \in \{-1, 0, 1\}$. $s = \text{sign}(a - b)$ is the sign of $a - b$. At most the smaller number of digits of a and b must be compared, so $t^+ = \min\{L(a), L(b)\}$, $t^- = 1$, $c = 0$.

$c \leftarrow ISUM(a, b)$ $a, b, c \in \mathcal{I}$. $c = a + b$ is the sum of a and b. At least the smaller number of digits of a and b must be added, so $t^- = \min\{L(a), L(b)\}$, $c^- = \min\{L(a), L(b)\}$. If always carrys occur, the maximal computing time is proportional to the greater number of digits of a and b, so $t^+ = \max\{L(a), L(b)\}$, $c^+ = \max\{L(a), L(b)\}$. However carrys occur 'seldom', so $t^* = t^-$, $c^* = c^-$.

$c \leftarrow IDIF(a, b)$ $a, b, c \in \mathcal{I}$. $c = a - b$ is the difference of a and b. b is negated, then the sum is computed, so $t^- = L(b)$, $c^- = L(b)$. The maximal computing time is proportional to the maximal computing time of $ISUM$, so $t^+ = \max\{L(a), L(b)\}$, $c^+ = \max\{L(a), L(b)\}$.

$c \leftarrow IPROD(a, b)$ $a, b, c \in \mathcal{I}$. $c = a \cdot b$ is the product of a and b. The product is computed by the classical method, multiplying each β–digit of a by each β–digit of b, so $t^+ = L(a) \cdot L(b) = L(a)^2$ if $L(a) = L(b)$. $c^+ = L(a) + L(b)$ since the cells of the result are reused and not taken from the available cells during summation.

$c \leftarrow IPRODK(a, b)$ $a, b, c \in \mathcal{I}$. $c = a \cdot b$ is the product of a and b. The product is computed by Karatsubas method, splitting each integer in two halves and then using only 3 multiplications, 4 summations and some shifting to obtain the product. More precisely if $a = a_1\lambda + a_0$, $b = b_1\lambda + b_0$ where λ such that $L(a_1) \approx L(a_0)$, then

$$a \cdot b = a_1b_1\lambda^2 + (a_1b_1 + (a_1 - a_0)(b_0 - b_1) + a_0b_0)\lambda + a_0b_0.$$

If $L(a) = L(b)$ then $t^+ = L(a)^{\log_2(3)} = L(a)^{1.585\ldots}$. However $c^+ = L(a) \cdot L(b)$, since the summation algorithms are used explicitly.
Trade–off: $IPRODK$ is only superior to $IPROD$ if the length of the integers exceed 55 β–digits, i.e. the integers exceed about 500 decimal digits (= 8 lines full of decimal digits on the screen).

$IQR(a, b, q, r)$ $a, b, q, r \in \mathcal{I}$. $b \neq 0$, $q = \lfloor a/b \rfloor$, $r = a - q \cdot b$. The classical division method (dividing the leading β–digits and subtracting b times the trial quotient digit) has been refined to the division of the two

leading digits and subtracting b times the trial quotient digit. The advantage of this refinement is that the trial quotient digit is in nearly all cases the true quotient digit of the quotient, so seldom adjustments of the remainder are necessary. The computing time is proportional to the time for the computation of the product of b and q, so $t^+ = L(b) \cdot L(q)$ and $c^+ = L(b) \cdot L(q)$.

$q \leftarrow IQ(a,b)$ $a, b, q \in \mathcal{I}$. $b \neq 0$, $q = \lfloor a/b \rfloor$. Internally IQR is called, see IQR for the computing time.

$r \leftarrow IREM(a,b)$ $a, b, r \in \mathcal{I}$. $b \neq 0$, $r = a - \lfloor a/b \rfloor \cdot b$. Internally IQR is called, see IQR for the computing time.

$c \leftarrow IEXP(a,n)$ $a, c \in \mathcal{I}$, $0 \leq n \in \mathcal{A}$. $c = a^n$ is the n–th power of a (exponentiation). c is computed by a binary exponentiation method, i.e. c is recursively computed as $(a^{\lfloor n/2 \rfloor})^2 \cdot a^{n - \lfloor n/2 \rfloor 2}$. The computing time is proportional to the time for the computation of the 'biggest' product, so $t^+ = (n \cdot L(a))^2$, $c^+ = 2 \cdot n \cdot L(a)$, 2 because of the classical multiplication algorithm.

$c \leftarrow IGCD(a,b)$ $a, b, c \in \mathcal{I}$. $c = \gcd(a,b)$ is the greatest common divisor of a and b. The classical Euclidean algorithm uses the invariant $\gcd(a,b) = \gcd(b, \mathrm{rem}(a,b))$ for the computation of the gcd until $b = 0$. Assume w.l.o.g. $L(a) \geq L(b)$, then the maximal computing time is $t^+ = L(b) \cdot (L(a) - L(\gcd(a,b)) + 1)$. Thus if $\gcd(a,b) = 1$ and $L(a) = L(b)$ then $t^+ = L(a)^2$ and $c^+ = L(a)^2$. The number of division steps is bounded by $\lceil \log_\phi(\sqrt{5}a) \rceil - 2 \approx 4.8 \log_{10}(a) - 0.32$. The average number of division steps is given by $\frac{12 \ln(2)}{\pi^2} \ln(a) \approx 1.9 \log_{10}(a)$.

$c \leftarrow ILCM(a,b)$ $a, b, c \in \mathcal{I}$. $c = \mathrm{lcm}(a,b)$ is the least common multiple of a and b. c is computed as $\frac{a}{\gcd(a,b)} \cdot b$. So the maximal computing time is $t^+ = 2L(a)^2$.

$a \leftarrow IRAND(n)$ $0 \leq n \in \mathcal{A}$, $a \in \mathcal{I}$. a is a random integer with random sign and $|a| < 2^n$. $t = L(a)$ and $c = L(a)$.

$a \leftarrow IREAD()$ $a \in \mathcal{I}$. An integer a is read from the actual input stream. A conversion from decimal representation to β representation is done, so $t^+ = L(a)^2$ and $c^+ = L(a)^2$.
The syntax for integers is:

```
int = [ "+" | "-" ]
      [ "0" | ... | "9" ] { "0" | ... | "9" }
```

Note: No blanks may appear within an integer !

$IWRITE(a)$ $a \in \mathcal{I}$. The integer a is written to the actual output stream. A conversion from β representation to decimal representation is done, so $t^+ = L(a)^2$ and $c^+ = L(a)^2$. The syntax is the same as for $IREAD$.

$ILWRIT(a)$	$a \in \mathcal{L}(\mathcal{I})$. a is a list of integers. Each element of a is written to the output stream by $IWRITE$. So $t^+ = t_{\text{IWRITE}} \cdot \text{length}(a)$ and $c^+ = c_{\text{IWRITE}} \cdot \text{length}(a)$.
$l \leftarrow IFACT(a)$	$a \in \mathcal{I}, l \in \mathcal{L}(\mathcal{I})$. l is the list of the integer prime factors of a, multiple factors occur multiple times in l. (e.g. $IFACT(8) = (2,2,2)$.) For large $L(a)$ the computing time has exponential growth: $t^+ = c^{L(a)}$.
$l \leftarrow SMPRM()$	$l \in \mathcal{L}(\mathcal{A})$. l is the list of the prime numbers between 2 and 1000.
$l \leftarrow DPGEN(m,k)$	$l \in \mathcal{L}(\mathcal{A})$, $m, k \in \mathcal{A}$, m and k positive, $k \leq 1000$, $m + 2 \cdot k < \beta$. l is the list of all prime numbers p, such that $m \leq p < m + 2 \cdot k$.

This concludes the summary of integer arithmetic functions.

Examples:

```
a:=LIST(0,7,3).     --> (0 7 3)
IWRITE(a).          --> 864691132213231616
ISIGN(a).           --> 1
INEG(a).            --> (0 -7 -3)
ISUM(a,a).          --> (0 14 6)

b:=IEXP(2,29).      --> (0 1)
IREM(a,b).          --> 0
IQ(a,b).            --> (7 3)
IPROD(a,b).         --> (0 0 7 3)
ICOMP(a,b).         --> 1

IFACT(a).           --> (2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
                         2 2 2 2 2 2 2 2 2 2 2 23 70026641)
```

In the example an integer a is set to the list $(0,7,3)$. That means a represents the integer $0 \cdot \beta^0 + 7 \cdot \beta^1 + 3 \cdot \beta^2$. The decimal representation of a is generated in the second line. Since $\text{sign}(a) = \text{sign}(7) = 1$ we have $a > 0$. $-a$ has the representation $(0,-7,-3)$. $a + a$ has the representation $(0,14,6)$.

Next b is set to $\beta = 2^{29}$, represented by the list $(0,1)$. The remainder by division by b is zero, a rem $b = 0$, (= the first element of the list representation of a). The quotient a/b is the rest list of the representation of a i.e. $(7,3)$. $a \cdot b$ shifts a one β–digit to the left $(0,0,7,3)$. Since $7 > 1$ and the list representation of a is longer than that of b we have $\text{sign}(a - b) = 1$, i.e. $a > b$.

Finally the prime factors of a are computed, since a is divisible by β we must have 29 occurrences of 2 in the prime factors. And $7 + 3\beta$ seems to have the prime factors 23 and 70026641.

For illustration we list the algorithms `ICOMP` and `IPROD` in Modula–2 in MAS. The function of the algorithms should be clear from the step comments and the integer representation discussed before.

```
PROCEDURE ICOMP(A,B: LIST): LIST;
```

```
(*Integer comparison.  A and B are integers.  s=SIGN(A-B).*)
VAR   AL, AP, BL, BP, DL, SL, UL, VL: LIST;
BEGIN
(*1*) (*A and B single-precision.*)
      IF (A < BETA) AND (B < BETA) THEN SL:=MASSIGN(A-B);
         RETURN(SL); END;
(*2*) (*A single-precision.*)
      IF A < BETA THEN SL:=-ISIGNF(B); RETURN(SL); END;
(*3*) (*B single-precision.*)
      IF B < BETA THEN SL:=ISIGNF(A); RETURN(SL); END;
(*4*) (*compare corresponding digits.*) SL:=0; AP:=A; BP:=B;
      REPEAT ADV(AP,AL,AP); ADV(BP,BL,BP); UL:=MASSIGN(AL);
             VL:=MASSIGN(BL);
             IF UL*VL = -1 THEN SL:=UL; RETURN(SL); END;
             DL:=AL-BL;
             IF DL <> 0 THEN SL:=MASSIGN(DL); END;
             UNTIL (AP = SIL) OR (BP = SIL);
(*5*) (*same length*)
      IF (AP = SIL) AND (BP = SIL) THEN RETURN(SL); END;
(*6*) (*use sign of longer input.*)
      IF AP = SIL THEN SL:=-ISIGNF(BP); ELSE
         SL:=ISIGNF(AP); END;
      RETURN(SL);
(*9*) END ICOMP;
```

BETA denotes β, MASSIGN denotes the sign function for β–integers. ISIGNF denotes the sign function for integers in the libraries (for compatibility with the ALDES / SAC–2 libraries). SIL denotes the empty list.

```
PROCEDURE IPROD(A,B: LIST): LIST;
(*Integer product.  A and B are integers.  C=A*B.*)
VAR   AL, AP, APP, BL, BP, C, C2, CL, CLP, CP, CPP, EL, FL,
      I, ML, NL, TL: LIST;
BEGIN
(*1*) (*A or B zero.*)
      IF (A = 0) OR (B = 0) THEN C:=0; RETURN(C); END;
(*2*) (*A and B single-precision.*)
      IF (A < BETA) AND (B < BETA) THEN DPR(A,B,CLP,CL);
         IF CLP = 0 THEN C:=CL; ELSE C:=LIST2(CL,CLP); END;
         RETURN(C); END;
(*3*) (*A or B single-precision.*)
      IF A < BETA THEN C:=IDPR(B,A); RETURN(C); END;
      IF B < BETA THEN C:=IDPR(A,B); RETURN(C); END;
(*4*) (*interchange if B is longer.*) ML:=LENGTH(A); NL:=LENGTH(B);
      IF ML >= NL THEN AP:=A; BP:=B; ELSE AP:=B; BP:=A; END;
(*5*) (*set product to zero.*) C2:=LIST2(0,0); C:=C2;
      FOR I:=1 TO ML+NL-2 DO C:=COMP(0,C); END;
(*6*) (*multiply digits and add products.*) CP:=C;
      REPEAT APP:=AP; ADV(BP,BL,BP);
             IF BL <> 0 THEN CPP:=CP; CLP:=0;
                REPEAT ADV(APP,AL,APP); DPR(AL,BL,EL,FL);
                       CL:=FIRST(CPP); CL:=CL+FL; CL:=CL+CLP;
                       CLP:=CL DIV BETA; TL:=CLP*BETA; CL:=CL-TL;
                       SFIRST(CPP,CL); CLP:=EL+CLP; CPP:=RED(CPP);
                       UNTIL APP = SIL;
                SFIRST(CPP,CLP); END;
             CP:=RED(CP);
             UNTIL BP = SIL;
(*7*) (*leading digit zero*)
      IF CLP = 0 THEN SRED(C2,SIL); END;
```

```
     RETURN(C);
(*9*) END IPROD;
```

DPR denotes the β–digit product, IDPR denotes the integer product with a β–digit, SFIRST (Set FIRST) and SRED (Set RED) are list modifying functions not available for interactive use in MAS.

6.1.2 Exercises

1. Compute the gcd of 156562431911123 and 442677773754356 in three ways:
   (a) Use the the built–in algorithm for the gcd.
   (b) Write an recursive algorithm in MAS and use it for the computation.
   (c) Write an iterative algorithm in MAS and use it for the computation.
   (d) $*$ Write an iterative algorithm for the extended gcd.

2. Write algorithms to compute the determinant of a square matrix over the integers. Assume that the matrices are represented as lists of lists of integers. Use the Laplace method for the expansion of the determinant. Therefore write the following sub–algorithms:
   (a) Deletion of an element of a vector (represented as list).
   (b) Deletion of a column of a matrix (represented as list of vectors).
   (c) Determinant expansion using Laplace's method.
   (d) An algorithm to generate matrices with respect to a function of the matrix entries.
   (e) $*$ An algorithm with Gauss's method for the computation of the determinant.
   (f) $*$ Write the determinant algorithm for other coefficient domains, e.g. rational numbers.

The solutions to the exercises are discussed in the sequel.

Exercise 1. The built–in gcd algorithm has the name IGCD.

For the MAS algorithms we use the euclidean method:

$$\gcd(a,b) = \begin{cases} a & \text{if } \ b = 0, \\ \gcd(b, \mathrm{rem}(a,b)) & \text{else.} \end{cases}$$

With this definition the recursive gcd algorithm can be formulated as follows:

```
PROCEDURE ggt(a,b);
IF b = 0 THEN RETURN(a)
         ELSE RETURN(ggt(b,IREM(a,b)))
   END ggt.
```

The integer remainder is called IREM.

The iterative gcd algorithm can be formulated using the WHILE–statement and an additional variable d:

```
PROCEDURE GGT(a,b);
VAR   d: ANY;
BEGIN
WHILE b # 0 DO d:=b;
      b:=IREM(a,b); a:=d END;
RETURN(a) END GGT.
```

The numbers can be converted to internal representation by `IREAD`. The sample output with computing times follows:

```
a:=IREAD(). 156562431911123
b:=IREAD(). 442677773754356

c:=IGCD(a,b).
{0 sec} ANS: 7

c:=ggt(a,b).
{4 sec} ANS: 7

c:=GGT(a,b).
{2 sec} ANS: 7
```

The gcd is always 7. The built–in algorithm is the fastest: it needs unmeasurable many seconds. The recursive algorithm is the slowest: it needs about 4 seconds and the iterative algorithm need 2 seconds.

Exercise 2. Let $A = (a_{ij})$ be a $n \times n$ matrix over a commutative ring without zero divisors. Then the determinant has the following expansion:

$$\det(A) = \sum_{i=1}^{n} (-1)^{i+j} \cdot a_{ij} \cdot \det(A_{ij}) \quad \text{for some} \quad j, (1 \leq j \leq n).$$

Where A_{ij} is the matrix A without the i–th row and the j–th column.

The computing time can be estimated by the following considerations: The determinant expansion produces $n!$ summands of products of n factors. Assume $L(A) = \max\{L(a_{ij}), 1 \leq i, j \leq n\}$. So $t^+ = n! \cdot L(A)^n$ and $t^- = L(A)^n$, e.g. if the matrix is in triangular form.

In contrast it needs about n^3 field operations and n^5 ring operations to transform a matrix to upper triangular from by the Gauss respective the Bareiss algorithm. I.e. the Laplace expansion is very slow.

The determinant expansion algorithm can be formulated as follows

```
PROCEDURE det(M);
(*Determinant. M is a matrix (i.e a list
of row lists). The determinant of M is computed. *)
VAR   i, d, dp, s, N, MP, V, VP, v: LIST;
BEGIN
(*1*) d:=0;
      IF M = NIL THEN RETURN(d) END;
(*2*) ADV(M,V,MP);
```

```
      IF MP = NIL THEN RETURN(FIRST(V)) END;
(*3*) s:=1; i:=0;
      WHILE V # NIL DO ADV(V,v,V); i:=i+1;
            IF v # 0 THEN
               N:=delcolumn(MP,i); dp:=det(N);
               dp:=IPROD(v,dp);
               IF s < 0 THEN dp:=INEG(dp) END;
               d:=ISUM(d,dp);
               END;
            s:=-s; END;
(*4*) RETURN(d);
(*9*) END det.
```

Step 2 handles the recursion base of a 1×1 matrix. In step 3 the determinant is expanded w.r.t. the first row (that is $j = 1$ in our formula). The test if some $a_{ij} = 0$ (denoted by `v`) is important, since in this case the expansion of the sub–matrix is useless and much computing time can be saved. Since the first row of the matrix has been removed in step 2, it is only necessary to remove the i–th column of the matrix. The matrices A_{i1} are denoted by `N`. The factor $(-1)^{i+1}$ is handled by a sign flag denoted by `s`.

The vector element deletion algorithm is

```
PROCEDURE delelem(V,i);
(*Delete element in list. V is a list. The i-th
element of V is deleted. 0 <= i <= length(V). *)
VAR   U, VP, v, j: ANY;
BEGIN
(*1*) IF i <= 0  THEN RETURN(V) END;
      IF V = NIL THEN RETURN(V) END;
(*2*) VP:=V; j:=0; U:=NIL;
      REPEAT j:=j+1;
             IF VP = NIL THEN RETURN(V) END;
             ADV(VP,v,VP); U:=COMP(v,U);
             UNTIL j = i;
(*3*) U:=RED(U); U:=INV(U); U:=CONC(U,VP);
      RETURN(U);
(*9*) END delelem.
```

Step 1 does some precondition checks. In step 2 the beginning of the list is copied to avoid the modification of existing data. In step 3 the (new) beginning is reversed and concatenated with the rest of the vector.

The column delete algorithm is straight forward: for each row the previous algorithm is called.

```
PROCEDURE delcolumn(M,i);
(*Delete column in matrix. M is a matrix (i.e a list
of row lists). In each row the i-th element is
deleted. *)
VAR   N, MP, V: ANY;
BEGIN
```

```
(*1*) IF i <= 0  THEN RETURN(M) END;
      IF M = NIL THEN RETURN(M) END;
(*2*) MP:=M; N:=NIL;
      WHILE MP # NIL DO ADV(MP,V,MP);
            V:=delelem(V,i); N:=COMP(V,N);
            END;
(*3*) N:=INV(N); RETURN(N);
(*9*) END delcolumn.
```

The sample matrices are:

1. the unit matrix $E = (e_{ij})$ with: $e_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{else,} \end{cases}$
2. an upper triangular matrix $U = (u_{ij})$ with: $u_{ij} = \begin{cases} j & \text{if } i \leq j, \\ 0 & \text{else,} \end{cases}$
3. a lower triangular matrix $L = (l_{ij})$ with: $l_{ij} = \begin{cases} j & \text{if } i \geq j, \\ 0 & \text{else.} \end{cases}$

The generating functions for the matrices are as follows:

```
PROCEDURE eh(i,j);
IF i = j THEN RETURN(1) ELSE RETURN(0) END eh.

PROCEDURE ut(i,j);
IF i <= j THEN RETURN(j) ELSE RETURN(0) END ut.

PROCEDURE lt(i,j);
IF i >= j THEN RETURN(j) ELSE RETURN(0) END lt.
```

`eh` means 'Einheitsmatrix' = unit matrix, `ut` means upper triangular matrix and `lt` means lower triangular matrix.

The matrix generation procedure takes the generating function `f` and the dimension `k` as input and delivers the appropriate matrix.

```
PROCEDURE mat(f,k);
VAR   i, j, V, M: LIST;
BEGIN
(*1*) M:=NIL;
      IF k <= 0 THEN RETURN(M) END;
(*2*) i:=0;
      WHILE i < k DO i:=i+1; j:=0; V:=NIL;
            WHILE j < k DO j:=j+1;
                  V:=COMP(f(i,j),V) END;
            V:=INV(V); M:=COMP(V,M);
            END;
(*4*) M:=INV(M); RETURN(M);
(*9*) END mat.
```

A sample output follows:

```
A:=mat(ut,5).
{4 sec} ANS: ((1 2 3 4 5) (0 2 3 4 5) (0 0 3 4 5)
              (0 0 0 4 5) (0 0 0 0 5))

d:=det(A). {246 sec} ANS: 120

A:=mat(lt,10).
{14 sec} ANS: ((1 0 0 0 0 0 0 0 0 0) (1 2 0 0 0 0
0 0 0 0) (1 2 3 0 0 0 0 0 0 0) (1 2 3 4 0 0 0 0 0 0)
(1 2 3 4 5 0 0 0 0 0) (1 2 3 4 5 6 0 0 0 0) (1 2 3
4 5 6 7 0 0 0) (1 2 3 4 5 6 7 8 0 0) (1 2 3 4 5 6 7
8 9 0) (1 2 3 4 5 6 7 8 9 10))

d:=det(A). {64 sec} ANS: 3628800
```

In the first example a 5×5 upper triangular matrix is generated. The determinant is 120 and the computation takes 246 seconds.

In the second example a 10×10 lower triangular matrix is generated. The determinant is 3628800 and the computing time is 64 seconds.

The second computation is much faster, since the determinant is expanded with respect to the first row. And the lower triangular matrix has most times zero in these rows.

6.2 Rational Number Arithmetic

The elements of $\mathbf{Q}$, fractions of integers are in the following called **rational numbers**. We will first discuss the representation of rational numbers by lists.

Note: Let $a \in \mathbf{Q}$, $a = \frac{p}{q}$, $p, q \in \mathbf{Z}$, then there exist integers $p', q' \in \mathbf{Z}$, with $q' > 0$ and $\gcd(p', q') = 1$ such that:

$$\frac{p}{q} = \frac{p'}{q'}.$$

p and p' are called denominator and q and q' are called numerator of the rational number a.

Definition: List representation of rational numbers.

$0 \in \mathbf{Q}$ is represented by the atom 0.
$\frac{p}{q} \in \mathbf{Q}$, is represented by the list (p', q'), where p', q' are represented as integers.
p', q' defined as before.

Example:

$\frac{7}{3}$ is represented as $(7, 3)$.
$-\frac{1}{2}$ is represented as $(-1, 2)$.
$\frac{0}{3}$ is represented as 0.

The representation of $\frac{\beta}{3}$ is $((0,1),3)$.
$-\frac{\beta}{3}$ is represented as $((0,-1),3)$.
$\frac{4}{8}$ is represented as $(1,2)$.

6.2.1 Algorithms

The programs of the most important rational number algorithms and their complexity are summarized in the following.

Let $\mathcal{A}$ be the set of atoms, $\mathcal{L}$ be the set of lists, $\mathcal{O} = \mathcal{A} \cup \mathcal{L}$ be the set of objects, $\mathcal{I} = \{x \in \mathcal{O} : \ x \text{ represents an element of } \mathbf{Z}\}$, be the set of integers and $\mathcal{R} = \{x \in \mathcal{O} : \ x$ represents an element of $\mathbf{Q}\}$ be the set of rational numbers. Further let $L(a)$ denote the maximum of the length of the numerator and denominator of a ($L(a) = \max\{L(p'), L(q')\}$). We will also write $L(a)$ for $O(L(a))$, i.e. we will not count for constant factors. The computing time functions t, t^+, t^-, t^* are defined as before in section 5.7. The methods for efficient product and sum algorithms are due to P. Henrici. The time analyses are due to G. E. Collins.

$b \leftarrow RNINT(a)$ — $a \in \mathcal{I}$, $b \in \mathcal{R}$. $b = \frac{a}{1}$ is the embedding of the integer a into the rational numbers. Since the list $(a,1)$ is built $t^+ = 2$, $c^+ = 2$, $t^- = 1$, $c^- = 0$.

$c \leftarrow RNRED(a,b)$ — $a, b \in \mathcal{I}$, $c \in \mathcal{R}$. $c = \frac{a}{b}$ is the construction of a rational number from an integer numerator and a denominator. a and b are reduced to lowest terms. The list (a', b') is built with $\frac{a'}{b'} = \frac{a}{b}$, $b' > 0$ and $\gcd(a', b') = 1$. $t^+ = t^+_{\text{IGCD}} = L(c)^2$, $c^+ = L(c)^2$, $t^- = 1$, $c^- = 0$.

$b \leftarrow RNDEN(a)$ — $a \in \mathcal{R}$, $b \in \mathcal{I}$. $a = \frac{n}{b}$, b is the denominator of a. $t = 1$, $c = 0$.

$b \leftarrow RNNUM(a)$ — $a \in \mathcal{R}$, $b \in \mathcal{I}$. $a = \frac{b}{d}$, b is the numerator of a. $t = 1$, $c = 0$.

$b \leftarrow RNNEG(a)$ — $a, b \in \mathcal{R}$. $b = -a$ is the negative of a. The sign of the denominator must be changed, so $t^+ = t^+_{\text{INEG}} = L(a)$, $c^+ = L(a)$.

$s \leftarrow RNSIGN(a)$ — $a \in \mathcal{R}$, $s \in \{-1, 0, 1\}$. s is the sign of a. The sign of the denominator must be determined, so $t^+ = t^+_{\text{ISIGN}} = L(a)$, $c = 0$.

$b \leftarrow RNABS(a)$ — $a, b \in \mathcal{R}$. $b = |a|$ is the absolute value of a. If $\text{sign}(a) = -1$ then a must change the sign, so $t^+ = L(a)$, $t^- = 1$, $c^+ = L(a)$, $c^- = 0$.

$s \leftarrow RNCOMP(a,b)$ — $a, b \in \mathcal{R}$, $s \in \{-1, 0, 1\}$. $s = \text{sign}(a - b)$ is the sign of $a - b$. Let $a = \frac{a_1}{a_2}$, $b = \frac{b_1}{b_2}$, then $a < b \iff a_1 b_2 < b_1 a_2$. Therefore possibly integer products must be formed and the computing time is proportional to integer product and comparison. So if $L(a) = L(b)$ we have $t^+ = t^+_{\text{IPROD}} + t^+_{\text{ICOMP}} = L(a)^2$, $t^- = 2t^-_{\text{ISIGN}} = 2$, $c^+ = 2L(a)$, $c^- = 0$.

$c \leftarrow RNPROD(a,b)$ $a, b, c \in \mathcal{R}$. $c = a \cdot b$ is the product of a and b. Let $a = \frac{a_1}{a_2}$, $b = \frac{b_1}{b_2}$. The product, defined as

$$c = \frac{c_1}{c_2} = \frac{a_1}{a_2} \cdot \frac{b_1}{b_2} = \frac{a_1 \cdot b_1}{a_2 \cdot b_2},$$

is computed in a way that exploits the precondition that the denominator and numerator have gcd = 1. Therefore let $d_1 = \gcd(a_1, b_2)$, $d_2 = \gcd(a_2, b_1)$ and let $a_1 = d_1 a_1'$, $b_2 = d_1 b_2'$, $b_1 = d_2 b_1'$ and $a_2 = d_2 a_2'$. So $c = \frac{a_1' \cdot b_1'}{a_2' \cdot b_2'}$, and we obtain the postcondition $\gcd(a_1' b_1', a_2' b_2') = 1$ without actually computing that gcd. If $L(a) = L(b)$, the computing time is therefore only $t^+ = L(a)^2$ instead of $(2L(a))^2$ for the classical method. $c^+ = L(a)^2$, $t^- = 2$ and $c^- = 2$.

$b \leftarrow RNEXP(a,n)$ $a, b \in \mathcal{R}$, $n \in \mathbf{N}$. $c = a^n$ is the n–th power of a (exponentiation). Like the integer exponentiation c is computed by a binary exponentiation method. The computing time is proportional to the time for the computation of the 'biggest' product, so $t^+ = (n \cdot L(a))^2$, $c^+ = 2 \cdot n \cdot L(a)$. $t^- = 2$ and $c^- = 2$.

$b \leftarrow RNINV(a)$ $a, b \in \mathcal{R}$. $b = 1/a$ is the inverse of a. If $a = \frac{a_1}{a_2}$ then $1/a = \frac{a_2}{a_1}$. Only if the denominator has negative sign, the signs of the denominator and numerator must be changed. $t^+ = 2t^+_{\mathrm{INEG}} = 2L(a)$, $c^+ = 2c^+_{\mathrm{INEG}} = 2L(a)$, $t^- = 2$ and $c^- = 2$.

$c \leftarrow RNQ(a,b)$ $a, b, c \in \mathcal{R}$. $c = a/b$ is the quotient of a and b. The inverse of b is determined and the product of a and $1/b$ is computed. So the computing time is $t^+ = t^+_{\mathrm{RNINV}} + t^+_{\mathrm{RNPROD}} = L(a)^2$. $c^+ = L(a)^2$, $t^- = 4$ and $c^- = 4$.

$c \leftarrow RNSUM(a,b)$ $a, b, c \in \mathcal{R}$. $c = a + b$ is the sum of a and b. Let $a = \frac{a_1}{a_2}$, $b = \frac{b_1}{b_2}$. The sum is defined as

$$c = \frac{c_1}{c_2} = \frac{a_1}{a_2} + \frac{b_1}{b_2} = \frac{a_1 \cdot b_2 + a_2 \cdot b_1}{a_2 \cdot b_2}.$$

We will also exploit the preconditions that the denominator and numerator have gcd = 1 to minimize the computing time. Therefore let $d = \gcd(a_2, b_2)$ and let $a_2 = d a_2'$, $b_2 = d b_2'$. Further let $t = a_1 b_2 + a_2 b_1$, $t' = a_1 b_2' + a_2' b_1$ and $e = \gcd(t', d)$. Now observe that

$$\gcd(t, a_2 b_2) = d \cdot \gcd(t', a_2 b_2') = d \cdot \gcd(t', d) = d \cdot e.$$

Since d divides $a_1 b_2 + a_2 b_1$ and $a_2 b_2$ the first equation holds. And since both a_2' and b_2' have gcd = 1 with $a_1 b_2' + a_2' b_1$, the second equation holds. Let $t' = t'' e$ and $a_2 = a_2'' e$ then the sum is $c = \frac{t''}{a_2'' \cdot b_2'}$. The postcondition $\gcd(t'', a_2'' b_2') = 1$ holds by construction of t'' and a_2''. If $L(a) = L(b)$ the computing time is $t^+ = t^+_{\mathrm{IGCD}_d} + 2t^+_{\mathrm{IPROD}_{t'}} + t^+_{\mathrm{ISUM}_{t'}} + t^+_{\mathrm{IGCD}_e}$. $t^+ = L(a)(L(a) -$

$L(d)+1)+2L(a)(L(a)-L(d)+1)+2L(a)+2L(a)(L(d)-L(e)+1)$. For $L(d)=1$ we obtain $t^+ = L(a)^2+2L(a)^2+4L(a) = 3L(a)^2$ and for $L(d)=L(a)$, $t^+ = L(a)+4L(a)+2L(a)^2 = 3L(a)^2$. For the classical method we get $t^+ = 3L(a)^2+2L(a)+(2L(a))^2 = 7L(a)^2$. Further $c^+ = L(a)^2$, t^- = const and $c^- = 2$.

$c \leftarrow RNDIF(a,b)$ $a,b,c \in \mathcal{R}$. $c = a-b$ is the difference of a and b. b is negated, then the sum is computed, so $t^- = L(b)$, $c^- = L(b)$. The maximal computing time is proportional to the maximal computing time of $RNSUM$, so $t^+ = \max\{L(a),L(b)\}$, $c^+ = \max\{L(a),L(b)\}$.

$a \leftarrow RNREAD()$ $a \in \mathcal{R}$. A rational number a is read from the actual input stream. A conversion from decimal representation to β representation is done and the numerator and denominator are reduced to lowest terms, so $t^+ = L(a)^2$ and $c^+ = L(a)^2$.
The syntax accepted by $RNREAD$ for rational numbers is:

```
rat = int [ "/" int ]
```

$a \leftarrow RNDRD()$ $a \in \mathcal{R}$. Same as $RNREAD$ except for a different syntax. The syntax accepted by $RNDRD$ for rational numbers is:

```
rat = int [ "/" int
          | "." unsigned-int [ "E" beta-int ] ]
```

Note: No blanks are allowed before `"E"`.

$RNWRIT(a)$ $a \in \mathcal{R}$. The rational number a is written to the actual output stream. A conversion from β representation to decimal representation is done, so $t^+ = L(a)^2$ and $c^+ = L(a)^2$. The syntax is the same as for $RNREAD$.

$RNDWR(a,s)$ $a \in \mathcal{R}$, $s \in \mathcal{A}$. The rational number a is written to the actual output stream. With s digits following the decimal point. The syntax is:

```
rat = int "." unsigned-int [ "+" | "-" ]
```

A trailing `"+"` (or `"-"`) indicates whether the rational number is greater (or smaller) than the written decimal approximation.

$a \leftarrow RNRAND(n)$ $0 \leq n \in \mathcal{A}$, $a \in \mathcal{R}$. $a = \frac{a_1}{|a_2|+1}$ reduced to lowest terms is a random rational number with $a_{1,2} = IRAND(n)$. $t^+ = L(a)^2$ and $c^+ = L(a)^2$.

This concludes the summary of rational number arithmetic functions.
Examples:

```
a:=RNRED(6,-8).       --> (-3,4)

a:=RNREAD(). 3/4      --> (3,4)
```

```
a:=RNREAD(). 3/-4     --> (-3,4)
a:=RNDRD(). 3.4       --> (17,5)
a:=RNDRD(). 3.14E7    --> (31400000,1)

RNWRIT(a).            --> -3/4
RNDWR(a,10).          --> -0.7500000000
RNDWR(a,0).           --> -1.-
RNDWR(a,1).           --> -0.7+
```

The first statement shows the normalization of a rational number $\frac{6}{-8} = \frac{-3}{4}$ with $\gcd(3,4) = 1$ and $4 > 0$.

In the next two statements RNREAD is used to read two rational numbers. Observe that 3/-4 has negative numerator which is normalized during input. RNDRD is able to read the decimal fraction 3.4 which is $3 + \frac{4}{10} = 3 + \frac{2}{5} = \frac{15+2}{5} = \frac{17}{5}$ as shown. The number 3.14E7 is interpreted as $(3 + \frac{14}{100}) \cdot 10^7$, which gives the rational number $\frac{31400000}{1}$.

The output routine RNWRIT needs no further explanation. RNDWR writes a rational number as decimal fraction with specified number of digits after the decimal point. $\frac{-3}{4}$ with 10 digits after the decimal point is exactly -0.7500000000. With 0 digits after the decimal point $-1.-$ is printed, the trailing $-$ indicates that $-1. < -\frac{3}{4}$. With 1 digit after the decimal point $-0.7+$ is printed which indicates that $-0.7 > -\frac{3}{4}$.

For illustration we list the algorithms RNSUM and RNPROD in Modula–2 in MAS. The function of the algorithms should be clear from the step comments and the integer representation discussed before.

```
PROCEDURE RNPROD(R,S: LIST): LIST;
(*Rational number product.  R and S are rational numbers.  T=R*S.*)
VAR   D1, D2, R1, R2, RB1, RB2, S1, S2, SB1, SB2, T, T1, T2: LIST;
BEGIN
(*1*) (*r=0 or s=0.*)
      IF (R = 0) OR (S = 0) THEN T:=0; RETURN(T); END;
(*2*) (*obtain numerators and denominators.*) FIRST2(R,R1,R2);
      FIRST2(S,S1,S2);
(*3*) (*r and s integers.*)
      IF (R2 = 1) AND (S2 = 1) THEN T1:=IPROD(R1,S1);
         T:=LIST2(T1,1); RETURN(T); END;
(*4*) (*r or s an integer.*)
      IF R2 = 1 THEN IGCDCF(R1,S2,D1,RB1,SB2); T1:=IPROD(RB1,S1);
         T:=LIST2(T1,SB2); RETURN(T); END;
      IF S2 = 1 THEN IGCDCF(S1,R2,D2,SB1,RB2); T1:=IPROD(SB1,R1);
         T:=LIST2(T1,RB2); RETURN(T); END;
(*5*) (*general case.*) IGCDCF(R1,S2,D1,RB1,SB2);
      IGCDCF(S1,R2,D2,SB1,RB2); T1:=IPROD(RB1,SB1); T2:=IPROD(RB2,SB2);
      T:=LIST2(T1,T2); RETURN(T);
(*8*) END RNPROD;
```

FIRST2 accesses the first two elements of a list. LIST2 constructs a list of two elements. IGCDCF computes the gcd (4–th (VAR) parameter) together with the cofactors (5-th and 6-th parameter). IPROD is the product of two integers. The correctness of step 5 is discussed in the summary.

```
PROCEDURE RNSUM(R,S: LIST): LIST;
(*Rational number sum.  R and S are rational numbers.  T=R+S.*)
VAR   D, E, R1, R2, RB2, S1, S2, SB2, T, T1, T2: LIST;
```

```
BEGIN
(*1*) (*r=0 or s=0.*)
      IF R = 0 THEN T:=S; RETURN(T); END;
      IF S = 0 THEN T:=R; RETURN(T); END;
(*2*) (*obtain numerators and denominators.*) FIRST2(R,R1,R2);
      FIRST2(S,S1,S2);
(*3*) (*r and s integers.*)
      IF (R2 = 1) AND (S2 = 1) THEN T1:=ISUM(R1,S1);
         IF T1 = 0 THEN T:=0; ELSE T:=LIST2(T1,1); END;
         RETURN(T); END;
(*4*) (*r or s an integer.*)
      IF R2 = 1 THEN T1:=IPROD(R1,S2); T1:=ISUM(T1,S1);
         T:=LIST2(T1,S2); RETURN(T); END;
      IF S2 = 1 THEN T1:=IPROD(R2,S1); T1:=ISUM(T1,R1);
         T:=LIST2(T1,R2); RETURN(T); END;
(*5*) (*general case.*) IGCDCF(R2,S2,D,RB2,SB2);
      T1:=ISUM(IPROD(R1,SB2),IPROD(RB2,S1));
      IF T1 = 0 THEN T:=0; RETURN(T); END;
      IF D <> 1 THEN E:=IGCD(T1,D);
         IF E <> 1 THEN T1:=IQ(T1,E); R2:=IQ(R2,E); END;
         END;
      T2:=IPROD(R2,SB2); T:=LIST2(T1,T2); RETURN(T);
(*8*) END RNSUM;
```

`ISUM` denotes the sum of two integers. `IGCD` denotes the integer gcd. `IQ` is the integer quotient. The other algorithms are mentioned in the description of `RNPROD`. The correctness of step 5 is discussed in the summary.

6.2.2 Exercises

1. Write an algorithm to compute the exponential series exp over the rational numbers up to a desired precision $\varepsilon > 0$. With this algorithm compute $e = \exp(1)$ up to 50 decimal digits.

The solution to the exercise is discussed in the sequel.

The exponential series is defined for complex x by

$$\exp(x) = \sum_{n=1}^{\infty} \frac{x^n}{n!}.$$

The exponential series can be approximated by $\sum_{n=1}^{N} \frac{x^n}{n!} + r_{N+1}(x)$. Where for the rest we have

$$r_{N+1}(x) < 2\frac{|x|^{N+1}}{(N+1)!}.$$

Let $\varepsilon > 0$ be fixed. We want to approximate $\exp(x)$ such that $r_{N+1}(x) < \varepsilon$, therefore the summand must satisfy $\frac{|x|^{N+1}}{(N+1)!} \leq |\frac{x^{N+1}}{(N+1)!}| < \frac{\varepsilon}{2}$.

With this information the algorithm can be formulated as follows:

```
PROCEDURE Exp(x,eps);
(*Exponential function. eps is the desired precision. *)
VAR   s, xp, ep, i, y: ANY;
```

```
BEGIN
(*1*) y:=RNINT(1); s:=RNINT(1); i:=0;
      ep:=RNPROD(eps,RNRED(1,2));
(*2*) REPEAT i:=i+1; xp:=RNRED(1,i);
             y:=RNPROD(y,x); y:=RNPROD(y,xp);
             s:=RNSUM(s,y)
             UNTIL RNCOMP(RNABS(y),ep) <= 0;
      RETURN(s)
(*9*) END Exp.
```

The complexity can be estimated as follows: the most expensive operations in the REPEAT–loop are `RNPROD(y,x)`, `RNSUM(s,y)` and `RNCOMP( . ,ep)`, which have computing times proportional to $L(.)^2$. For the 'biggest' product $x^{n-1} \cdot x$ we have $t^+_{\mathrm{RNPROD}} = (nL(x))L(x)$. For the 'biggest' sum $(\sum_{i=1}^{n-1} \frac{x^i}{i!}) + \frac{x^n}{n!}$ we have $t^+_{\mathrm{RNSUM}} = (nL(x))^2$. The comparison needs also $t^+_{\mathrm{RNCOMP}} = (nL(x))^2$. By this the computing time for the exponential series is $t^+_{\mathrm{Exp}} = (nL(x))^2$.

As an example we are going to compute $e = \mathrm{Exp}(1)$ with precision $\varepsilon = \frac{1}{10^{50}}$. The sample output follows:

```
dig:=50.   Eps:=RNRED(1,IEXP(10,dig)).

one:=RNINT(1).

e:=Exp(one,Eps).
{20 sec} ANS: ((119769761 450433631 444044040 360650700
               406458113 17125896) (0338539520 159342123
               356614372 176392732 6300265))

BEGIN CLOUT("AbsErr = "); RNDWR(Eps,dig); BLINES(0);
      CLOUT("Result = "); RNDWR(e,dig); BLINES(0);
      CLOUT("Result = "); RNWRIT(e); BLINES(0) END.

AbsErr = 0.00000000000000000000000000000000
           00000000000000001

Result = 2.71828182845904523536028747135266 2
           49775724709369996-

Result = 763840519752288597376564549384146 88
         9815927382313633/281001223550575979
         708628521248902313987276800000000
```

The computation needs 20 seconds on an Atari ST. Then the absolute error, e as decimal fraction and e as rational number are printed. Observe that $L(e) = 6$ by counting the β–integers from the internal representation of e.

6.3 Arbitrary Precision Floating Point Arithmetic

The arbitrary precision floating point numbers are called **floating point numbers** for short. We will first discuss the representation of floating point numbers by lists.

Note: Let $a \in \mathbf{Q}$, then there exists an integer $e \in \mathbf{Z}$ and a rational number $m \in \mathbf{Q}$ with $\frac{1}{2} \leq |m| < 1$ such that

$$a = m \cdot 2^e.$$

In this case $e = \lfloor \log_2(|a|) \rfloor + 1$.

The binary precision z of a floating point number is the number of binary digits of the fraction part. We call $m' = \lfloor m \cdot 2^z \rfloor \in \mathbf{Z}$ the fraction part (also called mantissa) of the floating point number a and we call e the exponent of a.

Definition: List representation of floating point numbers.

> $0 \in \mathbf{Q}$ is represented by the list $(0, 0)$.
> $a \in \mathbf{Q}$, $a \neq 0$ with $-\beta < e < \beta$, is represented by the list (e', m'). $e = e' \in \mathcal{A}$ and $m' \in \mathcal{I}$ are represented as atom respectively integer. e, m' defined as before.
> For $a \in \mathbf{Q}$ with $e \leq -\beta$ or $\beta \leq e$ there is no representation defined.

Recall that $\beta = 2^\zeta$ is a power of 2, $\zeta = 29$. The binary precision z is determined from the decimal precision d in the following way. Let

$$\rho = \lceil \log_\beta(10^d) \rceil$$

then

$$z = \lfloor \log_2(\beta^\rho) \rfloor - 1 = \frac{\rho}{\zeta} - 1.$$

The fraction part m' of a floating point number is then an integer with $\lceil \log_2(|m'|) \rceil = z$ and so $\lceil \log_\beta(|m'|) \rceil = \rho$.

Example:
Let $d = 20$, i.e. the desired decimal precision is 20 decimal digits. Then $\lceil \log_\beta(10^{20}) \rceil = 3 = \rho$ and we have $z = \zeta\rho - 1 = 29 \cdot 3 - 1 = 86$.

> $\frac{1}{2} = \frac{1}{2} \cdot 2^0$ is represented as $(0, (0, 0, 134217728))$, since $e = 0$, and $\lfloor \frac{1}{2} \cdot 2^z \rfloor$ $= 0\beta^0 + 0\beta^2 + 134217728\beta^2$ which is represented by $(0, 0, 134217728)$.
> $1 = \frac{1}{2} \cdot 2^1$ is represented by $(1, (0, 0, 134217728))$,
> $\frac{1}{4} = \frac{1}{2} \cdot 2^{-1}$ is represented by $(-1, (0, 0, 134217728))$
> and $-2 = \frac{-1}{2} \cdot 2^2$ is represented by $(2, (0, 0, -134217728))$.

6.3.1 Algorithms

The programs of the most important floating point algorithms and their complexity are summarized in the following.

Let $\mathcal{A}$ be the set of atoms, $\mathcal{L}$ be the set of lists, $\mathcal{O} = \mathcal{A} \cup \mathcal{L}$ be the set of objects, $\mathcal{I} = \{x \in \mathcal{O} : \ x \text{ represents an element of } \mathbf{Z}\}$ be the set of integers, $\mathcal{R} = \{x \in \mathcal{O} : \ x$ represents an element of $\mathbf{Q}\}$ be the set of rational numbers and $\mathcal{F} = \{x \in \mathcal{O} : \ x$ represents a floating point number$\}$ be the set of floating point numbers. Let $\rho = L(a)$ denote the

β–length of the fraction part m' of a floating point number a. The length of integers and rational numbers are defined as in the previous section. We will also write $L(a)$ for $O(L(a))$, i.e. we will not count for constant factors. The computing time functions t, t^+, t^-, t^* are defined as before in section 5.7.

$APSPRE(n)$ $n \in \mathcal{A}$. The precision of floating point numbers is set to n decimal digits. $t^+ = t^+_{\text{IEXP}} = n^2 = O(\rho^2)$, $c^+ = 2n = O(2\rho)$.

$b \leftarrow APFINT(a)$ $a \in \mathcal{I}$, $b \in \mathcal{F}$. $b = m'2^{e'}$ is the embedding of the integer a into the floating point numbers. Since a is shifted $t^+ = L(a) + |L(a) - L(b)|$ $= \max\{L(a), \rho\}$, $c^+ = \rho$, $t^- = L(a)$, $c^- = 0$.

$b \leftarrow APFRN(a)$ $a \in \mathcal{R}$, $b \in \mathcal{F}$. $b = m'2^{e'}$ is the embedding of the rational number a into the floating point numbers. The numerator and denominator of a are converted to floating point numbers, then their quotient is formed. $t^+ = 2t^+_{\text{APFINT}} + t^+_{\text{APQ}} = 2\max\{L(a), \rho\} + \rho^2$, $c^+ = 2\rho + \rho^2$.

$b \leftarrow RNFAP(a)$ $a \in \mathcal{F}$, $b \in \mathcal{R}$. $b = \frac{m}{2^{-e}}$ is the the rational number b which corresponds to the floating point number a. b is reduced to lowest terms. $t^+ = t^+_{\text{IEXP}} + t^+_{\text{RNRED}} = e^2 + \max\{\rho, \frac{e}{\zeta}\}^2$, $c^+ = e^2 + \max\{\rho, \frac{e}{\zeta}\}^2$.

$b \leftarrow APNEG(a)$ $a, b \in \mathcal{F}$. $b = -a$ is the negative of a. The fraction part of a is negated, so $t = t_{\text{INEG}} = \rho$, $c = \rho$.

$s \leftarrow APSIGN(a)$ $a \in \mathcal{F}$, $s \in \{-1, 0, +1\}$. $s = \text{sign}(a)$ is the sign of a. The integer sign of the fraction part of a is determined, so $t^+ = \rho$, $t^- = 1$, $c = 0$.

$b \leftarrow APABS(a)$ $a, b \in \mathcal{F}$. $b = |a|$ is the absolute value of a. The fraction part of a is possibly negated, so $t^+ = t^+_{\text{APNEG}} = \rho$, $c^+ = \rho$. $t^- = 1$, $c^- = 0$.

$s \leftarrow APCMPR(a, b)$ $a, b \in \mathcal{F}$, $s \in \{-1, 0, +1\}$. $s = \text{sign}(a - b)$ is the sign of the difference of a and b. The fraction part of a and b must be compared if the signs of the fraction parts and the exponents are equal, so $t^+ = \rho$, $t^- = 1$, $c = 0$.

$c \leftarrow APPROD(a, b)$ $a, b, c \in \mathcal{F}$. $c = a \cdot b$ is the product of a and b. The fraction parts of a and b are multiplied the result is then truncated, so $t = \rho^2$, $c = \rho^2$.

$c \leftarrow APQ(a, b)$ $a, b, c \in \mathcal{F}$. $c = a/b$ is the quotient of a and b. The fraction part of a is shifted by 2^z and then an integer quotient is computed, so $t = 2\rho^2$, $c = \rho^2$.

$c \leftarrow APSUM(a, b)$ $a, b, c \in \mathcal{F}$. $c = a + b$ is the sum of a and b. The fraction part of a or b is shifted to bring the decimal points in the same place, then the integers are added and normalized. Let e_a and e_b be the exponents of a and b respectively. Assume further that $|e_a - e_b| < z$, since otherwise nothing is to be done, then $t^+ = t^+_{\text{IMP2}} + t^+_{\text{ISUM}} + t^+_{\text{APFINT}}$ $= L(2^{|e_a - e_b|}) + 2\rho + 2\rho = O(2\rho)$. $c^+ = 2\rho$, $t^- = \rho$, $c^- = \rho$.

$c \leftarrow APDIFF(a,b)$ $a,b,c \in \mathcal{F}$. $c = a - b$ is the difference of a and b. b is negated, then the sum of a and $-b$ is computed. $t^+ = t^+_{\mathrm{APSUM}}$, $t^- = \rho$.

$b \leftarrow APEXP(a,n)$ $a,b \in \mathcal{F}$, $n \in \mathcal{A}$. $b = a^n$ is the n-th power of a (exponentiation). The binary exponentiation method is used as described with $IEXP$. Since the length of the products are always ρ we have to count the number of products and so $t^+ = \log_2(n)\rho^2$.

$b \leftarrow APROOT(a,n)$ $a,b \in \mathcal{F}$, $n \in \mathcal{A}$. $b = \sqrt[n]{a}$ is the n-th root of $|a|$. b is computed by Newtons method. The most expensive part is the computation of a^n, so $t^+ = \log_2(n)\rho^2$.

$a \leftarrow APPI()$ $a \in \mathcal{F}$. $a = \pi$. π is computed by the method of Salamin using the arithmetic–geometric mean approximation for an elliptic integral representation of π. $t^+ = \mathrm{const}\rho^2$, since square roots are computed.

$a \leftarrow APREAD()$ $a \in \mathcal{F}$. $APREAD$ is be defined as $APFRN(RNDRD())$ and the accepted syntax is that of $RNDWR$.

$APWRIT(a)$ $a \in \mathcal{F}$. The floating point number a is written to the current output stream. The syntax is:

```
rat = int "." unsigned-int
                [ "E" unsigned-beta-int ]
```

This syntax is accepted by $RNDRD$ for input.

This concludes the summary of floating point number arithmetic functions.

Note: When the precision of the floating point numbers is changed, then the already existing numbers are not automatically converted to the new precision. The conversion can be accomplished by first converting the floating point number to a rational number. Then change the precision and finally reconvert the rational number to a floating point number.

Example: Assume the current decimal precision is d_2 and we want to convert numbers which are represented in precision d_1. The following algorithm does the conversion:

```
PROCEDURE Crep(a,d1,d2);
(*Change the representation of the
floating point number a, with precision d1 to
precision d2.*)
VAR   r: LIST;
BEGIN
(*1*) (*set old precision and convert. *)
      APSPRE(d1); r:=RNFAP(a);
(*2*) (*set new precision and convert. *)
      APSPRE(d2); r:=APFRN(r);
      RETURN(r);
(*3*) END Crep.
```

For illustration we list the algorithms APFINT and APSUM in Modula–2 in MAS. The function of the algorithms should be clear from the step comments and the floating point number representation discussed before.

```
PROCEDURE APFINT(N: LIST): LIST;
(*Arbitrary precision floating point from integer.
The integer N is converted to the arbitrary precision
floating point number A.*)
VAR   A, EL, FL, ML: LIST;
BEGIN
(*1*) (*n=0.*)
      IF N = 0 THEN A:=APCOMP(0,0); RETURN(A); END;
(*2*) (*normalize.*) EL:=ILOG2(N); FL:=EL-1-APPR2; (*1=log2(2).*)
      IF FL >= 0
         THEN ML:=IDP2(N,FL); (*truncate*)
         ELSE ML:=IMP2(N,-FL); (*fill up*) END;
(*3*) (*round.*) ML:=ISUM(ML,1);
      ML:=IDP2(ML,1);
(*4*) (*finish.*) A:=APCOMP(ML,EL); RETURN(A);
(*6*) END APFINT;
```

APCOMP denotes the composition of an exponent and the fraction part of a floating point number. ILOG2 means the integer logarithm base 2, IDP2 denotes 'integer division by power of 2' and IMP2 denotes 'integer multiplication by power of 2'. All three algorithms exploit the β representation of integers and the fact, that β is itself a power of 2. ISUM denotes integer sum. APPR2 means the number z of binary digits of the representation.

```
PROCEDURE APSUM(A,B: LIST): LIST;
(*Arbitrary precision floating point sum.
A, B and C are arbitrary precision floating point numbers.
C is the sum of A and B. C=A+B.*)
VAR   C, EL, EL1, EL2, ML, ML1, ML2: LIST;
BEGIN
(*1*) (*A or B zero.*)
      ML1:=APMANT(A); ML2:=APMANT(B);
      IF ML1 = 0 THEN RETURN(B) END;
      IF ML2 = 0 THEN RETURN(A) END;
(*2*) (*check exponent range.*)
      EL1:=APEXPT(A); EL2:=APEXPT(B);
      EL:=MASABS(EL1-EL2);
      IF EL > APPR2 THEN
         IF EL1 > EL2 THEN RETURN(A) ELSE RETURN(B) END;
         END;
(*3*) (*normalize mantisa and add.*)
      EL:=IMIN(EL1,EL2);
      ML1:=IMP2(ML1,EL1-EL); ML2:=IMP2(ML2,EL2-EL);
      ML:=ISUM(ML1,ML2); C:=APFINT(ML);
(*4*) (*shift.*) EL:=EL-APPR2; C:=APSHFT(C,EL);
(*5*) (*finish.*) RETURN(C);
(*6*) END APSUM;
```

APEXPT and APMANT extract the exponent and the fraction part (mantissa) of a floating point number. MASABS determines the absolute value of the argument. IMP2, IMIN and ISUM denote integer multiplication by power of 2, integer minimum and integer sum respectively. APSHFT adds the second parameter to the exponent of the floating point number and checks the exponent for overflow or underflow.

6.3.2 Exercises

1. Let $\mathbf{R}$ denote the real numbers. Use Newtons method to write an algorithm to approximate a zero of a function $f : \mathbf{R} \longrightarrow \mathbf{R}$ up to a desired precision $\varepsilon > 0$. With this algorithm compute a zero of the function $f(x) = x^2 - 2$ up to 50 decimal digits.

The Newton iteration is defined as:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad \text{for} \quad x_n \in \mathbf{R},\ n \in \mathbf{N}.$$

Recall the properties of the Newton iteration:

Proposition: Let $D = [a, b] \subset \mathbf{R}$ be a closed and bounded interval in the real numbers and let $f : D \longrightarrow \mathbf{R}$ be a two times continuous differentiable function on D with

1. $f(a) \cdot f(b) < 0$,
2. $f'(\xi) \neq 0$ for all $\xi \in D$,
3. $f''(\xi) \geq 0$ or $f''(\xi) \leq 0$ for all $\xi \in D$.

If further $x_1 \in D$, then the Newton sequence $\{x_n\}_{n \in \mathbf{N}}$ converges for all $x_0 \in (a, b)$ monotonous against the unique zero ξ of f.

With the conditions of the proposition we obtain three terminating conditions:

1. If n is greater than a maximal allowable number of iterations, then condition 1) is not fulfilled in the neighbourhood of x_0, i.e. there is possibly no zero near x_0.
2. If $f'(x_n) < \varepsilon$, then condition 2) is not fulfilled i.e. there is possibly a singularity near x_n.
3. $|x_{n+1} - x_n| < \varepsilon$ i.e. $|\frac{f(x_n)}{f'(x_n)}| < \varepsilon$. Then x_{n+1} is an approximation for ξ.

With this information the algorithm can be formulated as follows:

```
dig:=50. APSPRE(dig).
AbsErr:=APFRN(RNRED(1,IEXP(10,dig/2))).
MaxIter:=100.

PROCEDURE Newton(f,fp,x);
(*Newton iteration. f and fp are functions.
x is the starting value for iteration. A fix point
of x-f(x)/fp(x) is returned. *)
VAR   i, y, z, zp, w: ANY;
BEGIN
(*1*) i:=0; y:=x;
(*2*) WHILE i < MaxIter DO i:=i+1;
            z:=f(y); zp:=fp(y);
            IF APCMPR(APABS(zp),AbsErr) <= 0 THEN
               CLOUT("Derivation becommes zero.");
               BLINES(1); RETURN(y) END;
```

```
            w:=APQ(z,zp); y:=APDIFF(y,w);
            IF APCMPR(APABS(z),AbsErr) <= 0 THEN
               RETURN(y) END;
            END;
      CLOUT("Maximal number of iterations reached.");
      BLINES(1); RETURN(y);
(*9*) END Newton.
```

The maximal allowed number of iterations is set to 100 (variable `MaxIter`). The absolute error is to be no greater than 10^{-50} (variable `AbsErr`). If one of the terminating conditions 1) or 2) is reached, a message is printed and the computation is stopped. Otherwise the algorithm terminates by condition 3) and returns an approximation of the zero of f.

The function $f(x) = x^2 - 2$ with derivation $f'(x) = 2x$ can be formulated as algorithm as follows:

```
zwei:=APFINT(2).

PROCEDURE E(x);
(*Expression function. An expression is evaluated at x. *)
VAR   y: ANY;
BEGIN
(*1*) (* x**2 - 2 *)
      y:=APEXP(x,2); y:=APDIFF(y,zwei);
      RETURN(y);
(*9*) END E.

PROCEDURE Ep(x);
(*Expression function derivation. The derivation of
an expression is evaluated at x. *)
VAR   y: ANY;
BEGIN
(*1*) (* 2 x *)
      y:=APPROD(x,zwei);
      RETURN(y);
(*9*) END Ep.
```

A sample output follows:

```
start:=APFINT(1).
{0 sec} ANS: (1 (0 0 0 0 0 134217728))

b:=Newton(E,Ep,start).
{6 sec} ANS: (1 (202854696 513744239 228305493 18243426
                 133414089 189812531))

BEGIN CLOUT("AbsErr = "); APWRIT(AbsErr); BLINES(0);
      CLOUT("Result = "); APWRIT(b); BLINES(0);
      CLOUT("W2     = "); APWRIT(APROOT(zwei,2));
      BLINES(0) END.
```

```
AbsErr = 0.10000000000000000000000000000000000000
           000000000000000E-24

Result = 0.14142135623730950488016887242096980785
           696718753769480E1

W2     = 0.14142135623730950488016887242096980785
           696718753769480E1
```

First the starting point for the iteration is set to 1. Then the function `Newton` is called with the function `E`, its derivation `Ep` and the starting point as input. The computation needs 6 seconds on an Atari ST. Finally the absolute error, the zero and for comparison the square root of 2 are printed.

Chapter 7

Polynomial Systems

Besides the integers and rational numbers the most important data types are polynomials. Polynomials are always represented in some (internal) canonical form and not as general LISP S–expressions. The most important canonical representations are:

- recursive representation,
- distributive (or distributed) representation,
- dense representation.

These representations will be discussed in the following sections. For every representation there are algorithms to read and write polynomials, select parts of polynomials, construct polynomials and to perform basic arithmetic of polynomials (like sum, product, remainder, evaluation, substitution).

For more advanced methods like polynomial greatest common divisors or multivariate polynomial factorization there are algorithms for the recursive polynomial repesentation. For Gröbner bases and polynomial ideal decomposition or solving systems of polynomial equations there are algorithms for the distributive polynomial representation. The dense representation is mainly used for algorithms for fast univariate polynomial remainder computations.

There is a varity of application dependent 'fine tunings' of representations to optimize space, time or programming complexity of the algorithms which are not discussed here (and which are only partly available in the current system).

Program libraries are composed from the ALDES / SAC-2 system [Collins, Loos 1982] and from the DIP system [Gebauer, Kredel 1983] which is based on the former. The collection of algorithms and global variables are called 'systems'. The systems are broken into modules according to specific characteristics of subcollections of the algorithms.

The available ALDES / SAC–2 polynomial libraries are the following:

ALDES / SAC–2 Polynomial System,
ALDES / SAC–2 Algebraic Number System,
ALDES / SAC–2 Polynomial GCD and Resultant System,
ALDES / SAC–2 Polynomial Factorization System,
ALDES / SAC–2 Real Root System.

The available DIP polynomial libraries are the following:

DIP Common Distributive Polynomial System,
DIP Distributive Integral Polynomial System,
DIP Distributive Rational Polynomial System,
DIP Distributive Arbitrary Domain Polynomial System,
DIP Buchberger Algorithm System (Gröbner bases),
DIP Polynomial Ideal Dimension System,
DIP Zero–dimensional Polynomial Ideal Decomposition System,
DIP Zero–dimensional Polynomial Ideal Real Root System.

As extension to the DIP system there are libraries for non–commutative polynomial rings of solvable type:

MAS Non–commutative Rational Distributive Polynomial System,
MAS Non–commutative Gröbner Base System,
MAS Non–commutative Polynomial Center System.

Further libraries are

MAS Comprehensive Gröbner Base System,
MAS Syzygy and Module Gröbner Base System,

7.1 Coefficient Rings

Although the representation of polynomials is independent of the representation of the coefficients the algorithms are implemented for specific coefficient rings.

Programms that work independently of the coefficient ring start with the program prefix 'P' in case of the recursive polynomial representation and with 'DI' in case of the distributive polynomial representation.

For the recursive representation there are algorithms for the following coefficient rings:

- integral numbers: $\mathbf{Z}$, program prefix 'IP' for 'integral polynomial'
- rational numbers: $\mathbf{Q}$, program prefix 'RP' for 'rational number polynomial'
- integral numbers modulo m: $\mathbf{Z}/_{(m)}$, program prefix 'MIP' for 'modular integral polynomial'
- algebraic numbers over the rational numbers: $\mathbf{Q}[X]/_{(m_\alpha(X))}$, where $m_\alpha(X)$ denotes the minimal polynomial of α over $\mathbf{Q}$, program prefix 'AFP' for 'algebraic number field polynomial'.

In the section on the recursive polynomial representation we will discuss only polynomials over the integers.

For the distributive representation there are algorithms for the following coefficient rings:

- integral numbers: $\mathbf{Z}$, program prefix 'DIIP' for 'distributive integral polynomial'

- rational numbers: $\mathbf{Q}$, program prefix 'DIRP' for 'distributive rational number polynomial'

In the so called 'distributive arbitrary domain polynomial system' there are algorithms for further coefficient rings. The common program prefix is 'DIP' for 'distributive arbitrary domain polynomial'.

- integral numbers modulo m: $\mathbf{Z}/_{(m)}$,
- rational complex numbers,
- rational quaternion numbers (yielding a non-commutative polynomial ring),
- rational octonion numbers (yielding a non-commutative, non-associative polynomial ring),
- finite field numbers $GF(p,n)$, $n > 1$,
- algebraic numbers over the rational numbers: $\mathbf{Q}[X]/_{(m_\alpha(X))}$, where $m_\alpha(X)$ denotes the minimal polynomial of α over $\mathbf{Q}$,
- polynomial functions in n variables over the integers: $\mathbf{Z}[X_1, \ldots, X_n]$,
- rational functions in n variables over the integers: $\mathrm{Quot}(\mathbf{Z}[X_1, \ldots, X_n]) \cong \mathbf{Q}(X_1, \ldots, X_n)$,
- rational polynomials in n variables modulo a polynomial ideal: $\mathbf{Q}[X_1, \ldots, X_n]/_{(m_1, \ldots, m_k)}$.

In the section on the distributive polynomial representation we will discuss only polynomials over the rational numbers.

7.2 Recursive Polynomial System

Let R be a commutative ring with 1 and let $S = R[X_1, \ldots, X_r]$ denote a (commutative) polynomial ring in $r \geq 0$ variables (indeterminates) $X_1, \ldots, X_r$. S is isomorphic to an univariate polynomial ring over a polynomial ring with one less variable:

$$S' = (\ldots((R[X_1])[X_2])\ldots)[X_r].$$

The elements of S' are univariate polynomials in the **main variable** X_r with coefficients being polynomials in the ring $(\ldots(R[X_1])\ldots)[X_{r-1}]$ when $r \geq 1$.

Definition: Let $A(X_1, \ldots, X_r) \in S'$, $A \neq 0$ and $r \geq 1$, then

$$A(X_1, \ldots, X_r) = \sum_{i=1}^{k} A_i(X_1, \ldots, X_{r-1}) X_r^{e_i}$$

with $A_i \neq 0$ for $i = 1, \ldots, k$ and $e_k > e_{k-1} > \ldots > e_2 > e_1$. The **recursive representation** of A is the list

$$\alpha = (e_k, \alpha_k, \ldots, e_2, \alpha_2, e_1, \alpha_1)$$

where the α_i denote the recursive representations of the A_i and the e_i are non–negative β–integers, $i = 1, \ldots, k$. If $A = 0$ then $\alpha = 0$ and if $r = 0$ then α is defined by the representation of the base coefficient ring.

Notes:

1. The variables $X_1, \ldots, X_r$ are not stored in the representing list. This is different to other computer algebra systems like REDUCE or muMATH.
2. The representation is sparse in the sence, that only coefficients $\neq 0$ are stored.

Examples:

1. Let $S = \mathbf{Z}[X]$, that is $R = \mathbf{Z}$ and $r = 1$. Let

   $$A = 3X^4 + 5,$$

   then $k = 2$ and $e_2 = 4, A_2 = 3, e_1 = 0, A_1 = 5$. The representation is then

   $$\alpha = (4, 3, 0, 5).$$

2. Let $S = \mathbf{Z}[X, Y]$, that is $R = \mathbf{Z}$ and $r = 2$. Let

   $$A = (3X + 2)Y^2 + 5X,$$

   then $k = 2$ and $e_2 = 2, A_2 = 3X + 2, e_1 = 0, A_1 = 5X$. The representation is then

   $$\alpha = (2, (1, 3, 0, 2), 0, (1, 5)).$$

3. Let $S = \mathbf{Q}[X, Y]$, that is $R = \mathbf{Q}$ and $r = 2$. Let

   $$A = \frac{1}{4}X^2Y - \frac{3}{5}$$

   then $k = 2$ and $e_2 = 1, A_2 = \frac{1}{4}X^2, e_1 = 0, A_1 = \frac{-3}{5}X^0$. The representation is then

   $$\alpha = (1, (2, (1, 4)), 0, (0, (-3, 5))).$$

7.2.1 Algorithms

The programs of the most important recursive polynomial algorithms and their complexity are summarized in the following. First the main complexity numbers are defined and then integral polynomial programs are discussed.

As the recursive definition of the polynomial ring and the recursive representation suggests the algorithms will be constructed in the following way:

1. Check for recursion base, case $r = 0$. Perform the desired operations on the base coefficients.
2. If $r \geq 1$ then loop on the exponents in the main variable and call the algorithm recursively on the coefficients. Construct resulting polynomials.

3. Return the results.

The complexity of polynomial algorithms will therefore depend mainly on three factors:

1. the size of the base coefficients,
2. the degree of the polynomials and
3. the number of variables.

These quantities are defined precisely as follows.
Let $A \in S = R[X_1, \ldots, X_r]$, $r \geq 0$, $A(X_1, \ldots, X_r) = \sum_{i=1}^{k} A_i(X_1, \ldots, X_{r-1}) X_r^{e_i}$. Then $\deg_i(A)$ denotes the degree of the polynomial A in the variable X_i for $i = 1, \ldots, r$. That is

$$\deg_i(A) = \begin{cases} 0 & \text{if } A = 0 \text{ or } r = 0 \\ e_k & \text{if } i = r \\ \max\{\deg_i(A_j) \mid j = 1, \ldots, k\} & \text{otherwise.} \end{cases}$$

Further define $d = \deg(A) = \max\{\deg_i(A) \mid i = 1, \ldots, r\}$.
With $L(A)$ we will denote the maximum of the length of the *base coefficients*. That is

$$L(A) = \begin{cases} L(A_1) & \text{if } r = 0 \\ \max\{L(A_j) \mid j = 1, \ldots, k\} & \text{otherwise.} \end{cases}$$

The length of integers and rational numbers are defined as in section 5.7. Further let $L(A, B) = \max\{L(A), L(B)\}$.
Since only coefficients $\neq 0$ are stored in the polynomials one would like to measure the number of terms in the polynomials. This gives a more precise indication of the complexity in view of the size of the base coefficients. So we define

$$\text{term}(A) = \begin{cases} 1 & \text{if } r = 0 \\ \sum_{j=1}^{k} \text{term}(A_j) & \text{otherwise.} \end{cases}$$

Clearly it is bounded by

$$\text{term}(A) \leq (\deg(A) + 1)^r = (d + 1)^r.$$

We will continue to write $L(a)$ for $O(L(a))$, i.e. we will not count for constant factors. The computing time functions t, t^+, t^-, t^* are defined as before in section 5.7.

Let $\mathcal{A}$ be the set of atoms, $\mathcal{L}$ be the set of lists, $\mathcal{O} = \mathcal{A} \cup \mathcal{L}$ be the set of objects, $\mathcal{I} = \{x \in \mathcal{O} : \ x \text{ represents an element of } \mathbf{Z}\}$ be the set of integers, $\mathcal{P}_r = \{x \in \mathcal{O} : \ x$ represents a multivariate polynomial in r variables $\}$ be the set of recursive polynomials, $\mathcal{IP}_r = \{x \in \mathcal{O} : \ x$ represents a multivariate polynomial over $\mathbf{Z}$ in r variables $\}$ be the set of integral polynomials.

We will first summarize selector functions which are independent of the base coefficient ring and then turn to the algorithms for integral polynomials. For decomposition and construction of polynomials the list processing functions $ADV2$, $FIRST2$ and $COMP2$ are used and will not be discussed here.

$e \leftarrow PDEG(A)$ $\quad$ $A \in \mathcal{P}_r$, $e \in \mathcal{A}$. $e = \deg_r(A) = e_k$ is the degree of A in the main variable. In recursive representation $t = 1$, $c = 0$.

$a \leftarrow PLDCF(A)$ $A \in \mathcal{P}_r$, $a \in \mathcal{P}_{r-1}$. $a = \mathrm{ldcf}(A) = A_k$ is the leading coefficient of A. In recursive representation $t = 2$, $c = 0$.

$A' \leftarrow PRED(A)$ $A \in \mathcal{P}_r$, $A' \in \mathcal{P}_r$. $A' = \mathrm{red}(A) = \sum_{i=1,\ldots,k-1} A_i X_r^{e_i}$ is the polynomial reductum of A. $t = 2$, $c = 0$.

$a \leftarrow PTRCF(A)$ $A \in \mathcal{P}_r$, $a \in \mathcal{P}_{r-1}$. $a = \mathrm{trcf}(A) = A_1$ is the trailing coefficient of A. $t^+ = 2\deg_r(A)$, $c = 0$. The factor 2 comes in because also the exponents are stored in the representing list of a polynomial.

$a \leftarrow PLBCF(r, A)$ $A \in \mathcal{P}_r$, $a \in \mathcal{P}_0$. a is the leading base coefficient of A defined as

$$a = \mathrm{lbcf}(r, A) = \begin{cases} \mathrm{ldcf}(A) & \text{if } r = 1 \\ \mathrm{lbcf}(\mathrm{ldcf}(A)) & \text{if } r > 1. \end{cases}$$

$t = 2r$, $c = 0$.

$a \leftarrow PTBCF(r, A)$ $A \in \mathcal{P}_r$, $a \in \mathcal{P}_0$. a is the trailing base coefficient of A defined as

$$a = \mathrm{tbcf}(r, A) = \begin{cases} \mathrm{trcf}(A) & \text{if } r = 1 \\ \mathrm{tbcf}(\mathrm{trcf}(A)) & \text{if } r > 1. \end{cases}$$

$t^+ = 2r\deg(A)$, $c = 0$.

We turn now to the discussion of polynomial algorithms which depend on the base coefficient field. Only integer base coefficients will be treated.

$s \leftarrow IPSIGN(r, A)$ $A \in \mathcal{IP}_r$, $s \in \{-1, 0, +1\}$. $s = ISIGN(\mathrm{lbcf}(A))$ is the sign of the leading base coeficient of A. In recursive representation $t^+ = 2r + t^+_{ISIGN} = r + L(A)$, $c = 0$; $t^- = 2r + t^-_{ISIGN} = r + 1 = O(r)$.

$A' \leftarrow IPNEG(r, A)$ $A, A' \in \mathcal{IP}_r$. $A' = -A$ is the negation of A. $t^+ = \mathrm{term}(A) \cdot t^+_{INEG} \leq \deg(A)^r L(A) = d^r L(A)$, $c^+ = d^r L(A)$.

$A' \leftarrow IPABS(r, A)$ $A, A' \in \mathcal{IP}_r$. If $\mathrm{sign}(\mathrm{lbcf}(A)) = -1$ then $A' = -A$ else $A' = A$. $t^- = t^-_{IPSIGN} = r$, $c^- = 0$; $t^+ = t^+_{IPNEG} \leq d^r L(A)$, $c^+ = c^+_{IPNEG} \leq d^r L(A)$.

$C \leftarrow IPSUM(r, A, B)$ $A, B, C \in \mathcal{IP}_r$. $C = A + B$ is the sum of A and B. Let $l = \max\{\mathrm{term}(A), \mathrm{term}(B)\}$, $d = \max\{\deg(A), \deg(B)\}$ then $t^+ = l \cdot t^+_{ISUM} \leq d^r L(A, B)$, $c^+ = d^r L(A, B)$.

$C \leftarrow IPDIF(r, A, B)$ $A, B, C \in \mathcal{IP}_r$. $C = A - B$ is the difference of A and B. The computing times are the same as for $IPSUM$.

$C \leftarrow IPPROD(r, A, B)$ $A, B, C \in \mathcal{IP}_r$. $C = A * B$ is the product of A and B. Let $l = \max\{\mathrm{term}(A), \mathrm{term}(B)\}$, $d = \max\{\deg(A), \deg(B)\}$ then $t^+ = l^3 \cdot t^+_{IPROD} \leq d^{2r} L(A, B)^2$, $c^+ = d^{2r} L(A, B)^2$.

$IPQR(r, A, B; C, D)$ $A, B, C, D \in \mathcal{IP}_r$. $C = A/B$ is the quotient of A and B if it exists, in this case $D = 0$ and $A = CB$. (A sufficient condition is that $\mathrm{ldcf}(B)$ is a unit in the coefficient ring.) If the quotient does not exist, then D is a polynomial of minimal degree (not necessarily $\deg_r(D) < \deg_r(B)$, but $\deg_r(D) \leq \deg_r(A)$) such that $A = CB + D$. Assume that the quotient exists, then the computing time is proportional to that of the product of C and B. The method used is also called **trial division**, since it is not a priori guaranteed that it succeeds.

$C \leftarrow IPPSR(r, A, B)$ $A, B, C \in \mathcal{IP}_r$. C is the pseudo remainder of A and B. The pseudo remainder allways exists and is defined as

$$\mathrm{ldcf}(B)^{\delta} \cdot A = Q \cdot B + C$$

where $\delta \geq \deg_r(A) - \deg_r(B)$ and $\deg_r(C) < \deg_r(B)$. The computing time is proportional to that of the product of Q and B.

$IPREAD(; r, A, V)$ $A \in \mathcal{IP}_r$, V a variable list. The polynomial A, the number of variables r and the variable list V are read from the input stream. The accepted syntax is:

```
coeff = ( integer | poly )
term  = coeff "*" identifier "**" atom
poly  = "(" term { ( "+" | "-" ) term } ")"
```

With the context conditions:

1. all `identifier`s in the `term`s of a `poly`nomial must be equal,
2. the `atom`s must be non–negative and given in strictly decreasing order.

Note further that there are no optional parts in this definition ! Due to this restricted syntax it is often more convienient to use distributed representation for input and output and to convert the polynomials between the representations.

$IPWRIT(r, A, V)$ $A \in \mathcal{IP}_r$, V a variable list. The polynomial A is written to the output stream. The output syntax is equal to the input syntax of $IPREAD$.

This concludes the summary of integral polynomial arithmetic functions.

For illustration we list the algorithms `IPPROD` and `IPPGSD` in Modula–2 in MAS. The function of the algorithms should be clear from the step comments and polynomial representation discussed before.

```
PROCEDURE IPPROD(RL,A,B: LIST): LIST;
(*Integral polynomial product.  A and B are integral
polynomials in r variables, r ge 0.  C=A*B.*)
VAR  AL, AP, AS, BL, BS, C, C1, CL, EL, FL, RLP: LIST;
BEGIN
(*1*) (*a or b zero.*)
      IF (A = 0) OR (B = 0) THEN C:=0; RETURN(C); END;
(*2*) (*rl=0.*)
      IF RL = 0 THEN C:=IPROD(A,B); RETURN(C); END;
(*3*) (*general case.*) AS:=CINV(A); BS:=CINV(B); C:=0;
      RLP:=RL-1;
      REPEAT ADV2(BS, BL,FL,BS); AP:=AS; C1:=SIL;
             REPEAT ADV2(AP, AL,EL,AP);
                    IF RLP = 0 THEN CL:=IPROD(AL,BL);
                       ELSE CL:=IPPROD(RLP,AL,BL); END;
                    C1:=COMP2(EL+FL,CL,C1);
                    UNTIL AP = SIL;
             C:=IPSUM(RL,C,C1);
             UNTIL BS = SIL;
      RETURN(C);
(*6*) END IPPROD;
```

The constructors and selectors for polynomials are the list processing functions `COMP2` and `ADV2`. `CINV` means the constructive inverse list of its argument. `IPROD` denotes the integer product. `IPSUM` is the polynomial sum.

```
PROCEDURE IPPGSD(RL,A: LIST): LIST;
(*Integral polynomial primitive greatest squarefree divisor.
A is an integral polynomial in r variables.  If A=0 then B=0.
Otherwise B is the greatest squarefree divisor of the primitive
part of A.*)
VAR  B, BP, C, D: LIST;
BEGIN
(*1*) (*a=0.*)
      IF A = 0 THEN B:=0; RETURN(B); END;
(*2*) (*a ne 0.*) B:=IPPP(RL,A);
      IF FIRST(B) > 0 THEN BP:=IPDMV(RL,B);
         IPGCDC(RL,B,BP, C,B,D); END;
      RETURN(B);
(*5*) END IPPGSD;
```

`IPPP` denotes the algorithm which computes the primitive part of its argument. `IPDMV` stands for derivation in the main variable and `IPGCDC` means greatest common divisor and cofactors. In abuse `FIRST` is used to determine the degree of the polynomial `B`.

7.2.2 Exercises

Since in the recursive polynomial system are only clumsy input / output facilities, we defer the exercises to the next section. There we will discuss also the conversions between the polynomial representations.

7.3 Dense Polynomial System

A short description of the data structures of the dense polynomial system is contained in this section. No algorithms and no exercises will be presented.

Again let R be a commutative ring with 1 and let $S = R[X_1, \ldots, X_r]$ denote a (commutative) polynomial ring in $r \geq 0$ variables (indeterminates) $X_1, \ldots, X_r$. S is isomorphic to an univariate polynomial ring over a polynomial ring with one less variable:

$$S' = (\ldots((R[X_1])[X_2])\ldots)[X_r].$$

The elements of S' are univariate polynomials in the **main variable** X_r with coefficients being polynomials in the ring $(\ldots(R[X_1])\ldots)[X_{r-1}]$ when $r \geq 1$.

Definition: Let $A(X_1, \ldots, X_r) \in S'$, $A \neq 0$ and $r \geq 1$, then

$$A(X_1, \ldots, X_r) = \sum_{i=0}^{k} A_i(X_1, \ldots, X_{r-1}) X_r^i$$

with $A_k \neq 0$. The **dense representation** of A is the list

$$\alpha = (k, \alpha_k, \ldots, \alpha_1, \alpha_0)$$

where the α_i denote the dense representations of the A_i, $i = 0, \ldots, k$. If $A = 0$ then $\alpha = 0$ and if $r = 0$ then α is defined by the representation of the base coefficient ring.

Notes:

1. The variables $X_1, \ldots, X_r$ are not stored in the representing list. This is different to other computer algebra systems.
2. The representation is dense in the sence, that all coefficients, even those which are $= 0$, are stored.

Examples:

1. Let $S = \mathbf{Z}[X]$, that is $R = \mathbf{Z}$ and $r = 1$. Let

   $$A = 3X^4 + 5,$$

   then $k = 4$, $A_4 = 3, A_0 = 5$ the other $A_i = 0$. The representation is then

   $$\alpha = (4, 3, 0, 0, 0, 5).$$

2. Let $S = \mathbf{Z}[X, Y]$, that is $R = \mathbf{Z}$ and $r = 2$. Let

   $$A = (3X + 2)Y^2 + 5X,$$

   then $k = 2$, $A_2 = 3X + 2, A_1 = 0, A_0 = 5X$. The representation is then

   $$\alpha = (2, (1, 3, 2), 0, (1, 5, 0)).$$

3. Let $S = \mathbf{Q}[X, Y]$, that is $R = \mathbf{Q}$ and $r = 2$. Let

   $$A = \frac{1}{4}X^2Y - \frac{3}{5}$$

   then $k = 1$, and $A_1 = \frac{1}{4}X^2, A_0 = \frac{-3}{5}X^0$. The representation is then

   $$\alpha = (1, (2, (1, 4), 0, 0), (0, (-3, 5))).$$

We will not discuss algorithms for the dense polynomial representation since they are only used in special situations.

7.4 Distributive Polynomial System

Again let R be a commutative ring with 1 and let $S = R[X_1, \ldots, X_r]$ denote a (commutative) polynomial ring in $r \geq 0$ variables (indeterminates) $X_1, \ldots, X_r$. The elements of S are sums of **monomials**, where each monomial is a product of a **base coefficient** and a **term** (power product).

Definition: Let $A(X_1, \ldots, X_r) \in S$, $A \neq 0$ and $r \geq 1$, then

$$A(X_1, \ldots, X_r) = \sum_{i=1}^{k} a_i X_1^{e_{i1}} \cdot \ldots \cdot X_r^{e_{ir}} = \sum_{i=1}^{k} a_i X^{e_i}$$

with $a_i \neq 0$ for $i = 1, \ldots, k$ and natural numbers e_{ij} for $i = 1, \ldots, k$ and $j = 1, \ldots, r$. X^{e_i} is an abreviation for $X_1^{e_{i1}} \cdot \ldots \cdot X_r^{e_{ir}}$. k is the number of terms of A. For $r > 0$ the representation of an **exponent vector** $e_i = (e_{i1}, \ldots, e_{i,r-1}, e_{ir})$ is the list

$$\epsilon_i = (e_{ir}, \ldots, e_{i2}, e_{i1}).$$

For $r = 0$ let $\epsilon = ()$, the empty list. The **distributive representation** of A is the list

$$\alpha = (\epsilon_k, \alpha_k, \ldots, \epsilon_2, \alpha_2, \epsilon_1, \alpha_1)$$

where the α_i denote the representations of the a_i and the ϵ_i are the representation of the exponent vectors, $i = 1, \ldots, k$. If $A = 0$ then $\alpha = 0$ and if $r = 0$ then $\alpha = ((), \alpha_1)$.

Notes:

1. The variables $X_1, \ldots, X_r$ are not stored in the representing list. This is different to other computer algebra systems like REDUCE or muMATH.
2. The representation is sparse in the sence, that only base coefficients $\neq 0$ are stored.
3. The representation of the exponent vectors is dense in the sence, that also exponents $= 0$ are stored.
4. The number of variables r can be determined from the length of an exponent vector (representation).

Examples:

1. Let $S = \mathbf{Z}[X]$, that is $R = \mathbf{Z}$ and $r = 1$. Let

   $$A = 3X^4 + 5,$$

   then $k = 2$ and $e_2 = (4), a_1 = 3, e_1 = (0), a_2 = 5$. The representation is then

   $$\alpha = ((4), 3, (0), 5).$$

2. Let $S = \mathbf{Z}[X, Y]$, that is $R = \mathbf{Z}$ and $r = 2$. Let

   $$A = (3X + 2)Y^2 + 5X = 3XY^2 + 2Y^2 + 5X,$$

   then $k = 3$ and $e_3 = (1, 2), a_3 = 3, e_2 = (0, 2), a_2 = 2, e_1 = (1, 0), a_1 = 5$. The representation is then

   $$\alpha = ((2, 1), 3, (2, 0), 2, (0, 1), 5).$$

3. Let $S = \mathbf{Q}[X, Y]$, that is $R = \mathbf{Q}$ and $r = 2$. Let

$$A = \frac{1}{4}X^2Y - \frac{3}{5},$$

then $k = 2$ and $e_2 = (2, 1), a_2 = \frac{1}{4}, e_1 = (0, 0), a_1 = \frac{-3}{5}$. The representation is then

$$\alpha = ((1, 2), (1, 4), (0, 0), (-3, 5)).$$

4. Let $S = \mathbf{Z}[X_1, X_2, X_3, X_4, X_5]$, that is $R = \mathbf{Z}$ and $r = 5$. Let

$$A = 5X_2X_3 + 7X_1^2,$$

then $k = 2$ and $e_2 = (0, 1, 1, 0, 0), a_2 = 5, e_1 = (2, 0, 0, 0, 0), a_1 = 7$. The representation is then

$$\alpha = ((0, 0, 1, 1, 0), 5, (0, 0, 0, 0, 2), 7).$$

Up to now we have assumed that there is a unique way to determine the ordering and the sequence of the terms in a polynomial. In general there are many (contiuum many) total orderings of the set of terms, which are in addition compatible with the multiplication of terms. Such term orders are called admissible. We will now turn to the explicit definition of some important term orders and then give a general characterization by linear forms.

7.4.1 Term Orders

Recall that T denotes the set of terms $u = X_1^{e_1} \cdot \ldots \cdot X_r^{e_r} = X^e$ where the $e_i, (i = 1, \ldots, r)$ are nonnegative integers.

The degree of a term $u = X_1^{e_1} \cdot \ldots \cdot X_r^{e_r}$ is defined as $deg(u) = \sum_{i=1}^{r} e_i$. For exponent vectors e and l define $e - l = (e_1 - l_1, \ldots, e_r - l_r)$ (an vector of integers). The degree of an exponent vector is defined as $\deg(e) = \deg(X^e)$.

For an arbitrary total ordering $>_T$ on power products we define $u >_T v \iff X^e >_T X^l \iff e >_T l \iff e - l >_T 0 = (0, \ldots, 0)$. We write $u <_T v$ if $v >_T u$. The term orderings are called **admissible**, if they are monoton with respect to multiplication of power products and if $1 = X^0$ is the smallest element. That is

1. For $u \in T$, $u \neq 1$ it holds $1 <_T u$.
2. For $u, v, t \in T$ with $u <_T v$ it holds $ut <_T vt$.

In the sequel $>$ denotes the natural ordering on the integers. The currently implemented term orders can be checked with the procedure `EVOWRITE`.

Buchberger's term order

In 1965 Buchberger gave the definiton of the term ordering used by him for Gröbner bases calculation [Buchberger 1965, Buchberger 1970]. It can be defined as:

$$e >_B 0 \iff deg(e) > 0 \text{ or } \begin{cases} deg(e) = 0 \\ \text{and} \\ e >_{BL} 0 \end{cases}$$

Where $>_{BL}$ is defined in the following way:

$$e >_{BL} 0 \Longleftrightarrow \begin{cases} \exists k \in \{1,\ldots,r\} \\ \text{with } e_j = 0 \text{ for } j = 1,\ldots,k-1 \\ \text{and } e_k < 0 \end{cases}$$

Note that $>_{BL}$ alone is not an admissible term order. This ordering was implemented by Winkler in the SAC-1 computer algebra system [Winkler *et al.* 1985].

DIP term order

If the ordering on the power products in distributed representation is choosen to be compatible with the ordering induced by recursive representation one arrives at the so called 'inverse lexicographical' term order [Gebauer, Kredel 1983].

$$e >_{L} 0 \Longleftrightarrow \begin{cases} \exists k \in \{1,\ldots,r\} \\ \text{with } e_j = 0 \text{ for } j = r, r-1,\ldots,k+1 \\ \text{and } e_k > 0 \end{cases}$$

From this we derive the 'inverse graduated' ordering:

$$e >_{G} 0 \Longleftrightarrow deg(e) > 0 \quad \text{or} \quad \begin{cases} deg(e) = 0 \\ \text{and} \\ e >_{L} 0 \end{cases}$$

Which is the same as:

$$e >_{G} 0 \Longleftrightarrow (e, deg(e)) >_{L} (0, deg(0))$$

The examples in [Böge *et al.* 1986] are based on these orderings. And they are also used in REDUCE 3.3 Gröbner basis package, and by Trinks, [Hearn 1987, Trinks 1978].

Scratchpad II term order

The 'newdistributed polynomial' representation in the Scratchpad II System uses the following 'lexicographical' termordering:

$$e >_{S} 0 \Longleftrightarrow \begin{cases} \exists k \in \{1,\ldots,r\} \text{ with} \\ e_j = 0 \text{ for } j = 1,\ldots,k-1 \\ \text{and } e_k > 0 \end{cases}$$

[Jenks *et al.* 1984]. This termordering is also used by Robbiano [Robbiano 1985]. With a 'graduation' it was also implemented by Schrader [Schrader 1976] in ALGOL 60.

$$e >_{SG} 0 \Longleftrightarrow deg(e) > 0 \quad \text{or} \quad \begin{cases} deg(e) = 0 \\ \text{and} \\ e >_{S} 0 \end{cases}$$

Which is the same as:

$$e >_{SG} 0 \Longleftrightarrow (deg(e), e) >_{S} (deg(0), 0)$$

Example

We include an example from Jenks [Jenks *et al.* 1984], to illustrate the different Gröbner bases with respect to the different term orderings. The computation was done using the SAC–2 computer algebra system on an IBM 3090–200 VF. Let $R = \mathbf{Q}(A, B)$ and let $S = R[X, Y, Z, T]$. Consider the following set of polynomials

```
( ( (A+1)/B X**2 Y - 1 ),
  ( (-B-1)/A X**2 Z + Y**3 ),
  ( - 12 X**2 T + Z**3 ) ).
```

The Gröbner bases with respect to different term orderings are:

inverse lexicographical term order

```
(  X**2 Y -  B /(  A +1 ) )
(  Z -(  A**2 + A )/(  B**2 + B ) Y**4 )
(  T -(  A**7 +4 A**6 +6 A**5 +4 A**4 + A**3 )/
      ( 12 B**7 +36 B**6 +36 B**5 +12 B**4 ) Y**13 )
executed in 750 milliseconds, 16058 cells used.
```

inverse graduated term order

```
(  X**2 Y -  B /(  A +1 ) )
(  X**2 Z -  A /(  B +1 ) Y**3 )
(  X**2 T - 1 / 12  Z**3 )
(  Y**4 -(  B**2 + B )/(  A**2 + A ) Z )
(  Y Z**3 - 12 B /(  A +1 ) T )
(  Y**3 T -(  B +1 )/ 12 A  Z**4 )
(  Z**7 - 144 A B /(  A B + B + A +1 ) Y**2 T**2 )
executed in 960 milliseconds, 17154 cells used.
```

Buchberger total degree term order

```
(  X**2 Y -  B /(  A +1 ) )
(  Y**3 -(  B +1 )/  A  X**2 Z )
(  Z**3 - 12  X**2 T )
(  X**4 Z -  A B /(  A B + B + A +1 ) Y**2 )
(  X**6 T -  A B /( 12 A B +12 B +12 A +12 ) Y**2 Z**2 )
executed in 530 milliseconds, 7965 cells used.
```

Note also that the computing times vary for different term orders. A rule of thumb how to obtain 'optimal' term orders will be discussed in a seperate section.

7.4.2 Description of term orders by linear forms

The explicit characterizations of admissible term orders in the previous section are easy to implement and run fast. However there exist much more admissible orders on the set of terms. In this section a uniform way to describe *all* admissible term orders will be discussed. Although the implementation is much more involved, the running time is still acceptable. The increase of computing time is 'only' 50 % in typical examples.

Observe that the admissible linear orders $<_{\mathbf{N}}$ on exponent vectors $e \in \mathbf{N}^n$ can be extended to admissible linear orders $<_{\mathbf{Q}}$ on $\mathbf{Q}^n$ such that $<_{\mathbf{N}} = <_{\mathbf{Q}} \cap \mathbf{N}^n$.

Let $S = \mathbf{R}[t]$ be the polynomial ring in one variable t over the real numbers $\mathbf{R}$. Define a linear order on S by

$$f > 0 \Longleftrightarrow \mathrm{ldcf}(f) >_{\mathbf{R}} 0 \quad \text{and} \quad f > g \Longleftrightarrow f - g >_{\mathbf{R}} 0,$$

for $f, g \in S$. For this order $t \in S$ is larger than all elements of $\mathbf{R}$, that is $t > a$ for all $a \in \mathbf{R}$.

Let $\mathcal{AL} = \{a = (a_1, \ldots, a_n) \in S^n\}$ such that the a_i, $i = 1, \ldots, n$ are strictly positive and linear independent over the rational numbers $\mathbf{Q}$ ($\subset \mathbf{R}$).

Then it has been shown by Robbiano and Weispfenning that any $a \in \mathcal{AL}$ induces an admissible linear order $<$ on $\mathbf{Q}^n$ and any addmissible linear order on $\mathbf{Q}^n$ is induced by such an $a \in \mathcal{AL}$ in the following way

$$x < y \Longleftrightarrow a \cdot x < a \cdot y,$$

where $x = (x_1, \ldots, x_n)$, $y = (y_1, \ldots, y_n) \in \mathbf{Q}^n$ and

$$a \cdot x = \sum_{i=1}^{n} a_i x_i \quad \in S.$$

[Robbiano 1985, Weispfenning 1987]

For the term orders $<_T$ defined in the previous section the corresponding $a_T \in \mathcal{AL}$ are as follows:

1. Buchberger's graduated (total degree) lexicographical term order $<_B$:

$$a_B = (t^n - t^{n-1}, t^n - t^{n-2}, \ldots, t^n - t^2, t^n - t, t^n - 1).$$

2. Inverse lexicographical term order $<_L$:

$$a_L = (1, t, t^2, \ldots, t^{n-2}, t^{n-1}).$$

3. Inverse graduated (total degree) term order $<_G$:

$$a_G = (1 + t^n, t + t^n, t^2 + t^n, \ldots, t^{n-2} + t^n, t^{n-1} + t^n).$$

4. Scratchpad II lexicographical term order $<_S$:

$$a_S = (t^{n-1}, t^{n-2}, \ldots, t^2, t, 1).$$

5. Scratchpad II graduated (total degree) lexicographical term order $<_{SG}$:

$$a_{SG} = (t^{n-1} + t^n, t^{n-2} + t^n, \ldots, t^2 + t^n, t + t^n, 1 + t^n).$$

6. Splitted inverse lexicographical term orders $<_T = (<_{T_1}, <_{T_2})$ on $T = T_1 * T_2$, defined as $u_1 u_2 \leq_T v_1 v_2 \Longleftrightarrow u_2 <_{T_2} v_2$ or ($u_2 = v_2$ and $u_1 \leq_{T_1} v_1$), where $u_1, v_1 \in T_1$ and $u_2, v_2 \in T_2$. If $a_{T_1} = (a_1, \ldots, a_i)$ and $a_{T_2} = (a_{i+1}, \ldots, a_n)$ for some $1 \leq i \leq n$, then

$$a_T = (a_1, \ldots, a_i, a_{i+1} * t^{n'}, \ldots, a_n * t^{n'}),$$

where $n' > \deg(a_j)$, $j = 1, \ldots, i$.

7. Varying term orders for $<_T$ with $a_T = (a_1, \ldots, a_n)$ and where $n' > \deg(a_j)$, $j = 1, \ldots, n$: for $i = 0, \ldots, n'$ let

$$a_{T_i} = (a_1 + b_i t^i, \ldots, a_n + b_i t^i),$$

where $b_i \in \mathbf{R}$ such that $a_{T_i} \in \mathcal{AL}$. Then we have $a_{T_0} = a_T$ (for $b_0 = 0$) and $a_{T_{n'}}$ a total degree term order (for $b_{n'} = 1$).

The currently implemented term orders can be checked with the procedure `EVOWRITE`. How these term orders can be specified in MAS is discussed in the next section.

7.4.3 Algorithms

The programs of the most important distributive polynomial algorithms and their complexity are summarized in the following. The main complexity numbers are as defined in the previous section. Only rational polynomial programs are discussed.

As the definition of the polynomial ring and the distributive representation suggests the algorithms will be constructed in the following way:

1. Loop on the monomials in the polynomials:
   (a) perform operations on the base coefficients,
   (b) perform operations on exponent vectors,

   Construct resulting polynomials.
2. Return the results.

The complexity of polynomial algorithms will therefore depend mainly on three factors:

1. the size of the base coefficients,
2. the number of terms,
3. the number of variables.

These quantities are as defined in the section on the recursive representation algorithms. The number of terms is now easily determined as half of the length of the polynomial in distributive representation.

We will continue to write $L(a)$ for $O(L(a))$, that means we will not count for constant factors. The computing time functions t, t^+, t^-, t^* are defined as before in section 5.7.

Let $\mathcal{A}$ be the set of atoms, $\mathcal{L}$ be the set of lists, $\mathcal{O} = \mathcal{A} \cup \mathcal{L}$ be the set of objects, $\mathcal{R} = \{x \in \mathcal{O} : \ x \text{ represents an element of } \mathbf{Q}\}$ be the set of rational numbers, $\mathcal{B} = \{x \in \mathcal{O} : \ x \text{ represents an element of a base coefficient ring } \}$ be the set of base coefficients, $\mathcal{D}_r = \{x \in \mathcal{O} : \ x \text{ represents a multivariate polynomial in } r \text{ variables } \}$ be the set of distributive polynomials, $\mathcal{DR}_r = \{x \in \mathcal{O} : \ x \text{ represents a multivariate polynomial over } \mathbf{Q} \text{ in } r \text{ variables } \}$ be the set of distributive rational polynomials.

We will first summarize selector functions which are independent of the base coefficient ring and then turn to the algorithms for rational polynomials.

$e \leftarrow DIPDEG(A)$ $A \in \mathcal{D}_r$, $e \in \mathcal{A}$. $e = \deg_r(A) = e_{kr}$ is the degree of A in the main variable. In distributive representation $t = 2$, $c = 0$.

$e \leftarrow DIPEVL(A)$ $A \in \mathcal{D}_r$, $e \in \mathcal{A}^r$. e is the exponent vector of the highest term. In distributive representation $t = 1$, $c = 0$.

$a \leftarrow DIPLBC(A)$ $A \in \mathcal{D}_r$, $a \in \mathcal{B}$. $a = a_k$ is the leading base coefficient. $t = 2$, $c = 0$.

$DIPMAD(A; a, e, A')$ $A, A' \in \mathcal{D}_r$, $e \in \mathcal{A}$, $a \in \mathcal{B}$. Monomial advance. $e = e_k$ is the exponent vector of the highest term. $a = a_k$ is the leading base coefficient. $t = 2$, $c = 0$.

$A' \leftarrow DIPMCP(a, e, A)$ $A, A' \in \mathcal{D}_r$, $e \in \mathcal{A}$, $a \in \mathcal{B}$. Monomial composition. e becomes the exponent vector of the highest term of A'. a becomes the leading base coefficient of A'. $t = 2$, $c = 2$.

$A \leftarrow DIPFMO(a, e)$ $A \in \mathcal{D}_r$, $e \in \mathcal{A}$, $a \in \mathcal{B}$. Polynomial from monomial. e becomes the exponent vector of the highest term of A. a becomes the leading base coefficient of A. $t = 2$, $c = 2$.

$DIPBSO(A)$ $A \in \mathcal{D}_r$. Polynomial (bubble) sort. The terms (which must be distinct) in A are sorted by the actual term order, the representing list is modified. There are other sorting algorithms which also allow for equal terms in A, but then coefficient arithmetic is required.

We turn now to the discussion of polynomial algorithms which depend on the base coefficient field. Only rational numbers as base coefficients will be treated.

$s \leftarrow DIRPSG(A)$ $A \in \mathcal{DR}_r$, $s \in \{-1, 0, +1\}$. s is the sign of the leading base coefficient of A. In our representation $t^+ = 2 + t^+_{RNSIGN} = 2 + L(A)$, $t^- = 2 + t^-_{RNSIGN} = 2 + 2 = 4$, $c = 0$.

$A' \leftarrow DIRPNG(A)$ $A, A' \in \mathcal{DR}_r$. $A' = -A$ is the negative of A. $t^+ = \mathrm{term}(A) \cdot t^+_{RNNEG} = kL(A)$, $c^+ = \mathrm{term}(A) \cdot c^+_{RNNEG} = kL(A)$. (In distributive representation no operations on exponent vectors are required.)

$A' \leftarrow DIRPAB(A)$ $A, A' \in \mathcal{DR}_r$. $A' = |A|$ is the absolute value of A, that is $\mathrm{sign}(A') \geq 0$. $t^+ = t^+_{DIRPNG} = kL(A)$, $c^+ = c^+_{DIRPNG} = kL(A)$; $t^- = 2 + t^-_{RNSIGN} = 2 + 1 = 3$, $c^- = 0$.

$C \leftarrow DIRPSM(A, B)$ $A, B, C \in \mathcal{DR}_r$. $C = A + B$ is the sum of A and B. Let $l = \max\{\mathrm{term}(A), \mathrm{term}(B)\}$. $t^+ = 2l \cdot t^+_{RNSUM} + t^+_{expon} = 2lL(A, B)^2 + 2lr$, $c^+ = 2l \cdot c^+_{RNSUM} + c^+_{expon} = 2lL(A, B)^2 + 2l$. t^+_{ecomp} denotes the time for comparing exponent vectors for term merge. If the set of terms of A and B are disjoint a pure merge is performed. In this case no operations on base coefficients are required and then $t^+ = 2lr$, $c^+ = 2l$. Although the exponent vectors are processed, they need not be reconstructed, thus $c^+ = 2l$ instead of $2lr$.

$C \leftarrow DIRPDF(A,B)$ $A,B,C \in \mathcal{DR}_r$. $C = A - B$ is the difference of A and B. The computing times are the same as for $DIRPSM$.

$C \leftarrow DIRPPR(A,B)$ $A,B,C \in \mathcal{DR}_r$. $C = A * B$ is the product of A and B. Let $l = \max\{\text{term}(A), \text{term}(B)\}$. $t^+ = l^2 \cdot t^+_{RNPROD} + t^+_{expon} = l^2 L(A,B)^2 + l^2 r$, $c^+ = l^2 \cdot c^+_{RNPROD} + c^+_{expon} = l^2 L(A,B)^2 + l^2 r$. The summations of intermediate polynomials are aranged to need only $l \log_2(l)$ summations of terms.

$B \leftarrow DIRPNF(P,A)$ $A,B \in \mathcal{DR}_r$, $P \in \mathcal{LDR}_r$. $B = \text{normal form}_P(A)$ is the normal form or completely reduced form of A with respect to P. Let $l = \text{term}(A)$. If A is irreducible then $t^- = l$, $c^- = l$. A one step reduction requires a monomial with polynomial product and one polynomial difference, thus $t^+_1 = 2lL(A,P)^2 + lr$, $c^+_1 = 2lL(A,P)^2 + lr$. The maximal computing time depends strongly on the used term order. As a worst case estimate consider $d = \max\{\deg(\text{HT}(p)) \mid p \in P\}$, $d' = \max\{\deg(p) \mid p \in P\}$ and let $d'' = \deg(A)$. Then B will contain at most d^r terms. In case of a total degree term order probably $m = d''^r - d^r$ terms will need reduction. So $t^+ = mt^+_1 = m2lL(A,P) \leq d''^r 2lL(A,P)$, $c^+ \leq d''^r 2lL(A,P)$. In case of a lexicographical termorder after each reduction step d'' may increase so termination can only be assured by Noetherian induction.

$Q \leftarrow DIRLIS(P)$ $P,Q \in \mathcal{LDR}_r$. Q is the irreducible set of P, that is every $p \in Q$ is irreducible with respect to $Q \setminus \{p\}$.

$G \leftarrow DIRPGB(P,t)$ $P,G \in \mathcal{LDR}_r$, $t \in \{0,1,2\}$. G is the Gröbner base of P. t is the 'trace flag', $t = 0$ if no intermediate output is required, $t = 1$ if the reduced S–polynomials are to be listed and $t = 2$ for maximal information.

This concludes the summary of distributive rational polynomial arithmetic functions. Finally we will summarize input / output functions and some often needed conversion functions.

$P \leftarrow PREAD()$ $P \in \mathcal{LDR}_r$. A list of distributive rational polynomials together with the variable list and a term order are read from the actual input stream. The accepted syntax (with start symbol `bunch`) is:

```
bunch    = varlist termord polylist
varlist  = "(" ident { "," ident } ")"
termord  = ( "L" | "G" | linform )
linform  = "(" univpoly { "," univpoly } ")"
polylist = "(" poly { "," poly } ")"
poly     = "(" term { ("+"|"-") term } ")"
term     = power { [ "*" ] power }
power    = factor [ "**" atom ]
factor   = ( rational | ident | "(" poly ")" )
```

With the context conditions:

1. the `atoms` must be nonnegative,
2. `idents` appearing in `poly` must be declared in `varlist`,
3. the right most variable in the variable list denotes the main variable,
4. `univpoly` must be an univariate `poly` in the variable `"T"` (=`ident`) over the rational numbers,
5. `linform` must be a valid linear form $\in \mathcal{AL}$,
6. the number of univariate polynomials must be equal to the number of variables in the variable list.

Examples will be discussed in the exercise section.

$PWRITE(P)$ — $P \in \mathcal{LDR}_r$. A list of distributive rational polynomials together with the variable list and a term order are written to the actual output stream. The output syntax of the polynomial list is equal to the input syntax of $PREAD$.

$B \leftarrow DIIFRP(A)$ — $A \in \mathcal{DR}_r$, $B \in \mathcal{D}_r$. $B = d \cdot A$ is converted to integral distributive representation, where d is the least common multiple of all denominators of base coefficients of A. Let $l = \mathrm{term}(A)$. $t^+ = l \cdot t^+_{ILCM} + l \cdot t^+_{RNPROD} = l2L(A)^2 + lL(A)^2 = 3lLA()^2$, $c^+ = l \cdot c^+_{ILCM} + l \cdot c^+_{RNPROD} = l2L(A)^2 + lL(A)^2 = 3lLA()^2$.

$B \leftarrow DIRFIP(A)$ — $A \in \mathcal{D}_r$, $B \in \mathcal{DR}_r$. $B = \frac{1}{d} \cdot A$ is converted to rational distributive representation, where $d = \mathrm{lbcf}(A)$. Let $l = \mathrm{term}(A)$. $t^+ = l \cdot t^+_{RNRED} = lL(A)^2$, $c^+ = l \cdot c^+_{RNRED} = lL(A)^2$.

$B \leftarrow DIPFP(r, A)$ — $A \in \mathcal{P}_r$, $B \in \mathcal{D}_r$. A in recursive representation is converted to B in distributive representation. Let $l = \mathrm{term}(A)$. $t^+ = 2lr$, $c^+ = 2lr$ since no operations on base coefficients are required.

$PFDIP(A; r, B)$ — $A \in \mathcal{D}_r$, $B \in \mathcal{P}_r$. A in distributed representation is converted to B in recursive representation. Let $l = \mathrm{term}(A)$. $t^+ = 2lr$, $c^+ = 2lr$ since no operations on base coefficients are required.

This concludes the summary of distributive polynomial functions.

For illustration we list the algorithms `DIRPNF` and `DIRLIS` in Modula–2 in MAS. The function of the algorithms should be clear from the step comments and the polynomial representation discussed before.

```
PROCEDURE DIRPNF(P,S: LIST): LIST;
(*Distributive rational polynomial normal form. P is a list
of non zero polynomials in distributive rational
representation in r variables. S is a distributive rational
polynomial. R is a polynomial such that S is reducible to R
modulo P and R is in normalform with respect to P. *)
VAR  AP, APP, BL, FL, PP, Q, QA, QE, QP, R, SL, SP, TA, TE: LIST;
BEGIN
(*1*) (*s=0. *)
```

```
      IF (S = 0) OR (P = SIL) THEN R:=S; RETURN(R); END;
(*2*) (*reduction step.*) R:=SIL; SP:=S;
      REPEAT DIPMAD(SP, TA,TE,SP);
             IF SP = SIL THEN SP:=0; END;
             PP:=P;
             REPEAT ADV(PP, Q,PP); DIPMAD(Q, QA,QE,QP);
                    SL:=EVMT(TE,QE);
                    UNTIL (PP = SIL) OR (SL = 1);
             IF SL = 0 THEN R:=DIPMCP(TE,TA,R);
                ELSE
                IF QP <> SIL THEN FL:=EVDIF(TE,QE);
                   BL:=RNQ(TA,QA);
                   AP:=DIPFMO(BL,FL); APP:=DIRPPR(QP,AP);
                   SP:=DIRPDF(SP,APP); END;
                END;
             UNTIL SP = 0;
(*3*) (*finish.*)
      IF R = SIL THEN R:=0; ELSE R:=INV(R); END;
(*6*) RETURN(R); END DIRPNF;
```

In the outer REPEAT–loop all terms of the polynomial S are considered for reduction. In the inner REPEAT–loop the head terms of polynomials in P are tested if they divide (EVMT exponent vector multiple test) the actual term of S. The IF–statement checks if a divisor was found, in which case the reduction is applied, or if the term was irreducible, in which case the term is copied to the result polynomial R.

```
PROCEDURE DIRLIS(P: LIST): LIST;
(*Distributive rational polynomial list irreducible set.
P is a list of distributive rational polynomials,
PP is the result of reducing each p element of P modulo P-(p)
until no further reductions are possible. *)
VAR  EL, FL, IRR, LL, PL, PP, PS, RL, RP, SL: LIST;
BEGIN
(*1*) (*initialise. *) PP:=P; PS:=SIL;
      WHILE PP <> SIL DO ADV(PP, PL,PP); PL:=DIRPMC(PL);
            IF PL <> 0 THEN PS:=COMP(PL,PS); END;
            END;
      RP:=PS; PP:=INV(PS); LL:=LENGTH(PP); IRR:=0;
      IF LL <= 1 THEN RETURN(PP); END;
(*2*) (*reduce until all polynomials are irreducible. *)
      LOOP ADV(PP, PL,PP); EL:=DIPEVL(PL); PL:=DIRPNF(PP,PL);
           IF PL = 0
              THEN LL:=LL-1;
                   IF LL <= 1 THEN EXIT END;
              ELSE FL:=DIPEVL(PL); SL:=EVSIGN(FL);
                   IF SL = 0 THEN PP:=LIST1(PL); EXIT END;
                   SL:=EQUAL(EL,FL);
                   IF SL = 1 THEN IRR:=IRR+1; ELSE IRR:=0;
                      PL:=DIRPMC(PL); END;
                   PS:=LIST1(PL); SRED(RP,PS); RP:=PS; END;
           IF IRR = LL THEN EXIT END;
           END;
(*3*) (*finish. *) RETURN(PP);
(*6*) END DIRLIS;
```

In the first step the polynomials in P are made monic (DIRPMC), that is their leading base coefficient is 1. In the main loop in step 2 all polynomials are reduced with respect to the rest of the polynomials. Then several case distinctions are made to check if the

polynomial reduced to 0 or if its head term EL, FL was reduced or not. The loop terminates if the number of head term irreducible polynomials (IRR) is equal to the total number of polynomials (LL). (The correctness proof is not totally trivial.)

7.4.4 Exercises

Let $S = \mathbf{Q}[X_1, X_2, X_3, X_4]$ and let $p_1 = X_1 + X_2 + X_3 + X_4 \in S$.

1. Read the polynomial p_1 in distributive rational representation (using the inverse lexicographical term order).
2. Compute $p_2 = p_1^3$ and print the result.
3. Convert p_1 to distributive integral representation and then to integral recursive representation q_1.
4. Compute $q_2 = q_1^3$. Compute the polynomial gcd q of q_1 and q_2. Is q equal to q_1 ?
5. Convert q_2 to distributive integral representation and then to distributive rational representation p_3. Is p_2 equal to p_3 ?
6. Use the following linear from $L = (1, t, t^2, t^3, t^4, t^5)$ to compute a Gröbner base of $(45P + 35S - 165B - 36, 35P + 40Z + 25T - 27S, 15W + 25SP + 30Z - 18T - 165B * *2, -9W + 15TP + 20SZ, PW + 2TZ - 11B * *3, 99W - 11BS + 3B * *2)$ with respect to this term order in $\mathbf{Q}[B, S, T, Z, P, W]$

Solution to step 1:
We use the function PREAD to input a list of polynomials, namely (p_1), in distributive rational representation. Then we take the first element of this list p_1. In addition we print the polynomial list. For the term order we use the inverse lexicographical term order L. The polynomial list can be printed by PWRITE. The variable list and the term order are remembered from the last call of PREAD.

```
P:=PREAD().
(x1,x2,x3,x4) L
(
( x1 + x2 + x3 + x4 )
)

PWRITE(P).
p1:=FIRST(P).
```

The result shows also the polynomials in distributive representation.

```
Enter polynomial list:
ANS: (((1 0 0 0) (1 1) (0 1 0 0) (1 1)
       (0 0 1 0) (1 1) (0 0 0 1) (1 1)))

Polynomial in the variables: (x1,x2,x3,x4)
Term ordering:  inverse lexicographical.
Polynomial list:
     (  x4 + x3 + x2 + x1 )

ANS: ((1 0 0 0) (1 1) (0 1 0 0) (1 1) (0 0 1 0) (1 1)
      (0 0 0 1) (1 1))
```

Solution to step 2:

The polynomial exponentiation function is called `DIRPEX`, `t` is the desired power. To print the result we reconstruct a polynomial list and use `PWRITE`.

```
t:=3.
p2:=DIRPEX(p1,t).
Q:=LIST(p2).
PWRITE(Q).
```

And this is the output of the second and fourth statement:

```
ANS: ((3 0 0 0) (1 1) (2 1 0 0) (3 1) (2 0 1 0) (3 1)
      (2 0 0 1) (3 1) (1 2 0 0) (3 1) (1 1 1 0) (6 1)
      (1 1 0 1) (6 1) (1 0 2 0) (3 1) (1 0 1 1) (6 1)
      (1 0 0 2) (3 1) (0 3 0 0) (1 1) (0 2 1 0) (3 1)
      (0 2 0 1) (3 1) (0 1 2 0) (3 1) (0 1 1 1) (6 1)
      (0 1 0 2) (3 1) (0 0 3 0) (1 1) (0 0 2 1) (3 1)
      (0 0 1 2) (3 1) (0 0 0 3) (1 1))

   Polynomial in the variables: (x1,x2,x3,x4)
   Term ordering:  inverse lexicographical.
   Polynomial list:
      (  x4**3 +3 x3 x4**2 +3 x2 x4**2 +3 x1 x4**2 +3 x3
      **2 x4 +6 x2 x3 x4 +6 x1 x3 x4 +3 x2**2 x4 +6 x1 x
      2 x4 +3 x1**2 x4 + x3**3 +3 x2 x3**2 +3 x1 x3**2 +
      3 x2**2 x3 +6 x1 x2 x3 +3 x1**2 x3 + x2**3 +3 x1 x
      2**2 +3 x1**2 x2 + x1**3 )
```

Solution to step 3:

`DIIFRP` converts a distributive rational polynomial into a distributive integral polynomial. `PFDIP` converts then to a recursive integral polynomial. Since `PFDIP` is no function, we must use the sequence `r:=r. q1:=q1.` to display the results.

```
q1:=DIIFRP(p1).
PFDIP(q1,r,q1). r:=r. q1:=q1.
```

The resulting output is:

```
ANS: ((1 0 0 0) 1 (0 1 0 0) 1 (0 0 1 0) 1 (0 0 0 1) 1)
ANS: 4
ANS: (1 (0 (0 (0 1))) 0 (1 (0 (0 1)) 0 (1 (0 1) 0 (1 1))))
```

Solution to step 4:

`IPEXP` denotes the integral polynomial exponentiation function. `r` is the number of variables as determined by `PFDIP`. The polynomial gcd function is called `IPGCDC`. It computes the $\gcd(q_1, q_2) = q$ and the cofactors of the gcd (that is $q_1 = qy$ and $q_2 = qz$). To display the results we need the sequence `q:=q. y:=y. z:=z.`

```
q2:=IPEXP(r,q1,t).
IPGCDC(r,q1,q2,q,y,z).
q:=q. y:=y. z:=z.
```

The output shows that $\gcd(q_1, q_2) = q_1$ as expected. The cofactors are $z = p_1^2$ and $y = 1$.

```
ANS: (3 (0 (0 (0 1))) 2 (1 (0 (0 3)) 0 (1 (0 3)
      0 (1 3))) 1 (2 (0 (0 3)) 1 (1 (0 6) 0 (1 6))
      0 (2 (0 3) 1 (1 6) 0 (2 3))) 0 (3 (0 (0 1))
      2 (1 (0 3) 0 (1 3)) 1 (2 (0 3) 1 (1 6) 0 (2 3))
      0 (3 (0 1) 2 (1 3) 1 (2 3) 0 (3 1))))

ANS: (1 (0 (0 (0 1))) 0 (1 (0 (0 1)) 0 (1 (0 1) 0 (1 1))))
ANS: (0 (0 (0 (0 1))))
ANS: (2 (0 (0 (0 1))) 1 (1 (0 (0 2)) 0 (1 (0 2)
      0 (1 2))) 0 (2 (0 (0 1)) 1 (1 (0 2) 0 (1 2)) 0
      (2 (0 1) 1 (1 2) 0 (2 1))))
```

Solution to step 5:

To suppress unwanted output we use a BEGIN–block. `DIFIP` converts a recursive polynomial to a distributive polynomial and `DIRFIP` converts to distributive rational representation. The resulting polynomial is put into a list and writen by `PWRITE`. Finally we test the equality of p_2 and q_3 with the function `EQUAL`.

```
BEGIN p3:=DIPFP(r,q2);
      p3:=DIRFIP(p3);
      Q:=LIST(p3);
      PWRITE(Q);
      IF EQUAL(p2,p3) = 1 THEN CLOUT("equal")
                          ELSE CLOUT("not equal") END;
      END.
```

The resulting output is as follows:

```
Polynomial in the variables: (x1,x2,x3,x4)
Term ordering:  inverse lexicographical.
Polynomial list:
        (  x4**3 +3 x3 x4**2 +3 x2 x4**2 +3 x1 x4**2 +3 x3
        **2 x4 +6 x2 x3 x4 +6 x1 x3 x4 +3 x2**2 x4 +6 x1 x
        2 x4 +3 x1**2 x4 + x3**3 +3 x2 x3**2 +3 x1 x3**2 +
        3 x2**2 x3 +6 x1 x2 x3 +3 x1**2 x3 + x2**3 +3 x1 x
        2**2 +3 x1**2 x2 + x1**3 )
equal
```

Solution to step 6:

We use the function `PREAD` to input a list of polynomials together with the linear form in distributive rational representation. The name `T` for the variable in the linear form is fixed and cannot changed. With the given linear form, the variable `B` becomes the lowest variable.

```
P:=PREAD().

(B,S,T,Z,P,W) ( 1, T, T**2, T**3, T**4, T**5 )
(
( 45 P + 35 S - 165 B - 36 ),
( 35 P + 40 Z + 25 T - 27 S ),
( 15 W + 25 S P + 30 Z - 18 T - 165 B**2 ),
( - 9 W + 15 T P + 20 S Z ),
( P W + 2 T Z - 11 B**3 ),
( 99 W - 11 B S + 3 B**2 )
( B**2 + 33/50 B + 2673/10000 )
)
```

```
PWRITE(P).
Q:=DIRPGB(P,1).
PWRITE(Q).
```

The resulting output is as follows:

```
LFCHECK: LF linearly independent.
Polynomial in the variables: (B,S,T,Z,P,W)
Term ordering:
    1
     T
     T**2
     T**3
     T**4
     T**5

Polynomial list:
   ( 45 P +35 S -165 B -36 )
   ( 35 P +40 Z +25 T -27 S )
   ( 15 W +25 S P +30 Z -18 T -165 B**2 )
   ( -9 W +15 T P +20 S Z )
   (  P W +2 T Z -11 B**3 )
   ( 99 W -11 B S +3 B**2 )
```

Note, that the linear form is checked if it is really linearly independent over the rational numbers to avoid mistakes.

The Gröbner base is as follows:

```
Polynomial in the variables: (B,S,T,Z,P,W)
Term ordering:
    1
     T
     T**2
     T**3
     T**4
     T**5

Polynomial list:
   (  B**2 +33/50 B +2673/10000 )
   (  S -5/2 B -9/200 )
   (  T -37/15 B +27/250 )
   (  Z +49/36 B +1143/2000 )
   (  P -31/18 B -153/200 )
   (  W +19/120 B +1323/20000 )
```

This concludes the discussion of the exercises.

7.5 Interface to the MAS language

In section 7.4.3 on algorithms for distributive polynomials we introduced the procedures *PREAD* and *PWRITE*. These procedures perform conversions between strings (input / output streams) and (lists) of distributive rational polynomials.

In this section we will discuss mainly two other conversion procedures:

- $POLY$ converts a list of MAS expressions to a list of distributive rational polynomials, if the MAS expressions have a meaning as polynomials.
- $TERM$ converts a list of distributive rational polynomials to a (quoted) MAS expression, if the polynomials have a meaning as MAS expression.

These two procedures improve the interactive usage of the distributive polynomial system. As a distinguishing feature the user of $POLY$ need no more worry about specifying a variable list, since all variables in the expressions are automaticaly added to the variable list, if not allready there. Acompanying procedures are available which set the desired term order and the variable list.

Recall some definitions: Let $\mathcal{L}$ be the set of lists, $\mathcal{O}$ be the set of objects, $\mathcal{S} = \{x \in \mathcal{O} : \ x$ represents a S–expression $\}$ be the set of S–expressions (they are the output of the MAS parser), $\mathcal{DR}_r = \{x \in \mathcal{O} : \ x$ represents a multivariate polynomial over $\mathbf{Q}$ in r variables $\}$ be the set of distributive rational polynomials.

The interface functions are defined as follows:

$P \leftarrow POLY(p_1, \ldots, p_n)$ $p_i \in \mathcal{S}$ for $i = 1, \ldots, n$ and $P \in \mathcal{LDR}_r$. The MAS expressions p_i are converted to distributive rational polynomials. All variables occuring in the p_i's, which are not in the system variable list are added to it as small variables. The actual defined term order is used. The syntax definition of $POLY$ is:

```
ident ":=" "POLY(" expr { "," expr } ")"
```

Where `expr` is an ordinary MAS expression which is interpretable as a polynomial over the rational numbers. That is the following context conditions apply:

1. Only the operators `"+"`, `"-"`, `"*"`, `"**"`, `"^"` and `"/"` may appear.
2. The division operator `"/"` may only by used between numbers (atoms).
3. String constants may occur and (the string contents) are interpreted as rational numbers.
4. The identifiers occuring in `expr` are per definition disjoint to identifiers used as MAS variables even if they have the same name.
5. No function names may apear in the MAS expressions.

$POLY$ is implemented by $DIP2SYM$.

$T \leftarrow TERM(P)$ $P \in \mathcal{LDR}_r$ and $T \in \mathcal{S}$. P is a list of distributive rational polynomials. T is a quoted list of S–expressions, that is a list of terms, which are marked that they are not to be evaluated by the MAS interpreter. For the variable names the names from the actual variable list are used. $TERM$ is implemented by $SYM2DIP$.

$V' \leftarrow DIPVDEF(V)$ $V, V' \in \mathcal{L}$. V and V' are variable lists. The list V is converted to a variable list and stored in the global variable $VALIS$. The new variable list is returned in V'.

$t' \leftarrow DIPTODEF(t)$ $t, t' \in \mathcal{L}$. t and t' are term order indicators as defined in the global variable $EVORD$. t becomes the new term order, the old term order t' is returned. Possible values for t are:

1 invers lexicographical term order ascending order,

2 invers lexicographical term order descending order, 2 corresponds to the term order letter `"L"` in $PREAD$,

3 invers graded lexicographical term order ascending order,

4 invers graded lexicographical term order descending order, 4 corresponds to the term order letter `"G"` in $PREAD$,

5 lexicographical term order ascending order,

6 lexicographical term order descending order,

7 total degree Buchberger lexicographical term order ascending order,

8 total degree Buchberger lexicographical term order descending order,

L a linear form, L is a list $L = (l_1, \ldots, l_n)$ where each $l_i \in \mathcal{IP}_1$, $i = 1, \ldots, n$ is an *univariate recursive integral* polynomial.

Note: The orders with even indicator numbers are admissible term orders, the others are in general not admissible.

$EVOWRITE(t)$ $t \in \mathcal{L}$. t is a term order indicator, the meaning of this indicator as term order is written to the output stream.

Since the argument expressions to $POLY$ are not evaluated, there is no way substitute a value for a variable in such an expression. (Substitution in polynomials can only performed by the library functions like $DIRPSM$ or $DIRPSV$.) This is different to other computer algebra systems like REDUCE where the polynomial variables are identified with the programming language variables.

The reason for this behavior lies in the MAS view of polynomials: they are constants from some algebraic structure and do not belong to the programming (meta) language.

Examples:

Let $S = \mathbf{Q}[X_1, X_2, X_3, X_4]$ and let $p, q \in S$, $p = X_1 + X_2 + X_3 + X_4$, $q = 3/4X_1^2$. We will first discuss some examples for the use of $POLY$, then of $TERM$ and finaly some examples for $DIPVDEF$ and $DIPTODEF$.

```
P:=POLY(x1+x2+x3+x4).
Variable(s) added to VALIS:  x4, x3, x2, x1
ANS: (((1 0 0 0) (1 1) (0 1 0 0) (1 1)
       (0 0 1 0) (1 1) (0 0 0 1) (1 1)))

p:=FIRST(POLY( x1 + x2 + x3 + x4 )).
```

```
ANS: ((1 0 0 0) (1 1) (0 1 0 0) (1 1)
      (0 0 1 0) (1 1) (0 0 0 1) (1 1))

q:=SECOND(POLY( x1 + x2 + x3 + x4, 3/4 x1^2 )).
ANS: ((2 0 0 0) (3 4))
```

In the first example the variables `x1, x2, x3, x4` have not been in the variable list. A message is printed that they are added to the variable list. Note that the order of the (self detected) variables can not be precisely determined, since it depends on the expression tree structure of the parsed MAS expression. If the order is important use *DIPVDEF* to set the variables explicitly. Observe that *POLY* allways produces a list of distributive polynomials. So in order to obtain one polynomial any list selector function can be used. The operands of `"/"` must be atoms, so `3/4` is ok.

```
P:=POLY( "11111/33" ).
ANS: (((0 0 0 0) (11111 33)))
```

String constants can be used in *POLY*. The string contents are interpreted as rational numbers (read by *RNDRD*).

```
P:=POLY( (x1+x2)^2 ).
ANS: (((2 0 0 0) (1 1) (1 1 0 0) (2 1)
       (0 2 0 0) (1 1)))
```

During the conversion from MAS expressions the distributive polynomials are expanded (or distributed as their name suggests).

In the opposite direction we will convert `P` back to a MAS expression.

```
T:=TERM(P).
ANS: QUOTE(((x1^2+2*x1*x2)+x2^2))
```

Observe that the MAS expressions have a tree structure, which is visible by the (useless) inner parenthesis around the first two summands.

```
T:=TERM( POLY( 3/4 ) ).
ANS: QUOTE((3/4))
```

This shows, that atoms are printed correctly, however integers are not printed correctly:

```
T:=TERM( POLY( "11111111111111111111111111111111/2" ) ).
ANS: QUOTE(((197423559 458992985 3)/2))
```

Lists of atoms are not printed as integers but as lists of atoms.

Some further examples for the use of *DIPVDEF* and *DIPTODEF*.

```
V:=DIPVDEF(LIST("x4", "x9", "x12", "x3")).
ANS: ((62 4) (62 9) (62 12) (62 3))
```

Polynomial variable names can be specified as character strings. `"x3"` now becomes the new main variable. The new variable list is returned. To extend (or modify) an existing variable list any list processing function can be used.

```
V:=CCONC(V,LIST("x5", "x6", "x7", "x8")).
ANS: ((62 4) (62 9) (62 1 2) (62 3)
      (62 5) (62 6) (62 7) (62 8))
DIPVDEF(V).
ANS: ((62 4) (62 9) (62 1 2) (62 3)
      (62 5) (62 6) (62 7) (62 8))
```

A final example is concerned with selection of term orders.

```
DIPTODEF(6).
ANS: 2
```

This call of $DIPTODEF$ switches to lexicographical term order. The old term order was 2 = inverse lexicographical. Setting a linear form defined term order is a bit more involved since there is at the moment no function wich converts a list of rational distributive polynomials into a list of integral recursive polynomials. But this will soon be available.

This concludes the overview of the MAS language to distributive polynomial interface.

7.6 Optimization of the Term Order

In most applications the computating time for a Gröbner Basis is strongly dependend on the chosen variable ordering and term ordering. (See the example below, table 7.6.) To find an 'optimal' variable ordering one looks at the reduced univariate polynomials:

Definition: Let $f(x_1, \ldots, x_r) \in \mathrm{R}$, then the *reduced univariate polynomial* corresponding to f for the variable $x_i (1 \leq i \leq r)$ is defined by:

$$p_i(x_i) = g(1, ..., 1, x_i, 1, ..., 1) \in \mathbf{N}[x_i]$$

with $g = \sum x^{(i)}$ when $f = \sum a_{(i)} x^{(i)}$. The reduced polynomial for a set of polynomials is the sum over all reduced polynomials corresponding to the elements of the set.

Experience: Tests for computing times of Gröbner bases and factorizing of multivariate polynomials have shown: The variable ordering is optimal if

$$p_1(x) \geq \ldots \geq p_r(x).$$

where the univariate polynomials are ordered according to the inverse lexicographical ordering of their coefficient vectors, that means:

$$h(x) > 0 \Longleftrightarrow ldcf(h) > 0 \quad \text{and} \quad h(x) > k(x) \Longleftrightarrow h(x) - k(x) > 0$$

The computation of the reduced univariate polynomials itself and the reordering of the variables is not much time consuming. So this optimization will in many cases lead to faster Gröbner bases computation.

There is an algorithm $DIPVOP$ in the distributive polynomial system which does this kind of term order optimization. Its specification is

$DIPVOP(P, V; P', V')$ with $P, P' \in \mathcal{D}_r$ and V, V' are variable lists. The representation of the polynomials in P and the variable list V are changed according to the optimization heuristics. P' is the new polynomial list and V' is the new variable list.

variable ordering	time term order = L	time term order = G
BSTZPW	1.95	10.92
SBTZPW	27.99	16.37
STBZPW	110.27	18.74
STZPBW	115.35	16.37
STZPWB	247.03	20.81
SZPWBT	67.68	19.78
PWBTSZ	103.16	20.61
ZWBSTP	> 3 600.00	32.31
TZPWBS	tfc	50.50
ZPWBST	> 3 600.00	39.32
PWBSTZ	37.18	20.88
WBSTZP	34.98	10.81

Computing time in seconds on IBM 370/168. 'tfc' = 'to few cells reclaimed', that means not enough storage was available. From [Böge *et al.* 1986].

Table 7.1: Computing times for different term orderings

$M \leftarrow DIPDEM(p)$	for one polynomial $p \in \mathcal{D}_r$, and
$M \leftarrow DIPLDM(P)$	for a list of polynomials $P \in \mathcal{LD}_r$ determine the tuples of reduced univariate polynomials $M \in \mathcal{LIP}_1$.

Example:

To demonstrate the influence of the term order on the computation of Gröbner bases we include the following example of an ideal generated by 7 polynomials in 6 variables [Trinks 1978]. Let $R = \mathbf{Q}$, $S = R[B, S, T, Z, P, W]$ and let the polynomials be

```
( + 45 P + 35S -165B -36 ,
  + 35 P +40 Z + 25 T -27 S ,
  + 15 W + 25 P S + 30 Z - 18 T - 165 B**2 ,
  - 9 W + 15 P T + 20 Z S ,
    W P + 2 Z T - 11 B**3 ,
  99 W - 11 S B + 3 B**2 ,
  B**2 + 33/50 B + 2673/10000   )
```

In table 7.6 the dependence is clearly visible. The reduced univariate polynomial for variable `B` has degree 3, so a term order where `B` is smaller than all other variables will be most desirable. Observe that as more right `B` stands in the variable list as worst become the computing times.

Note that especially the inverse lexicograpbical term order is very sensitive against reordering of variables, the inverse graduated term order is less sensitive. However in some applications (like ideal elimination theory) Gröbner bases are required with respect to the inverse lexicographical term order .

7.7 Non-commutative Solvable Polynomial System

A solvable polynomial ring is an ordinary commutative polynomial ring $R = \mathbf{K}[X_1, \ldots, X_n]$ equipped with a new non-commutative multiplication $*$. The field $\mathbf{K}$ is assumed to be commutative and to commute with the indeterminates $X_1, \ldots, X_n$. The set T of terms (power-products of indeterminates) is supposed to be linearly ordered by an admissible order $<_T$ such that the order is compatible with the new multiplication $*$. The axioms of solvable polynomial rings are discussed in section 8.6 and where first defined by [Kandri-Rody, Weispfenning 1988]. In the implementation we use the ordinary commutative distributive polynomial representation. The non-commutative product $*$ is defined via relations, which are elements of a free associative algebra. These relations are represented as triples

$$(u, v, p)$$

of commutative terms $u, v \in T$ and a commutative polynomial $p \in R$, such that

$$u * v = p$$

and p is of the form

$$c \cdot u \cdot v + p'$$

with $0 \neq c \in \mathbf{K}$ and $p' \in R$, where $p' <_T u \cdot v$. A table (implemented as list) of these relations is required as input parameter for most algorithms for solvable polynomials.

7.7.1 Algorithms

The programs of the most important non-commutative solvable type distributive polynomial algorithms are summarized in the following. Only rational polynomial programs are discussed.

As the definition of the solvable polynomial ring and the distributive representation suggests the algorithms will be constructed in the following way:

1. Loop on the monomials in the polynomials:
   (a) perform operations on exponent vectors by recursive application of the non-commutative product algorithm, using and updating the relation table,
   (b) perform operations on the base coefficients of the term and the non-commutative product polynomial,

   Construct resulting polynomials.
2. Return the results.

The complexity of polynomial algorithms will therefore depend mainly on five factors:

1. the size of the base coefficients of the polynomials,
2. the size of the base coefficients of the commutator relations,
3. the number of terms of the polynomials,
4. the degree and the number of terms of the commutator relations,

5. the number of variables.

These quantities are as defined in the section on the recursive representation algorithms. We will continue to write $L(a)$ for $O(L(a))$, that means we will not count for constant factors. The computing time functions t, t^+, t^-, t^* are defined as before in section 5.7.

Let $\mathcal{A}$ be the set of atoms, $\mathcal{A}^+$ be the set of non-negative atoms, $\mathcal{L}$ be the set of lists, $\mathcal{O} = \mathcal{A} \cup \mathcal{L}$ be the set of objects, $R = \{x \in \mathcal{O} : \ x$ represents an element of $\mathbf{Q}\}$ be the set of rational numbers, $\mathcal{B} = \{x \in \mathcal{O} : \ x$ represents an element of a base coefficient ring $\}$ be the set of base coefficients, $\mathcal{D}_r = \{x \in \mathcal{O} : \ x$ represents a multivariate polynomial in r variables $\}$ be the set of distributive polynomials, $\mathcal{DR}_r = \{x \in \mathcal{O} : \ x$ represents a multivariate polynomial over $\mathbf{Q}$ in r variables $\}$ be the set of distributive rational polynomials. $\mathcal{CR}_r = \{x \in \mathcal{O} : \ x$ represents triples of multivariate polynomials over $\mathbf{Q}$ in r variables $\}$ be the set of commutator relations of distributive rational polynomials.

The programs are summarized in the following table

$C \leftarrow DINPPR(T, A, B)$ $T \in \mathcal{CR}_r$, $A, B, C \in \mathcal{DR}_r$. $C = A * B$ is the non-commutative product of A and B in a solvable polynomial ring with commutator relations T.

$C \leftarrow DINCCO(T, A, B)$ $T \in \mathcal{CR}_r$, $A, B, C \in \mathcal{DR}_r$. $C = A*B-B*A$ is the commutator of A and B in a solvable polynomial ring with commutator relations T.

$B \leftarrow DINPEX(T, A, n)$ $T \in \mathcal{CR}_r$, $A, B \in \mathcal{DR}_r$, $n \in \mathcal{A}$. $B = A^n$ is the non-commutative n-th power of A in a solvable polynomial ring with commutator relations T.

$P \leftarrow NPREAD(T)$ $T \in \mathcal{CR}_r$, $P \in \mathcal{LDR}_r$. A list of distributive rational polynomials is read from the actual input stream. All multiplications of variables mean the non-commutative $*$-product, even if $*$ is not explicitly written. The accepted syntax (with start symbol `polylist`) is:

```
polylist = "(" [ poly { "," poly } ] ")"
poly     = "(" term { ("+"|"-") term } ")"
term     = power { [ "*" ] power }
power    = factor [ "**" atom ]
factor   = ( rational | ident | "(" poly ")" )
```

With the context conditions:

1. the '`atoms`' must be nonnegative,
2. '`ident`'s which appear in '`poly`' must be declared in the item '`varlist`' of $PREAD$ during the input of T,

Examples will be discussed in the exercise section.

$B \leftarrow DINLNF(T, P, A)$ $T \in \mathcal{CR}_r$, $A, B \in \mathcal{DR}_r$, $P \in \mathcal{LDR}_r$. B is the left normal form of A with respect to P in a solvable polynomial ring with commutator relations T: $B = \text{left} - \text{normalform}_P(A)$.

$Q \leftarrow LIRRSET(T, P)$ and

$Q \leftarrow DINLIS(T, P)$ $T \in \mathcal{CR}_r$, $P, Q \in \mathcal{LDR}_r$. Q is the left irreducible set of P in a solvable polynomial ring with commutator relations T. That means every $p \in Q$ is monic and irreducible with respect to $Q \setminus \{p\}$.

$G \leftarrow LGBASE(T, P, t)$ and

$G \leftarrow DINLGB(T, P, t)$ $T \in \mathcal{CR}_r$, $P, G \in \mathcal{LDR}_r$, $t \in \{0, 1, 2\}$. G is the left Gröbner base of P in a solvable polynomial ring with commutator relations T. t is the 'trace flag', $t = 0$ if no intermediate output is requred, $t = 1$ if the reduced S–polynomials are to be listed and $t = 2$ for maximal information.

$G \leftarrow TSGBASE(T, P, t)$ and

$G \leftarrow DINCGB(T, P, t)$ $T \in \mathcal{CR}_r$, $P, G \in \mathcal{LDR}_r$, $t \in \{0, 1, 2\}$. G is the two-sided Gröbner base of P in a solvable polynomial ring with commutator relations T. t is the 'trace flag', $t = 0$ if no intermediate output is requred, $t = 1$ if the reduced S–polynomials are to be listed and $t = 2$ for maximal information.

$P \leftarrow CenterPol(T, E)$ and

$P \leftarrow DINCCPpre(T, E)$ $T \in \mathcal{CR}_r$, $E \in \mathcal{L}(\mathcal{A}^{+r})$, $P \in \mathcal{LDR}_r$. P is a list of polynomials in r variables, where the terms are contained in E ($T(p) \subseteq E$, $p \in P$) and each $p \in P$ is in the center of a solvable polynomial ring with commutator relations T. Moreover the polynomials in P are written to the actual output stream.

$p \leftarrow DINCCP(T, E)$ $T \in \mathcal{CR}_r$, $E \in \mathcal{L}(\mathcal{A}^{+r})$, $p \in \mathcal{LDR}_{r+s}$. p is a polynomial in $r + s$ variables, $0 \leq s \leq |E|$, where the s 'new' variables indicate parametric coefficients, and the 'old' terms are contained in E ($T(p) \subseteq E$). For any specialization of the parameters to $\mathbf{Q}$, the specialized polynomial p is in the center of a solvable polynomial ring with commutator relations T.

$E \leftarrow EVLGTD(r, d, L)$ $r, d \in \mathcal{A}^+$, $L \in \mathcal{L}(\mathcal{A}^{+r})$, $E \in \mathcal{LL}(\mathcal{A}^{+r})$. r is the number of variables, d is the total degree and L is a list of already computed terms (used for internal recursion, and initially called with L empty). It returns a list $E = (E_0, \ldots, E_d)$ where each E_i, $1 \leq i \leq d$, is a list of exponents of terms in r variables of total degree exactly i.

$E \leftarrow EVLGIL(D)$ $D \in \mathcal{A}^{+r}$, $E \in \mathcal{L}(\mathcal{A}^{+r})$. $D = (d_1, \ldots, d_r)$ is a list of non-negative atoms and E is a list of of terms, such that for any $(e_1, \ldots, e_r) \in E$, $0 \leq e_i \leq d_i$ holds for $1 \leq i \leq r$.

For examples see section 8.6.2.

7.8 Arbitrary domain system

The arbitrary domain system consists of two parts:

1. the distributive polynomial system algorithms over arbitrary domain coefficients
2. and the arbitrary domain algorithms themselves.

Distributive polynomials with arbitrary domain coefficients are represented in the same way as distributive polynomials, except that the representation of the base coefficients is different. So all distributive polynomial algorithms which do not depend on the base coefficient ring can be applied to distributive polynomials with arbitrary domain coefficients.

Let $\mathcal{AD}$ denote the set of arbitrary domain elements and let $\mathcal{AD}_d$ denote the set of arbitrary domain elements from domain d. Then the representation of an arbitrary domain element $a \in \mathcal{AD}_d$ is the list

$$(n_d, a', \ldots).$$

Where $n_d \in \mathcal{A}$ is a unique atom which identifies the domain d, a' is the representation of a domain element (e.g. a rational number or an integer) and the rest of the list $(\ldots)$ represents internal information of the domain. The value n_d is uniquely assigned to a domain during programm initialization and may thus vary from one MAS run to another. The domain descriptor read functions associate the domain number with the external domain specification. The actually available domains depend on the totally available domains and the compiled version of MAS.

The available domains are discussed later together with examples for each domain. We turn now to the description of the algorithms.

7.8.1 Algorithms

We first summarize the programs of the most important distributive arbitrary domain polynomial algorithms and then we summarize the programs of the most important arbitrary domain algorithms.

Let $\mathcal{A}$ be the set of atoms, $\mathcal{L}$ be the set of lists, $\mathcal{O} = \mathcal{A} \cup \mathcal{L}$ be the set of objects, $\mathcal{AD} = \{x \in \mathcal{O} : \ x$ represents an element of an arbitrary domain base coefficient ring $\}$ be the set of arbitrary domain base coefficients, $\mathcal{DAD}_r = \{x \in \mathcal{O} : \ x$ represents a multivariate polynomial in r variables over arbitrary domain coefficients $\}$ be the set of distributive arbitrary domain polynomials. $\mathcal{D}_r = \{x \in \mathcal{O} : \ x$ represents a multivariate polynomial in r variables $\}$ be the set of distributive polynomials.

We discuss now the polynomial algorithms which depend on the base coefficient field. The polynomial algorithms which do not depend on the base coefficient ring are the same as discussed in the section on distributive polynomials.

$A' \leftarrow DIPNEG(A)$ $\quad A, A' \in \mathcal{DAD}_r$. $A' = -A$ is the negative of A.

$C \leftarrow DIPSUM(A, B)$ $\quad A, B, C \in \mathcal{DAD}_r$. $C = A + B$ is the sum of A and B.

$C \leftarrow DIPDIF(A, B)$ $\quad A, B, C \in \mathcal{DAD}_r$. $C = A - B$ is the difference of A and B.

$C \leftarrow DIPROD(A, B)$ $\quad A, B, C \in \mathcal{DAD}_r$. $C = A * B$ is the product of A and B.

$B \leftarrow DIPNOR(P, A)$ $A, B \in \mathcal{DAD}_r$, $P \in \mathcal{LDAD}_r$. $B = \text{normalform}_P(A)$ is the normal form or completely reduced form of A with respect to P. $\mathcal{AD}$ must be a field.

$Q \leftarrow DILIS(P)$ $P, Q \in \mathcal{LDAD}_r$. Q is the irreducible set of P, that is every $p \in Q$ is irreducible with respect to $Q \setminus \{p\}$. $\mathcal{AD}$ must be a field.

$G \leftarrow DIPGB(P, t)$ $P, G \in \mathcal{LDAD}_r$, $t \in \{0, 1, 2\}$. G is the Gröbner base of P. t is the 'trace flag', $t = 0$ if no intermediate output is requred, $t = 1$ if the reduced S–polynomials are to be listed and $t = 2$ for maximal information. $\mathcal{AD}$ must be a field.

$B \leftarrow DIIFNF(P, A', A)$ $A, A', B \in \mathcal{DAD}_r$, $P \in \mathcal{LDAD}_r$. $B = \text{normalform}_P(A)$ is the normal form or completely reduced form of A with respect to P. A' is a polynomial which has already been reduced. $\mathcal{AD}$ must be the ring of integers or integral functions and the polynomial A' and the polynomial A are multiplied by the head terms of the polynomials $p \in P$ which are used for reduction.

$Q \leftarrow DIIFLS(P)$ $P, Q \in \mathcal{LDAD}_r$. Q is the irreducible set of P, that is every $p \in Q$ is irreducible with respect to $Q \setminus \{p\}$. $\mathcal{AD}$ must be the ring of integers or integral functions.

$G \leftarrow DIIFGB(P, t)$ $P, G \in \mathcal{LDAD}_r$, $t \in \{0, 1, 2\}$. G is the Gröbner base of P. t is the 'trace flag', $t = 0$ if no intermediate output is requred, $t = 1$ if the reduced S–polynomials are to be listed and $t = 2$ for maximal information. $\mathcal{AD}$ must be the ring of integers or integral functions.

$B \leftarrow DIFIP(A, d)$ $A \in \mathcal{DI}_r$, $B \in \mathcal{DAD}_r$, $d \in \mathcal{AD}_d$. The distributive integral polynomial is converted to a distributive arbitrary domain polynomial with specified domain d.

$P \leftarrow DILRD(V, d)$ $P \in \mathcal{LDAD}_r$. A list of distributive rational polynomials with the variable list V, $d \in \mathcal{AD}_d$ and term order as defined in `EVORD` are read from the actual input stream. The accepted syntax (with start symbol `polylist`) is:

```
polylist = "(" poly { "," poly } ")"
poly     = "(" term { ("+"|"-") term } ")"
term     = power { [ "*" ] power }
power    = factor [ "**" atom ]
factor   = ( ident | "(" domain element ")" )
```

With the context conditions:

1. the `atoms` must be nonnegative,
2. `idents` appearing in `poly` must be declared in `varlist`,
3. the right most variable in the variable list denotes the main variable,

4. `domain element` must be a valid element of the domain as defined by d.

Examples will be discussed later.

$DILWR(P,V)$ — $P \in \mathcal{LDAD}_r$. A list of distributive rational polynomials is written to the actual output stream using the variable list V. The output syntax of the polynomial list is similar to the input syntax of $DILRD$.

This concludes the summary of the distributive arbitrary domain polynomial functions.

We turn now to the summary of the programs of the most important arbitrary domain algorithms. As before let $\mathcal{AD} = \{x \in \mathcal{O} : \ x$ represents an element of an arbitrary domain base coefficient ring $\}$ be the set of arbitrary domain base coefficients.

$s \leftarrow ADONE(A)$ — $A \in \mathcal{AD}$, $s \in \mathcal{A}$. $s = 1$ if $A = 1_{\mathcal{AD}}$ otherwise $s \neq 1$. If the domain has no one-test function defined an error occurs.

$s \leftarrow ADSIGN(A)$ — $A \in \mathcal{AD}$, $s \in \mathcal{A}$. $s = 1$ if $A >_{\mathcal{AD}} 0$, $s = 0$ if $A =_{\mathcal{AD}} 0$ and $s = -1$ if $A <_{\mathcal{AD}} 0$. If the domain has no sign function defined an error occurs.

$s \leftarrow ADINVT(A)$ — $A \in \mathcal{AD}$, $s \in \mathcal{A}$. $s = 1$ if A is invertible in $\mathcal{AD}$ otherwise $s = 0$. If the domain has no invertibility test function defined an error occurs.

$s \leftarrow ADCNST(A)$ — $A \in \mathcal{AD}$, $s \in \mathcal{A}$. $s = 1$ if A is a constant in $\mathcal{AD}$ otherwise $s \neq 1$. If the domain has no constant test function defined an error occurs. The meaning of a constant depends on the domain. In polynomial like domains a constant is in general a polynomial of degree zero.

$A' \leftarrow ADNEG(A)$ — $A, A' \in \mathcal{AD}$. $A' = -A$ is the negative of A. If the domain has no negation function defined an error occurs.

$A' \leftarrow ADINV(A)$ — $A, A' \in \mathcal{AD}$. $A' = A^{-1}$ is the multiplicative inverse of A. If A is not invertible in the domain $\mathcal{AD}$ or the domain has no inverse element function defined an error occurs.

$C \leftarrow ADDIF(A,B)$ — $A, B, C \in \mathcal{AD}$. $C = A - B$ is the difference of A and B. If A and B are not elements the same domain or the domain has no difference function defined an error occurs.

$C \leftarrow ADSUM(A,B)$ — $A, B, C \in \mathcal{AD}$. $C = A + B$ is the sum of A and B. If A and B are not elements the same domain or the domain has no sum function defined an error occurs.

$C \leftarrow ADPROD(A,B)$ — $A, B, C \in \mathcal{AD}$. $C = A * B$ is the product of A and B. If A and B are not elements the same domain or the domain has no product function defined an error occurs.

$C \leftarrow ADQUOT(A, B)$ $A, B, C \in \mathcal{AD}$. $C = A/B$ is the quotient of A and B. If A and B are not elements the same domain or the domain has no quotient function defined or if the quotient does not exist an error occurs.

$B \leftarrow ADEXP(A, n)$ $A, B \in \mathcal{AD}$, $n \in \mathcal{A}^+$. $B = A^n$ is the n-th power of A. If the domain has no product function or no 'one' function defined (in case $n = 0$) an error occurs.

$C \leftarrow ADGCD(A, B)$ $A, B, C \in \mathcal{AD}$. $C = \gcd(A, B)$ is the greatest common divisor of A and B. If the domain has no greatest common divisor function defined an error occurs.

$ADGCDC(A, B; C, A', B')$ $A, B, C, A', B' \in \mathcal{AD}$. $C = \gcd(A, B)$ is the greatest common divisor of A and B. A' and B' are the cofactors of A and B respectively, i.e. $A * A' = C = B * B'$. If $C = 0$ then also $A' = B' = 0$. If the domain has no greatest common divisor function defined an error occurs.

$ADGCDE(A, B; C, A', B')$ $A, B, C, A', B' \in \mathcal{AD}$. $C = \gcd(A, B)$ is the greatest common divisor of A and B. A' and B' are elements of $\mathcal{AD}$ such that the gcd C is a linear combination of A and B with factors A' and B', i.e. $C = A * A' + B * B'$. If $C = 0$ then also $A' = B' = 0$. If the domain has no extended greatest common divisor function defined an error occurs.

$C \leftarrow ADLCM(A, B)$ $A, B, C \in \mathcal{AD}$. $C = \mathrm{lcm}(A, B)$ is the least common multiple of A and B. If the domain has no least common multiple function defined an error occurs.

$L \leftarrow ADFACT(A)$ $A \in \mathcal{AD}$, $L \in \mathcal{LAD}$. L is a list of prime respectively irreducible factors of A. If the domain has no factorization function defined an error occurs.

$B \leftarrow ADCONV(A, d)$ $A \in \mathcal{AD}$, $d, B \in \mathcal{AD}_d$. $B = A_d$. A is converted to an element of domain $\mathcal{AD}_d$. If the conversion is not defined an error occurs.

$B \leftarrow ADFI(d, A)$ $A \in \mathcal{I}$, $d, B \in \mathcal{AD}_d$. $B = A_d$. An integer A is converted to an element of domain $\mathcal{AD}_d$. If the conversion is not defined an error occurs.

$B \leftarrow ADFIP(d, A)$ $A \in \mathcal{IP}_r$, $d, B \in \mathcal{AD}_d$. $B = A_d$. An integral polynomial A is converted to an element of domain $\mathcal{AD}_d$. The number of variables of the domain $\mathcal{AD}_d$ must be equal to r. If the conversion is not defined an error occurs.

$B \leftarrow ADTOIP(A; l)$ $B \in \mathcal{IP}_r$, $A \in \mathcal{AD}_d$, $l \in \mathcal{IP}_r$. l is the least common multiple of the coefficient-denominators. An domain element A is converted to an integral polynomial. If the conversion is not defined an error occurs.

$A \leftarrow ADREAD(d)$ $d, A \in \mathcal{AD}_d$, A domain element A is read from the actual input stream. If no input function is defined an error occurs.

$ADWRIT(A)$ — $A \in \mathcal{AD}$. A domain element A is written to the actual output stream. If no output function is defined an error occurs.

$d \leftarrow ADDDREAD()$ — $d \in \mathcal{AD}_d$ is a domain element. A domain descriptor is read from the actual input stream and an element (in general 0) of the domain $\mathcal{AD}_d$ is returned. If no input function is defined an error occurs. The accepted syntax (with start symbol `spec`) is:

```
spec     = dsymbol ddescription
dsymbol  = ( "INT" | "RN" | "MI" | "MD" | "APF" |
             "IP"  | "RP" | "RF" | "AF" |
             "C"   | "Q"  | "O"  | "FF" )
```

With the context conditions:

1. `ddescription` depends on the domain symbol and the syntax is discussed in the next subsection;
2. not all `dsymbols` may always be available and the available ones are discussed in the next subsection.

$ADDDWRIT(d)$ — $d \in \mathcal{AD}_d$ is a domain element. The domain descriptor of element d is written to the actual output stream. If no output function is defined an error occurs.

$V \leftarrow ADVLDD(d)$ — $d \in \mathcal{AD}_d$ is a domain element. V gets the variable list from domain element d. If the element has no variable list, or if no variable list function is defined an error occurs.

This concludes the summary of the arbitrary domain functions.

7.8.2 Available Domains

The general format of a domain specification is as defined in the syntax of `ADDDREAD`:

<domain symbol> <domain description>

Where <domain symbol> is a unique short name of a domain and <domain description> is a partly optional list of further specifications which are meaningful for the particular domain. The available domains are discussed in the this subsection. In the following subsections we discuss the various domains in more detail together with the specifications and with some examples.

Supported domains and domain symbols are:

INT Integral Numbers

RN Rational Numbers

MI Integral Numbers modulo an Integer

MD Integral Digits modulo a Digit

APF Arbitrary Precision Floating Point Numbers

IP Integral Polynomials

RP Rational Polynomials

RF Rational Functions

AF Algebraic Numbers

FF Finite Field Numbers

C Complex Numbers

Q Quaternion Numbers

O Octonion Numbers

The acutally available domains can be listed by the '`DOMAINS`' command.

```
DOMAINS. (*show available domains*)
```

The output shows all defined domains (in the order in which they are defined to the system).

```
List of all defined domains

AF Algebraic Number
INT Integer
IP Integral Polynomial
MD Modular Digit
MI Modular Integer
FF Finite Field
RF Rational Function
RN Rational Number
C Complex Number
Q Quaternion Number
O Octonion Number
RP Rational Polynomial
APF Arbitrary Precision Floating Point

13 defined domains.
```

The domain descriptions for some of the different domains are discussed in the next subsections together with some examples.

7.8.3 Integral Numbers

INT

For integral numbers there are no further options. If the computation requires to take quotients of integral numbers and the remainder is non–zero, an error message is displayed and the user is asked for interaction.

In the following example we read a list of distributive polynomials over the integral number and compute the 3-rd power of the first polyomial. First we read the domain symbol '`INT`'

with the domain descriptor read function 'ADDDREAD' and also print the descriptor with 'ADDDWRIT'. Then we setup a variable list using the 'LIST' function on strings of variable names 'V:=LIST("x","y","z")'. A list of two polynomials '(7 y x**4 z ...)' and '(y + ...)' is then read by 'DILRD', which requires as inputs a variable list 'V' and a domain descriptor 'dp'. The polynomials are then printed by 'DILWR'. Next the 3-rd power of the first polynomial is computed using the 'DILEX' function. Finally this polynomial is put into a list and printed on the output stream.

```
(*Integer ---------------------------------- *)
(*domain descriptor *)
dp:=ADDDREAD().          INT

ADDDWRIT(dp).

(*variable list *)
V:=LIST("x","y","z").

(*Polynomials *)
P:=DILRD(V,dp).

(
( 7 y x**4 z + 9 + 13 z**3 - x ),
( y + z**5 + 77 )
)

DILWR(P,V).

(* computations *)
p:=FIRST(P). q:=DIPEXP(p,3).

Q:=LIST(q). DILWR(Q,V).
```

Next we discuss the output as produced by the above example. '(2 0)' is the internal representation of the interger domain descriptor. 'INT (* Integer *)' is the printed domain descriptor, the comment gives a more verbose description of the domain. '((56) (58) (60))' is the internal representation of the variable list. Then the input polynomials are printed. Finally the time for computing the 3-rd power of the first polynomial is printed followed by the resulting polynomial itself.

```
ANS: (2 0)
MAS: INT (* Integer *)
ANS: ((56) (58) (60))

( 13 z**3 +7 x**4 y z - x +9 )
(  z**5 + y +77 )

Time: read = 16, eval = 34, print = 16, gc = 0.

( 2197 z**9 +3549 x**4 y z**7 -507 x z**6 +4563 z**6 +1911 x**8 y**2 z**5 -546
x**5 y z**4 +4914 x**4 y z**4 +343 x**12 y**3 z**3 +39 x**2 z**3 -702 x z**3 +3
159 z**3 -147 x**9 y**2 z**2 +1323 x**8 y**2 z**2 +21 x**6 y z -378 x**5 y z +1
701 x**4 y z - x**3 +27 x**2 -243 x +729 )
```

7.8.4 Rational Numbers

RN [s]

For rational numbers there is one option, the so called decimal flag 's'. It controls the printout of the rational numbers as follows:

If '$s = -1$' (this is the default) the rational numbers are printed as fractions of integral numbers (e.g. 33/100). If '$s \geq 0$' the rational numbers are approximated by the nearest decimal fractions with s decimal digits following the decimal point (e.g. if s=5, 2/3 is printed as 0.66667). More precisely the decimal fraction approximation $d.d_1d_2 \ldots d_{s-1}d_s$ were $d \in \mathbf{Z}$, $d_i \in \{0, 1, \ldots, 9\}$ $(1 \leq i \leq s)$ for a rational number a means:

$$|a - d.d_1d_2 \ldots d_{s-1}d_s| \leq \frac{1}{2 \cdot 10^s}.$$

In the following example we read a list of distributive polynomials over the rational numbers, to be printed with 10 places after the decimal point, and compute the 3-rd power of the first polyomial. First we read the domain symbol '`RN`' and the decimal flag s as 10 with the domain descriptor read function '`ADDDREAD`' and also print the descriptor with '`ADDDWRIT`'. The rest of the example is as in the previous example with integer coefficients.

```
(*Rational Number  ---------------------------- *)
(*domain descriptor          <print precision> *)
dp:=ADDDREAD().          RN 10

ADDDWRIT(dp).

(*variable list *)
V:=LIST("x","y","z").

(*Polynomials *)
P:=DILRD(V,dp).

(
( 7 y x**4 z + 9 + 13 z**3 - x ),
( y + z**5 + 77 )
)

DILWR(P,V).

(* computations *)
p:=FIRST(P). q:=DIPEXP(p,3).

Q:=LIST(q). DILWR(Q,V).
```

Next we discuss the output as produced by the above example. '`(7 0 10)`' is the internal representation of the rational number domain descriptor with decimal flag 10. '`RN 10  (* Rational Number *)`' is the printed domain descriptor, the comment gives a more verbose description of the domain. The rest of the example is as in the previous example with integer coefficients except that all numbers are printed with 10 places after the decimal point.

```
ANS: (7 0 10)
MAS: RN 10  (* Rational Number *)
ANS: ((56) (58) (60))

( 13.0000000000 z**3 +7.0000000000 x**4 y z - x +9.0000000000 )
(  z**5 + y +77.0000000000 )
```

```
   Time: read = 0, eval = 50, print = 17, gc = 0.

( 2197.0000000000 z**9 +3549.0000000000 x**4 y z**7 -507.0000000000 x z**6 +456
3.0000000000 z**6 +1911.0000000000 x**8 y**2 z**5 -546.0000000000 x**5 y z**4 +
4914.0000000000 x**4 y z**4 +343.0000000000 x**12 y**3 z**3 +39.0000000000 x**2
 z**3 -702.0000000000 x z**3 +3159.0000000000 z**3 -147.0000000000 x**9 y**2 z*
*2 +1323.0000000000 x**8 y**2 z**2 +21.0000000000 x**6 y z -378.0000000000 x**5
 y z +1701.0000000000 x**4 y z - x**3 +27.0000000000 x**2 -243.0000000000 x +72
9.0000000000 )
```

7.8.5 Modular Integers and Digits

MI m

MD m

Integral numbers or digits modulo the integer 'm'. 'MI' uses the long integer arithmetic and 'MD' uses arithmetic of β-digits only. During input of the domain descriptor it is checked whether 'm' is prime or not and the result of this check is printed in a comment. If the computation requires to take inverses modulo 'm' and they do not exist in the case of 'm' beeing not prime, an error message is displayed and user interaction is requested.

In the following two examples we read a list of distributive polynomials over modular digits with modulus 7 respectively over modular integers with modulus 19 and compute the 3-rd power of the first polyomial. First we read the domain symbol 'MD' and the modulus 7 respectively 'MI' and the modulus 19 with the domain descriptor read function 'ADDDREAD' and also print the descriptor with 'ADDDWRIT'. The rest of the example is as in the previous examples with integer or rational number coefficients.

```
(*Modular Digit ----------------------------- *)
(*domain descriptor          <modulus> *)
dp:=ADDDREAD().          MD 7

ADDDWRIT(dp).

(*variable list *)
V:=LIST("x","y","z").

(*Polynomials *)
P:=DILRD(V,dp).

(
( 7 y x**4 z + 9 + 13 z**3 - x ),
( y + z**5 + 717 )
)

DILWR(P,V).

(* computations *)
p:=FIRST(P). q:=DIPEXP(p,3).

Q:=LIST(q). DILWR(Q,V).

(*Modular Integer ---------------------------- *)
(*domain descriptor          <modulus> *)
dp:=ADDDREAD().          MI 19

ADDDWRIT(dp).
```

```
(*variable list *)
V:=LIST("x","y","z").

(*Polynomials *)
P:=DILRD(V,dp).

(
( 7 y x**4 z + 13 + 19 z**3 - x ),
( y + z**5 + 77 )
)

DILWR(P,V).

(* computations *)
p:=FIRST(P). q:=DIPEXP(p,3).

Q:=LIST(q). DILWR(Q,V).
```

Next we discuss the output as produced by the above examples. '`(4 0 7 1)`' is the internal representation of the modular digit domain descriptor with modulus 7. (1 indicates that 7 is a prime number.) '`MD 7 (* prime. *) (* Modular Digit *)`' is the printed domain descriptor for modulus 7. The first comment '`(* prime. *)`' indicates, that the modulus is a prime number and the second comment gives a more verbose description of the domain. The rest of the example is as in the previous examples with integer or rational number coefficients. The differences to the previous examples is that the coefficients of the input polynomials are reduced modulo 7, respectively 19 and thus some might have been become zero and the respective terms disappear. E.g. '`7 y x**4 z`' reduces to zero, and '`19 z**3`' reduces to '`6 z**3`' modulo 7.

```
ANS: (4 0 7 1)
MAS: MD 7 (* prime. *) (* Modular Digit *)
ANS: ((56) (58) (60))

( 6 z**3 +6 x +2 )
(  z**5 + y +3 )

Time: read = 0, eval = 17, print = 16, gc = 0.

( 6 z**9 +4 x z**6 +6 z**6 +4 x**2 z**3 +5 x z**3 +2 z**3 +6 x**3 +6 x**2 +2 x
 +1 )

ANS: (5 0 19 1)
MAS: MI 19 (* prime. *) (* Modular Integer *)
ANS: ((56) (58) (60))

( 7 x**4 y z +18 x +13 )
(  z**5 + y +1 )

Time: read = 17, eval = 17, print = 16, gc = 0.

(  x**12 y**3 z**3 +5 x**9 y**2 z**2 +11 x**8 y**2 z**2 +2 x**6 y z +5 x**5 y z
 +15 x**4 y z +18 x**3 + x**2 +6 x +12 )
```

7.8.6 Arbitrary precision floating point numbers

APF [*s*]

Arbitrary precision floating point numbers. The flag '*s*' defines the precision of the numbers, i.e. the minimal number of decimal digits in the internal representation.

In the following example we read a list of distributive polynomials over arbitrary precision floating point numbers with precision 50 and compute the 3-rd power of the first polyomial. First we read the domain symbol '`APF`' and the precision 50 with the domain descriptor read function '`ADDDREAD`' and also print the descriptor with '`ADDDWRIT`'. The rest of the example is as in the previous examples with integer or rational number coefficients.

```
(*Arbitrary precision floating point -------- *)
(*domain descriptor         <precision> *)
dp:=ADDDREAD().          APF 50

ADDDWRIT(dp).

(*variable list *)
V:=LIST("x","y","z").

(*Polynomials *)
P:=DILRD(V,dp).

(
( 7 y x**4 z + 9 + 13 z**3 - x ),
( y + z**5 + 77 )
)

DILWR(P,V).

(* computations *)
p:=FIRST(P). q:=DIPEXP(p,3).

Q:=LIST(q). DILWR(Q,V).
```

Next we discuss the output as produced by the above examples. The first line indicates that the precision is defined to be at least 50 decimal digits and exactly 173 binary digits. '`(8 (0 0) 50)`' is the internal representation of the arbitrary precision floating point number domain descriptor with precision 50.
'`APF 50  (* Arbitrary Precision Floating Point *)`' is the printed domain descriptor for precision 50. The comment gives a more verbose description of the domain. The rest of the example is as in the previous examples with integer or rational number coefficients.

```
MAS: Floating point precision set to 50 digits = 173 bits.
ANS: (8 (0 0) 50)
MAS: APF 50  (* Arbitrary Precision Floating Point *)
ANS: ((56) (58) (60))

( 0.13000000000000000000000000000000000000000000000000E2 z**3 +0.7000000000000
0000000000000000000000000000000000000E1 x**4 y z -0.1000000000000000000000000000
0000000000000000000000E1 x +0.90000000000000000000000000000000000000000000000000
0000E1 )

( 0.10000000000000000000000000000000000000000000000000E1 z**5 +0.1000000000000
0000000000000000000000000000000000000E1 y +0.77000000000000000000000000000000000
00000000000000000E2 )

   Time: read = 0, eval = 234, print = 50, gc = 0.

( 0.21970000000000000000000000000000000000000000000000E4 z**9 +0.3549000000000
```

```
000000000000000000000000000000000000E4 x**4 y z**7 -0.5069999999999999999999
999999999999999999999999999E3 x z**6 +0.4563000000000000000000000000000000000
0000000000000E4 z**6 +0.191100000000000000000000000000000000000000000000000E4 x
**8 y**2 z**5 -0.5459999999999999999999999999999999999999999999999999E3 x**5 y z
**4 +0.491400000000000000000000000000000000000000000000000E4 x**4 y z**4 +0.343
000000000000000000000000000000000000000000000000E3 x**12 y**3 z**3 +0.389999999
999999999999999999999999999999999999999E2 x**2 z**3 -0.701999999999999999999
999999999999999999999999999999E3 x z**3 +0.315900000000000000000000000000000000
000000000000000E4 z**3 -0.146999999999999999999999999999999999999999999999999E3
 x**9 y**2 z**2 +0.132300000000000000000000000000000000000000000000000000E4 x**8 y
**2 z**2 +0.209999999999999999999999999999999999999999999999999999E2 x**6 y z -0.3
779999999999999999999999999999999999999999999999999E3 x**5 y z +0.17010000000000
000000000000000000000000000000000000E4 x**4 y z - x**3 +0.26999999999999999999
999999999999999999999999999999E2 x**2 -0.242999999999999999999999999999999999999
999999999999E3 x +0.729000000000000000000000000000000000000000000000000000E3 )
```

7.8.7 Integral Polynomials

IP (x1,...,xr)

For the integral polynomial domain the variables need to be specified in the variable list '(x1,...,xr)'. The 'xi' denote variable names (starting with a letter, followed by letters and/or decimal digits). The elements are represented as multivariate polynomials with integral coefficients. On input the coefficients should be enclosed in parenthesis: '(p)' where p denotes a multivariate polynomial with *integer* coefficients. If the computation requires to take quotients of integral numbers and the remainder is non–zero, an error message is displayed and user interaction is requested.

In the following example we read a list of distributive polynomials over integral polynomials in the variables a, b, c and compute the 3-rd power of the first polyomial. First we read the domain symbol '`IP`' and the list of variables '`(a,b,c)`' with the domain descriptor read function '`ADDDREAD`' and also print the descriptor with '`ADDDWRIT`'. As usual the rightmost variable '`c`' defines the main variable. The rest of the example is as in the previous examples with integer or rational number coefficients except that we use new polynomials. The coefficients '`( a + b + c )`' and '`( a )`' are enclosed in parenthesis to allow the distributive polynomial input routines to switch to integral polynomial input routines for the input of the coefficients.

```
(*Integral Polynomial ----------------------- *)
   (*domain descriptor        <var list> *)
   dp:=ADDDREAD().          IP(a,b,c)

   ADDDWRIT(dp).

   (*variable list 2 *)
   V:=LIST("x","y","z").

   (*Polynomials *)
   P:=DILRD(V,dp).

   (
   ( ( a + b + c ) z**3 - x ),
   ( y + z**5 + ( a ) )
   )

   DILWR(P,V).
```

```
(* computations *)
q:=DIPEXP(FIRST(P),3).

DILWR(LIST(q),V).
```

Next we discuss the output as produced by the above examples.
'(3 0 3 ((10) (12) (14)))' is the internal representation of the integral polynomial domain descriptor. '((10) (12) (14)))' is the internal representation of the variable list '(a,b,c)' and the second '3' denotes, that the integral polynomials are in 3 variables. The comment gives a more verbose description of the domain. The rest of the example is as in the previous examples with integer or rational number coefficients.

```
ANS: (3 0 3 ((10) (12) (14)))
MAS: IP(a,b,c) (* Integral Polynomial *)
ANS: ((56) (58) (60))

( (  c + b + a ) z**3 - x )

(  z**5 + y +  a  )

Time: read = 16, eval = 34, print = 33, gc = 0.

( (  c**3 +3 b c**2 +3 a c**2 +3 b**2 c +6 a b c +3 a**2 c + b**3 +3 a b**2 +3
a**2 b + a**3 ) z**9 -( 3 c**2 +6 b c +6 a c +3 b**2 +6 a b +3 a**2 ) x z**6 +(
 3 c +3 b +3 a ) x**2 z**3 - x**3 )
```

7.8.8 Rational Polynomials

RP (x1,...,xr)

For the rational polynomial domain the variables need to be specified in the variable list '(x1,...,xr)'. The 'xi' denote variable names (starting with a letter, followed by letters and/or decimal digits). The elements are represented as multivariate polynomials with rational coefficients. On input the coefficients should be enclosed in parenthesis: '(p)' where p denotes a multivariate polynomial with *rational* coefficients. If the computation requires to take quotients and the remainder is non–zero, an error message is displayed and user interaction is requested.

In the following example we read a list of distributive polynomials over rational polynomials in the variables a, b, c and compute the 3-rd power of the first polyomial. First we read the domain symbol 'RP' and the list of variables '(a,b,c)' with the domain descriptor read function 'ADDDREAD' and also print the descriptor with 'ADDDWRIT'. As usual the leftmost variable 'c' defines the main variable. The rest of the example is as in the previous example over integral polynomials. The coefficients '(a + b + c)' and '(a)' are enclosed in parenthesis to allow the distributive polynomial input routines to switch to rational polynomial input routines for the input of the coefficients.

```
(*Rational Polynomial ----------------------- *)
   (*domain descriptor        <var list> *)
   dp:=ADDDREAD().          RP(a,b,c)

   ADDDWRIT(dp).
```

```
(*variable list 2 *)
V:=LIST("x","y","z").

(*Polynomials *)
P:=DILRD(V,dp).

(
( ( a + b + c ) z**3 - x ),
( y + z**5 + ( a ) )
)

DILWR(P,V).

(* computations *)
q:=DIPEXP(FIRST(P),3).

DILWR(LIST(q),V).
```

Next we discuss the output as produced by the above examples.
'`(8 0 3 ((10) (12) (14)))`' is the internal representation of the rational polynomial domain descriptor. '`((10) (12) (14)))`' is the internal representation of the variable list '`(a,b,c)`' and the second '`3`' denotes, that the rational polynomials are in 3 variables. The comment gives a more verbose description of the domain. The rest of the example is as in the previous example with integral polynomial coefficients.

```
ANS: (8 0 3 ((10) (12) (14)))
MAS: RP(a,b,c) (* Rational Polynomial *)
ANS: ((56) (58) (60))

( (  c + b + a ) z**3 - x )

(  z**5 + y +  a  )

Time: read = 16, eval = 34, print = 50, gc = 0.

( (  c**3 +3 b c**2 +3 a c**2 +3 b**2 c +6 a b c +3 a**2 c + b**3 +3 a b**2 +3
a**2 b + a**3 ) z**9 -( 3 c**2 +6 b c +6 a c +3 b**2 +6 a b +3 a**2 ) x z**6 +(
 3 c +3 b +3 a ) x**2 z**3 - x**3 )
```

7.8.9 Rational Functions

RF (x1,...,xr)

For the rational function domain the variables need to be specified in the variable list '(x1,...,xr)'. The 'xi' denote variable names (starting with a letter, followed by letters and/or decimal digits). The rational functions are fractions of multivariate integral polynomials reduced to lowest terms. I.e. gcd(denominator,nominator) = 1 and leading base coefficient (nominator) > 0. On input the denominators and nominators should be enclosed in parenthesis: '(p)/(q)' where p and q denote multivariate polynomials with *integer* coefficients.

In the following example we read a list of distributive polynomials over rational functions in the variables a, b, c and compute the 3-rd power of the first polyomial. First we read the domain symbol '`IP`' and the list of variables '`(a,b,c)`' with the domain descriptor read function '`ADDDREAD`' and also print the descriptor with '`ADDDWRIT`'. As usual the leftmost

variable 'c' defines the main variable. The rest of the example is as in the previous examples with integer or rational number coefficients except that we use new polynomials. The coefficients '(a + b + c)(a**2 - 4)/' and '(a)' are enclosed in parenthesis to allow the distributive polynomial input routines to switch to rational function input routines for the input of the coefficients.

```
(*Rational Function ---------------------- *)
(*domain descriptor        <var list> *)
dp:=ADDDREAD().          RF(a,b,c)

ADDDWRIT(dp).

(*variable list 2 *)
V:=LIST("x","y","z").

(*Polynomials *)
P:=DILRD(V,dp).

(
( ( a + b + c )/( a**2 - 4 ) z**3 - x ),
( y + z**5 + ( a ) )
)

DILWR(P,V).

(* computations *)
q:=DIPEXP(FIRST(P),3).

DILWR(LIST(q),V).
```

Next we discuss the output as produced by the above examples.
'(6 (3 0) ((10) (12) (14)))' is the internal representation of the rational function domain descriptor. '((10) (12) (14)))' is the internal representation of the variable list '(a,b,c)'. The comment gives a more verbose description of the domain. The rest of the example is as in the previous examples with integer or rational number coefficients.

```
ANS: (6 (3 0) ((10) (12) (14)))
MAS: RF(a,b,c) (* Rational Function *)
ANS: ((56) (58) (60))

( (  c + b + a )/(  a**2 -4 ) z**3 - x )

(  z**5 + y +  a  )

Time: read = 17, eval = 333, print = 33, gc = 0.

( (  c**3 +3 b c**2 +3 a c**2 +3 b**2 c +6 a b c +3 a**2 c + b**3 +3 a b**2 +3
a**2 b + a**3 )/(  a**6 -12 a**4 +48 a**2 -64 ) z**9 -( 3 c**2 +6 b c +6 a c +3
 b**2 +6 a b +3 a**2 )/(  a**4 -8 a**2 +16 ) x z**6 +( 3 c +3 b +3 a )/(  a**2
-4 ) x**2 z**3 - x**3 )
```

7.8.10 Algebraic Numbers

AF (x, p(x), [(l,r) [, s]])

Algebraic numbers, i.e. elements of $\mathbf{Q}[x]/(p(x))$, are represented as univariate polynomials modulo an univariate polynomial $p \in \mathbf{Q}[x]$, together with an optional isolating intervall

(l, r) for the algebraic number α with $p(\alpha) = 0$ and $\alpha \in (l, r)$. If p is irreducible, then $\mathbf{Q}[x]/(p(x)) = \mathbf{Q}(\alpha)$ is a field. 'x' denotes the variable name, in which the algebraic numbers are written. 'p(x)' denotes the polynomial modulo which the computation is to be done. '(l,r)' is an optional isolating intervall, $(l, r) \in \mathbf{Q} \times \mathbf{Q}$, for the real algebraic number $\alpha \in (l, r] \subseteq \mathcal{R}$. 's' is the optional decimal flag. If 's $= -1$' the algebraic numbers are printed as polynomials in 'x'. If 's ≥ 0' decimal approximations for the algebraic numbers are printed. As with the case of the decimal approximation of the rational numbers we have:

$$|\beta - d.d_1 d_2 \ldots d_{s-1} d_s| \leq \frac{1}{2 \cdot 10^s}.$$

for $\beta \in \mathbf{Q}(\alpha)$, and $d \in \mathbf{Z}$, $d_i \in \{0, 1, \ldots, 9\}$ $(1 \leq i \leq s)$. The isolating intervall is only required if the decimal flag 's' is non-negative.

The polynomial 'p' is checked whether it is (1) not squarefree, (2) squarefree or (3) irreducible. The result of this check is printed together with the domain descriptor in a comment. It depends on the package if the computation can proceed correctly, e.g. real root isolation requires 'p' to be squarefree or Gröbner bases computation requires inverses in $\mathbf{Q}(\alpha)$, so 'p' must be irreducible over $\mathbf{Q}$. If the computation requires to take inverses modulo 'p' and they do not exist in the case of 'p' beeing not irreducible, an error message is displayed and user interaction is requested.

In the following three examples we read a list of distributive polynomials over three different algebraic number fields and compute the 3-rd power of the first polyomial. First we read the domain symbol '`AF`' and the specification of the algebraic number ring with the domain descriptor read function '`ADDDREAD`' and also print the descriptor with '`ADDDWRIT`'. The rest of the example is as in the previous examples with integer or rational number coefficients except that we use new polynomials. The coefficients enclosed in parenthesis to allow the distributive polynomial input routines to switch to rational function input routines for the input of the coefficients.

The first algebraic number ring is the field $\mathbf{Q}(i)$, where i is the imaginary number. The specification '`( i, ( i**2 + 1 ) )`' defines the variable '`i`' and the defining irreducible polynomial '`( i**2 + 1 )`' for 'i'.

```
(*Complex Algebraic Number ---------------------- *)
(*domain descriptor      <var> <minimal polynomial> *)
dp:=ADDDREAD().          AF( i, ( i**2 + 1 ) )

ADDDWRIT(dp).

(*variable list 2 *)
V:=LIST("x","y","z").

(*Polynomials *)
P:=DILRD(V,dp).

(
( ( 3 + 4 i ) z**3 - x ),
( y + z**5 + ( i**2 + 2 ) )
)

DILWR(P,V).

(* computations *)
q:=DIPEXP(FIRST(P),3).
```

```
DILWR(LIST(q),V).
```

Next we discuss the output as produced by the above example.
'(1 0 (2 (1 1) 0 (1 1)) (2 1 0 1) 1 ((26)) () -1)' is the internal representation of the algebraic number domain descriptor. '((26))' is the internal representation of the variable list '(i)'. '()' denotes an empty isolating interval and '-1' denotes the decimal flag $s = -1$ (which are the defaults). '(2 (1 1) 0 (1 1))' is the rational minimal polynomial and '(2 1 0 1)' is the integral minimal polynomial. '1' denotes that the polynomials are irreducible. The first comment '(* prime *)' indicates that the polynomial is prime and the second comment gives a more verbose description of the domain. The rest of the example is as in the previous examples with integer or rational number coefficients.

```
ANS: (1 0 (2 (1 1) 0 (1 1)) (2 1 0 1) 1 ((26)) () -1)
MAS: AF( i, (  i**2 +1 ) ) (* prime *)  (* Algebraic Number *)
ANS: ((56) (58) (60))

( ( 4 i +3 ) z**3 - x )

(  z**5 + y + 1  )

Time: read = 0, eval = 33, print = 17, gc = 0.

( ( 44 i -117 ) z**9 -( 72 i -21 ) x z**6 +( 12 i +9 ) x**2 z**3 - x**3 )
```

The second example is a real algebraic number ring $\mathbf{Q}[\alpha]$, where α is a root of $(X^2-2)(X-3)$. The specification '(w2, ((w2**2 - 2) (w2 - 3)))' defines the variable 'w2' and the defining squarefree polynomial '(w2**3 -3 w2**2 -2 w2 +6)' for 'w2'. '(1, 2)' defines the isolating interval $(1, 2]$ for $\sqrt{2}$ and '10' defines the decimal flag to be 10.

```
(*Real Algebraic Number 1---------------------- *)
(*domain descriptor      (<var>, <minimal polynomial>[, (r1,r2)[, s]] ) *)
dp:=ADDDREAD().          AF( w2, ( (w2**2 - 2) (w2 - 3) ), ( 1, 2 ), 10 )

ADDDWRIT(dp).

(*variable list 2 *)
V:=LIST("x","y","z").

(*Polynomials *)
P:=DILRD(V,dp).

(
( ( 1 + w2  ) z**3 - x ),
( y + z**5 + ( w2**2 + 32 ) )
)

DILWR(P,V).

(* computations *)
q:=DIPEXP(FIRST(P),3).

DILWR(LIST(q),V).
```

Next we discuss the output as produced by the above example.
'((1 0 (3 (1 1) 2 (-3 1) 1 (-2 1) 0 (6 1)) (3 1 2 -3 1 -2 0 6) 2

`((54 2)) ((1 1) (2 1)) 10)'` is the internal representation of the algebraic number domain descriptor. `'((54 2))'` is the internal representation of the variable list `'(w2)'`. `'((1 1) (2 1))'` is the internal representation of the isolating intervall $(1, 2]$ and `'10'` denotes the decimal flag $s = 10$. `'(3 (1 1) 2 (-3 1) 1 (-2 1) 0 (6 1))'` is the rational minimal polynomial and `'(3 1 2 -3 1 -2 0 6)'` is the integral minimal polynomial. `'2'` denotes that the polynomials are irreducible. The first comment `'(* squarefree *)'` indicates that the polynomial is squarefree and the second comment gives a more verbose description of the domain. The rest of the example is as in the previous examples with integer or rational number coefficients, only the polynomial coefficients are printed with 10 places after the decimal point.

```
MAS: (  w2**3 -3 w2**2 -2 w2 +6 )
ANS: (1 0 (3 (1 1) 2 (-3 1) 1 (-2 1) 0 (6 1)) (3 1 2 -3 1 -2 0 6) 2
          ((54 2)) ((1 1) (2 1)) 10)
Time: read = 17, eval = 216, print = 0, gc = 0.
MAS: AF( w2, (  w2**3 -3 w2**2 -2 w2 +6 ), ( 1, 2 ), 10 )
                (* squarefree *)  (* Algebraic Number *)
ANS: ((56) (58) (60))

( 2.4142135624 z**3 - x )

(  z**5 + y +34.0000000000 )

Time: read = 0, eval = 766, print = 0, gc = 0.

( 14.0710678119 z**9 -17.4852813742 x z**6 +7.2426406871 x**2 z**3 - x**3 )
```

The third example is a real algebraic number ring $\mathbf{Q}[\alpha]$, where α is a root of $(X^2 - 2)(X - 3)^2$. The specification `'( w2, ( (w2**2 - 2) (w2 - 3)**2 ) )'` defines the variable `'w2'` and the defining reducible and non squarefree polynomial `'( w2**4 -6 w2**3 +7 w2**2 +12 w2 -18 )'` for 'w2'. `'( 1, 2 )'` defines the isolating interval $(1, 2]$ for $\sqrt{2}$. No decimal flag is specified.

```
(*Real Algebraic Number 2---------------------- *)
(*domain descriptor      (<var>, <minimal polynomial>[, (r1,r2)[, s]] ) *)
dp:=ADDDREAD().          AF( w2, ( (w2**2 - 2) (w2 - 3)**2 ), ( 1, 2 ) )

ADDDWRIT(dp).

(*variable list 2 *)
V:=LIST("x","y","z").

(*Polynomials *)
P:=DILRD(V,dp).

(
( ( 1 + w2  ) z**3 - x ),
( y + z**5 + ( w2**2 + 32 ) )
)

DILWR(P,V).

(* computations *)
q:=DIPEXP(FIRST(P),3).

DILWR(LIST(q),V).
```

Next we discuss the output as produced by the above example.
'(1 0 (4 (1 1) 3 (-6 1) 2 (7 1) 1 (12 1) 0 (-18 1))
(4 1 3 -6 2 7 1 12 0 -18) 0 ((54 2)) ((1 1) (2 1)) -1)' is the internal representation of the algebraic number domain descriptor. '((54 2))' is the internal representation of the variable list '(w2)'. '((1 1) (2 1))' is the internal representation of the isolating intervall $(1, 2]$ and '-1' denotes the decimal flag $s = -1$.
'(4 (1 1) 3 (-6 1) 2 (7 1) 1 (12 1) 0 (-18 1))' is the rational minimal polynomial and '(4 1 3 -6 2 7 1 12 0 -18)' is the integral minimal polynomial. '0' denotes that the polynomials are reducible and not squarefree. The first comment '(* reducible *)' indicates that the polynomial is reducible and not squarefree and the second comment gives a more verbose description of the domain. The rest of the example is as in the previous examples with integer or rational number coefficients. The polynomial coefficients are printed as polynomials in 'w2', since the decimal flag was -1 by default.

```
MAS: (  w2**4 -6 w2**3 +7 w2**2 +12 w2 -18 )
ANS: (1 0 (4 (1 1) 3 (-6 1) 2 (7 1) 1 (12 1) 0 (-18 1))
          (4 1 3 -6 2 7 1 12 0 -18) 0 ((54 2)) ((1 1) (2 1)) -1)
Time: read = 16, eval = 50, print = 17, gc = 0.
MAS: AF( w2, (  w2**4 -6 w2**3 +7 w2**2 +12 w2 -18 ), ( 1, 2 ) )
                 (* reducible *)  (* Algebraic Number *)
ANS: ((56) (58) (60))

( (  w2 +1 ) z**3 - x )

(  z**5 + y +(  w2**2 +32 ) )

Time: read = 0, eval = 16, print = 0, gc = 0.

( (  w2**3 +3 w2**2 +3 w2 +1 ) z**9 -( 3 w2**2 +6 w2 +3 ) x z**6 +( 3 w2 +3 )
 x**2 z**3 - x**3 )
```

7.8.11 Gröbner bases over various domains

In this section we list two examples of Gröbner base computations over the domains of modular digits and rational functions. The input that follows is as described in the examples before, except that the algorithm for Gröbner base computation 'DIPGB' is called.

```
(*Modular Digit ------------------------------ *)
(*domain descriptor          <modulus> *)
dp:=ADDDREAD().          MD 17

ADDDWRIT(dp).

(*variable list *)
V:=LIST("B","S","T","Z","P","W"). xxx:=DIPVDEF(V).

(*Polynomials *)
P:=DILRD(V,dp).

(
( 45 P + 35 S - 165 B - 36 ),
( 35 P + 40 Z + 25 T - 27 S ),
( 15 W + 25 S P + 30 Z - 18 T - 165 B**2 ),
( - 9 W + 15 T P + 20 S Z ),
( P W + 2 T Z - 11 B**3 ),
```

```
   ( 99 W - 11 B S + 3 B**2 )
   )

   (*Syntax: <polynomial list> *)
   DILWR(P,V).

   (* computations *)
   Q:=DIPGB(P,1).
   DILWR(Q,V).
```

The computation takes place in the statement '`Q:=DIPGB(P,1).`' '`P`' is the list of input polynomials and '`1`' is the trace flag, which requests intermediate output of the reduced S-polynomials.

```
   ANS: (4 0 17 1)
   MAS: MD 17 (* prime. *) (* Modular Digit *)
   ANS: ((13) (47) (49) (61) (41) (55))

   ( 11 P + S +5 B +15 )
   (  P +6 Z +8 T +7 S )
   ( 15 W +8 S P +13 Z +16 T +5 B**2 )
   ( 8 W +15 T P +3 S Z )
   (  P W +2 T Z +6 B**3 )
   ( 14 W +6 B S +3 B**2 )

   2216 ms,  326 cells,
   1 crit3,  1 crit4,  1 spoly,  1 hpoly,
   15 pairs,  14 restp,  0.933+ quot.

   H=( 15 Z +3 T +8 S +12 B +2 )

   4466 ms,  752 cells,
   3 crit3,  2 crit4,  1 spoly,  1 hpoly,
   20 pairs,  17 restp,  0.850 quot.

   H=( 13 S T +5 B T +15 T +15 S**2 +8 S +10 B**2 )

   ...

   40066 ms,  41062 cells,
   152 crit3,  14 crit4,  4 spoly,  4 hpoly,
   152 pairs,  0 restp,  0.000 quot.

   Time: read = 0, eval = 42800, print = 100, gc = 0.

( 11 B**10 +13 B**9 +13 B**8 +8 B**7 +2 B**6 + B**5 +14 B**4 +4 B**3 +8 B**2 +6
  B +3 )
( 11 S +12 B**8 +2 B**7 +14 B**6 +9 B**5 +11 B**4 +12 B**3 +16 B**2 +5 B +6 )
( 5 T +12 B**9 +5 B**8 +13 B**7 +7 B**6 +5 B**5 +15 B**4 +2 B**3 +13 B**2
  +11 B +9 )
( 15 Z +3 B**9 +13 B**8 +13 B**7 +2 B**6 +9 B**5 +8 B**3 +9 B**2 +3 B +8 )
( 11 P +2 B**8 +6 B**7 +8 B**6 +10 B**5 +16 B**4 +2 B**3 +14 B**2 +3 B +16 )
( 14 W +12 B**9 +2 B**8 +14 B**7 +9 B**6 +11 B**5 +12 B**4 +16 B**3 +8 B**2
  +6 B )
```

In the next example we compute a Gröbner base over the rational function field in 4 variables. Except of the domain descriptor and the input polynomials, the example is as the previous example.

```
(*Rational Function ---------------------- *)
(*domain descriptor        <var list> *)
dp:=ADDDREAD().          RF(A1,A2,A3,A4)

ADDDWRIT(dp).

(*variable list 2 *)
V:=LIST("X1","X2","X3","X4").  xxx:=DIPVDEF(V).

(*Polynomials *)
P:=DILRD(V,dp).

(
( X4 - ( A4 - A2 )),
( X1 + X2 + X3 + X4 - ( A1 + A3 + A4 )),
( X1 X3 + X1 X4 + X2 X3 + X3 X4 - ( A1 A4 + A1 A3 + A3 A4 )),
( X1 X3 X4 - ( A1 A3 A4 ))
)

DILWR(P,V).

(* computations *)
Q:=DIPGB(P,1).
DILWR(Q,V).
```

The output is as follows.

```
ANS: (6 (4 0) ((11 1) (11 2) (11 3) (11 4)))
MAS: RF(A1,A2,A3,A4) (* Rational Function *)
ANS: ((57 1) (57 2) (57 3) (57 4))
ANS: ((13) (47) (49) (61) (41) (55))

(  X4 -(  A4 - A2 ) )
(  X4 + X3 + X2 + X1 -(  A4 + A3 + A1 ) )
(  X3 X4 + X1 X4 + X2 X3 + X1 X3 -(  A3 A4 + A1 A4 + A1 A3 ) )
(  X1 X3 X4 -  A1 A3 A4  )

1950 ms,  389 cells,
1 crit3,  1 crit4,  1 spoly,  1 hpoly,
6 pairs,  5 restp,  0.833+ quot.

H=(  X3 + X2 + X1 -(  A3 + A2 + A1 ) )

...

14584 ms,  63521 cells,
24 crit3,  6 crit4,  1 spoly,  1 hpoly,
24 pairs,  0 restp,  0.000 quot.

Time: read = 0, eval = 16316, print = 117, gc = 0.
```

```
(  X1**3 -(  A3 A4 + A1 A4 + A1 A3 )/(  A4 - A2 ) X1**2 +(  A1 A3 A4**2 + A1 A3
**2 A4 + A1**2 A3 A4 )/(  A4**2 -2 A2 A4 + A2**2 ) X1 -  A1**2 A3**2 A4**2 /(
A4**3 -3 A2 A4**2 +3 A2**2 A4 - A2**3 ) )

(  X2 +(  A4**2 -2 A2 A4 + A2**2 )/  A1 A3 A4  X1**2 -(  A3 A4**2 + A1 A4**2 -
A2 A3 A4 - A1 A2 A4 - A1 A2 A3 )/  A1 A3 A4  X1 +(  A4 - A2 ) )

(  X3 -(  A4**2 -2 A2 A4 + A2**2 )/  A1 A3 A4  X1**2 +(  A3 A4**2 + A1 A4**2 -
A2 A3 A4 + A1 A3 A4 - A1 A2 A4 - A1 A2 A3 )/  A1 A3 A4  X1 -(  A4 + A3 + A1 ) )
```

```
(  X4 -(  A4 - A2 ) )

   Time: read = 17, eval = 66, print = 0, gc = 0.
```

This is the end of the examples and so the end of the section on the arbitrary domain system.

Chapter 8

Packages

In this chapter we give an overview and describe the usage of several software packages from commutative and non-commutative algebra and algebraic geometry. For more information see the books by [Becker, Weispfenning 1993] on Gröbner Bases and commutative computational algebra, by [Geddes *et al.* 1992] on algorithms for computer algebra and by [Knuth 1981] on seminumerical algorithms as well as the original papers cited below.

8.1 Gröbner Bases

Let $\mathbf{R} = \mathbf{K}[X_1, \ldots, X_n]$ be a commutative polynomial ring in $n \geq 0$ variables $X_1, \ldots, X_n$ over a field $\mathbf{K}$. Let I be an ideal of $\mathbf{R}$ (a subset of $\mathbf{R}$ closed under addition and multiplication by arbitrary elements of $\mathbf{R}$). Ideals arise in the algebraic methods used for solving systems of multivariate non-linear (but polynomial) equations.

The first important problem in computational ideal theory is the *ideal membership problem*:

> given $I \subseteq \mathbf{R}$ and $f \in \mathbf{R}$, is $f \in I$?

It is known by Hilbert's basis theorem, that any ideal I in a polynomial ring over a field is finitely generated. So let I be generated by $m \in \mathbf{N}$ polynomals $p_1, \ldots, p_m \in \mathbf{R}$, notation $I = \text{ideal}(p_1, \ldots, p_m)$. Then $f \in I$ means, that there exist polynomials $h_1, \ldots, h_m \in \mathbf{R}$, such that

$$f = \sum_{i=1}^{m} h_i p_i.$$

Then $f \in I$ can be decided, iff one can find the polynomials $h_1, \ldots, h_m$. In case $\mathbf{R}$ is a univariate polynomial ring or in case the generating polynomials of the ideal are all linear, the solution to this problem has been known since several hundered years and the methods are named after Euclid and Gauss. In the general case the solution method is named after Gröbner and an algorithm by Buchberger. The solutions are as follows:

1. *univariate case*: $\mathbf{R} = \mathbf{K}[X]$ is a so called euclidean domain, i.e. there exists a division with remainder and as a consequence every ideal is generated by exactly one element. This element p can be computed by means of the euclidean algorithm, then $I =$

ideal$(p_1, \ldots, p_m)$ = ideal(p). Then $f \in I$ iff $f = hp$ for some $h \in \mathbf{R}$, i.e. $f \in I$ iff p divides f, which solves the problem.

2. *linear case*: If f is linear and the $p_1, \ldots, p_m$ are linear too, then by gaussian elimination it can be checked if f is a $\mathbf{K}$-linear combination of the $p_1, \ldots, p_m$. So the problem is again solved.

3. *general case*: The method is in some sense similar to the above two:

   first transform the generating set for the ideal to a suitable form, called Gröbner base, by Buchberger's algorithm,

   second check if f can be written as polynomial combination of the elements from the Gröbner base.

   The second step can be computed by a form of multi-polynomial division with remainder called polynomial reduction and if f reduces to zero, then $f \in I$. The difficulty is that for arbitrary generating sets of ideals the polynomial reduction does in general not reduce a polynomial in the ideal to zero. Only for reduction with respect to a Gröbner base the polynomial reduces to zero if it is in the ideal.

Once the ideal membership problem can be solved, many other problems in ideal theory can be solved by computational methods. To define Gröbner bases more precisely we need some preparations about reduction relations.

Let $\longrightarrow$ be a relation on R, (i.e. $\longrightarrow \subseteq R \times R$) then $\longrightarrow$ is called a *reduction relation* if it is a strictly anti-symmetric relation. An element $a \in R$ is called *irreducible* if for all $b \in R$ $(a, b) \notin \longrightarrow$. A reduction relation $\longrightarrow$ is called *Noetherian* if there does not exist an infinite reduction sequence.

Let $\longrightarrow$ be a reduction relation on R, then $\longrightarrow^*$ denotes the reflexive, transitive closure of $\longrightarrow$; $\longleftrightarrow^*$ denotes the reflexive, transitive closure of the symmetric closure of $\longrightarrow$; for $f, g, h \in R$ $f \downarrow g$ denotes that f and g reduce to a common element, that is there exists h with $f \longrightarrow^* h$ and $g \longrightarrow^* h$. In this case $f \longleftrightarrow^* g$.

Reduction relations are in general not confluent; this means that two different reductions of the same element may lead to two different irreducible elements. Let $\longrightarrow$ be a reduction relation on R, $a, b, c, d \in R$. Then $\longrightarrow$ is *confluent* if $a \longrightarrow^* c$ and $a \longrightarrow^* d$ implies $c \downarrow d$; $\longrightarrow$ is *locally confluent* if $a \longrightarrow c$ and $a \longrightarrow d$ implies $c \downarrow d$; $\longrightarrow$ has the *Church-Rosser property* if $a \longleftrightarrow^* b$ implies $a \downarrow b$; $\longrightarrow$ has *unique normal forms* if $a \longrightarrow^* b$, $a \longrightarrow^* c$ and b, c are irreducible implies $b = c$.

Now Newmann's lemma tells us that for a *Noetherian* reduction relation $\longrightarrow$ on R, $\longrightarrow$ is confluent iff $\longrightarrow$ is locally confluent iff $\longrightarrow$ has unique normalforms.

For polynomial ring $R = \mathbf{K}[X_1, \ldots, X_n]$, with respect to an admissible ordering $<$, over a field $\mathbf{K}$ a reduction relation can be defined as follows. Recall that a linear order $<$ on the set of terms $T = T(X_1, \ldots, X_n)$ is called admissible if

1. $1 < t$ for all $t \in T$ and

2. $u < v$ implies that $ut < vt$ for all $u, v, t \in T$.

Recall that HT(f) denotes the highest term of f with respect to the given term order $<$ and that $HC(f)$ denote the coefficient of HT(f) in f. Let $p \in R$, $t \in T$ and let $f, f' \in R$,

$t \in T(f)$ then

$$f \longrightarrow_{t,p} f'$$

iff there exists $u \in T$ such that $t = u\mathrm{HT}(p)$ and

$$f' = f - aup,$$

where $a \in \mathbf{K} \setminus \{0\}$ is the unique element of $\mathbf{K}$ such that $HC(f) = aHC(up)$. Furthermore we define $f \longrightarrow_p f'$ if $f \longrightarrow_{t,p} f'$ for some t in $T(f)$, and for a set P of polynomials we define $f \longrightarrow_P f'$ if for some $p \in P$, $f \longrightarrow_p f'$. By Dicksons lemma it can be shown, that the reduction relation $\longrightarrow_P$ is Noetherian for any $P \subset R$.

Let $R = \mathbf{K}[X_1, \ldots, X_n]$ be a polynomial ring, with respect to an admissible ordering, over a field $\mathbf{K}$. Let $G \subset R$ be a finite subset of polynomials of R. If the reduction relation $\longrightarrow_G$ is confluent, then G is called a *Gröbner base*.

For the computation of a Gröbner base in finitely many steps we need one more definition. For $f, g \in R$, the S-polynomial of f and g is defined as

$$SP(f, g) = auf - bvg,$$

where $u, v \in T(X_1, \ldots, X_n)$ are defined such that $u\mathrm{HT}(f) = v\mathrm{HT}(g) = w$ with $w = lcm(\mathrm{HT}(f), \mathrm{HT}(g))$ and $a, b \in \mathbf{K}$ are defined such that $HC(auf) = HC(bvg)$.

Theorem 8.1.1 *Let $R = \mathbf{K}[X_1, \ldots, X_n]$ be a polynomial ring, with respect to an admissible ordering, over a field $\mathbf{K}$. Let $G \subset R$ be a finite subset of polynomials of R. Then the following assertions are equivalent.*

1. *G is a Gröbner base.*
2. *For all $f \in \mathrm{ideal}(G)$, $f \longrightarrow_G^* 0$.*
3. *For all $h \in H = \{SP(f, g) \ : \ f, g \in G, f \neq g\}$, $h \longrightarrow_G^* 0$.*

Observe, that 3) is a finite condition whether an ideal base G is a Gröbner base or not. And the main idea of the Buchberger algorithm is, in case the test for a Gröbner base fails, to adjoin a $\longrightarrow_G$ normalform of the S-polynomial to the base G and to repeat the test. By a theorem we know, that for any finite $F \subset R$ of polynomials this repeated adjunction of normalforms of S-polynomials must eventualy terminate, so one can construct a Gröbner base G of ideal(F). The proof of this theorem presents the Buchberger algorithm for constructing Gröbner bases. For a complete treatment of the theory of Gröbner bases see [Becker, Weispfenning 1993].

8.1.1 Algorithms and Examples

The algorithms for the computation of Gröbner bases are taken from the 'The Buchberger Algorithm System', BAS as developed by [Gebauer, Kredel 1983]. Their implementation follows [Winkler *et al.* 1985] and contains various improvments to computing strategy and efficiency.

The package contains also several subroutines which are of independent interest: S-polynomials, polynomial reduction and normalforms, irreducible sets, minimal Gröbner

bases, optimization of termorderings, augmented computation of Gröbner bases, test for zero-dimensionality of ideals, construction of univariate polynomials of minimal degree in the ideal and others. There are also several packages which are buildt on top of the Gröbner base package or use it for precomputations: ideal dimension, zero-dimensional ideal decomposition, zero-dimensional ideal real root finding, syzygies and systems of linear polynomial equations and others.

The programs are mostly available for polynomials with rational, integer and arbitrary domain coefficients. The programs are contained in the Modula-2 libraries `DIPRNGB`, `DIPZ`, `DIPTOO`, `DIPIGB` and `DIPGB`.

Examples for Gröbner base computations have been given in several sections (e.g. section 7.4.3 or the next section).

8.2 Ideal dimension

We quote from the introduction of [Kredel, Weispfenning 1988] who developed the package. For a complete treatment of the theory and the extension of the method to arbitrary degree compatible term orders see [Becker, Weispfenning 1993].

Among the basic problems of the algorithmic theory of polynomial ideals, the computation of the dimension $dim(I)$ of an ideal I in a polynomial ring $\mathbf{R} = \mathbf{K}[X_1, \ldots, X_n]$ occupies a prominent place. The geometric definition of $dim(I)$ as the maximal dimension of all isolated prime ideals J associated with I is unfavourable for computation, since it involves the primary decomposition of I. Instead, $dim(I)$ can be described more directly as the largest number of elements in $\mathbf{R}$ that are independent modulo I in a natural sense (see e.g. [Gröbner 1968/70])

$$dim(I) = max\{\ |U|\ :\ U \subset \{X_1, \ldots, X_n\}, I \cap \mathbf{K}[U] = \{0\}\ \}.$$

A third approach characterizes $dim(I)$ as the degree of the Hilbert polynomial of I, that can be computed from the vector space dimension of the $\mathbf{K}$–linear spaces $S_m = \{f+I \,|\, deg(f) \leq m\} \subseteq \mathbf{R}/I$.

This last characterization has been combined successfully with the algorithmic technique of Gröbner bases introduced by [Buchberger 1965], see also [Möller, Mora 1983]. The resulting algorithms are applicable to non-trivial cases. The special problem to test whether $dim(I) \leq 0$ can be handled more easily by Buchberger's criterion (Method 6.9 in [Buchberger 1985]).

The Gröbner basis method can also be combined with the second characterization of $dim(I)$: Consider all pure lexicographical orderings $<_L$ of terms in $\mathbf{R}$ induced by permutations of the variables X_i. Compute a Gröbner base $G = G(<_L)$ of I with respect to $<_L$, and let $S = S(<_L)$ be the largest initial segment of the set of variables, such that no head term of a polynomial in G contains only variables from S. Then each S is independent modulo I and the largest S determines the dimension of I. (see [Kandri-Rody 1985], compare also Lemma 5 in [Kutzler, Stifter 1986]). The advantage of this method is that it determines besides $dim(I)$ also sets of variables independent modulo I. On the other hand, the number of Gröbner basis calculations involved in this method renders it useless for practical purposes in most cases. Further related methods for computing the dimension of a polynomial ideal appear in [Giusti 1984] and [Carra-Ferro 1986]. The application of resultant calculus to this problem has been described in [Kredel 1985].

The main algorithm of this package computes both the dimension of I and maximal independent sets of variables modulo I from a single Gröbner basis of I. And the algorithm is verified to be correct in the case of a pure lexicographical term order and a few other cases. A recent theorem of [Carra-Ferro 1987] implies that the algorithm is correct for an arbitrary admissible term order. The algorithm is significantly faster than the algorithms in [Möller, Mora 1983] using the Hilbert polynomial. It also provides considerable additional information on independent sets and dimensions of isolated prime ideals associated with I. Thus, it is particulary suitable to the method of geometrical theorem proving developed in [Kutzler, Stifter 1986]. It is also useful as a tool in the primary decomposition of I (compare [Gianni *et al.* 1986] and also [Kredel 1987]). The method provides parametrizations of affine algebraic sets. Thus it may be helpful in determining the spatial structure of molecules from algebraic equations describing this structure.

The verification of the algorithm employs the novel notion of strong independence modulo an ideal I, a concept that correlates well with Gröbner bases and may be of independent interest. The algorithm was first implemented in the ALDES/SAC-2 system by [Collins, Loos 1980] and has been tested successfully on substantial examples, including those studied in [Böge *et al.* 1986].

8.2.1 Algorithms and Examples

The specification of the main algorithm `DILDIM` is as follows. Given a Gröbner base G of multivariate polynomials with field coefficients, find the following

$d =$ dimension ideal(G),

$S =$ greatest maximal independent set of variables,

$M =$ set of all maximal independent sets.

The method is to compute strong independent sets of variables for the head terms of the polynomials in the Gröbner base as discussed in [Kredel, Weispfenning 1988].

The computed values are the dimension and strongly indepenedent sets in the case of the inverse lexicographical term order and in case of any total degree order. In case of a different term ordering the computed values are surely upper bounds for the dimension and independent sets. It is however claimed by [Carra-Ferro 1987] that for all admissible term orders the dimension and independent sets are found in this way. If G is not a Gröbner base the computed values are also upper bounds.

The computation is valid for all coefficient domains since it depends only on the headterms of the polynomials. The programs are contained in the Modula-2 library `DIPDIM`.

In the following example we compute first a Gröbner base of an ideal I and then we determine the dimension of the ideal. The input consists of a polynomial list in the usual syntax as described in section 7.4.3 and then of the two statements '`Q:=DIRPGB(P,1)`' and '`DIMENSION(Q)`' which request the computation of the Gröbner base and the dimension respectively. The later program calls the '`DILDIM`' routine and then prints the results to the current output stream.

```
P:=PREAD().

(A1,A2,A3,A4,X1,X2,X3,X4) L
```

```
(
( X4 - ( A4 - A2 )),
( X1 + X2 + X3 + X4 - ( A1 + A3 + A4 )),
( X1 X3 + X1 X4 + X2 X3 + X3 X4 - ( A1 A4 + A1 A3 + A3 A4 )),
( X1 X3 X4 - ( A1 A3 A4 ))
)

PWRITE(P).
Q:=DIRPGB(P,1).
PWRITE(Q).

CLOUT("Computing Dimension ... ").
DIMENSION(Q).
```

The following output shows first the computed Gröbner base and then the output of the dimension algorithm. The dimension turns out to be 4 and a maximaly independent set of variables is '`(A1,A2,A3,A4)`'. There is another maximal independent set of variables '`(A1,A2,A3,X1)`' as shown in the list of all maximal independent sets. The computing time for the computation of the dimension is neglectable.

```
MAS: Polynomial in the variables: (A1,A2,A3,A4,X1,X2,X3,X4)

Term ordering:  inverse lexicographical.

Polynomial list:
          (  X4 - A4 + A2 )
          (  X4 + X3 + X2 + X1 - A4 - A3 - A1 )
          (  X3 X4 + X1 X4 + X2 X3 + X1 X3 - A3 A4 - A1 A4 -
           A1 A3 )
          (  X1 X3 X4 - A1 A3 A4 )

Time: read = 0, eval = 1667, print = 167, gc = 0.
MAS: Polynomial in the variables: (A1,A2,A3,A4,X1,X2,X3,X4)

Term ordering:  inverse lexicographical.

Polynomial list:
          (  A4**3 X1**3 -3 A2 A4**2 X1**3 +3 A2**2 A4 X1**3
           - A2**3 X1**3 - A3 A4**3 X1**2 - A1 A4**3 X1**2 +
          2 A2 A3 A4**2 X1**2 - A1 A3 A4**2 X1**2 +2 A1 A2 A
          4**2 X1**2 - A2**2 A3 A4 X1**2 +2 A1 A2 A3 A4 X1**
          2 - A1 A2**2 A4 X1**2 - A1 A2**2 A3 X1**2 + A1 A3
          A4**3 X1 + A1 A3**2 A4**2 X1 - A1 A2 A3 A4**2 X1 +
           A1**2 A3 A4**2 X1 - A1 A2 A3**2 A4 X1 - A1**2 A2
          A3 A4 X1 - A1**2 A3**2 A4**2 )

          (  A1 A3 A4 X2 + A4**2 X1**2 -2 A2 A4 X1**2 + A2**
          2 X1**2 - A3 A4**2 X1 - A1 A4**2 X1 + A2 A3 A4 X1
          + A1 A2 A4 X1 + A1 A2 A3 X1 + A1 A3 A4**2 - A1 A2
          A3 A4 )

          (  A1 A2 A3 X1 X2 + A4**2 X1**3 -2 A2 A4 X1**3 + A
          2**2 X1**3 - A3 A4**2 X1**2 - A1 A4**2 X1**2 + A2
          A3 A4 X1**2 - A1 A3 A4 X1**2 + A1 A2 A4 X1**2 +2 A
          1 A2 A3 X1**2 + A1 A3 A4**2 X1 + A1 A3**2 A4 X1 +
          A1**2 A3 A4 X1 - A1 A2 A3**2 X1 - A1 A2**2 A3 X1 -
           A1**2 A2 A3 X1 - A1**2 A3**2 A4 )
```

```
            (  A4 X1 X2 - A2 X1 X2 + A4 X1**2 - A2 X1**2 - A3
            A4 X1 - A2 A4 X1 - A1 A4 X1 + A2 A3 X1 + A2**2 X1
            + A1 A2 X1 + A1 A3 A4 )

            (  X2**2 +2 X1 X2 + A4 X2 - A3 X2 -2 A2 X2 - A1 X2
             + X1**2 - A3 X1 - A2 X1 - A1 X1 - A2 A4 + A2 A3 +
             A1 A3 + A2**2 + A1 A2 )

            (  X3 + X2 + X1 - A3 - A2 - A1 )

            (  X4 - A4 + A2 )

MAS: Computing Dimension ...

Dimension = 4

Maximal independent set = (A1,A2,A3,A4)

All maximal independent sets = ( (A1,A2,A3,A4), (A1,A2,A3,X1) )

Time: read = 0, eval = 150, print = 0, gc = 0.
```

This is the end of the dimension example.

8.3 Zero-dimension ideal decomposition

We quote from the introduction of the package by [Kredel 1987]. For a complete treatment of the theory including the positive-dimension case see [Becker, Weispfenning 1993].

In a polynomial ring, a Noetherian Ring, it is known, that every ideal A can be decomposed into finitely many primary components q_i:

$$A = q_1 \cap \ldots \cap q_l$$

This fact was discovered by Lasker, Macaulay [Lasker 1905, Macaulay 1916], and in the special case of zero dimensional ideals already by M. Noether. By E. Noether this fact is true for every ring satisfying the so called 'Teilerkettensatz' which is equivalent to Hilberts Basis Theorem [Noether 1921].

In 1926 G. Hermann made an attempt to discribe a constructive way to find the primary decomposition of a given polynomial ideal over fields which allow constructive factorization. However she was not aware of the fact, that in general not for all such ground fields, the polynomials can be factorized constructively. Seidenberg gave an investigation of these topics [Hermann 1926, Seidenberg 1974].

The early constructive approaches were based on the method of solving linear equations in a module over the polynomial ring. Today we are equipped with the powerful method of Gröbner Bases, as investigated by B. Buchberger [Buchberger 1965, Buchberger 1985].

In 1976 R. Schrader proposed a way for zero dimensional polynomial ideals by constructing univariate polynomials, factorization and computing the primary ideals from the prime ideals by $q = A + p^\sigma$, where σ is the exponent of p [Schrader 1976]. However his primality test and the generation of prime ideals is hard to realize.

D. Lazard gave an algorithm to compute prime components and multiplicites for pure equidimensional ideals. In 1985 he published a careful study of the primary decomposition

in the case on a polynomial ring in two variables [Lazard 1982, Lazard 1985]. Kandri-Rody studied constructive ways to compute the radical of an ideal via the computation of associate isolated prime ideals [Kandri-Rody 1984].

P. Gianni, B. Trager & G. Zacharias developed a set of algorithms for primary decomposition in 1984. Their method uses ideals in 'generic position' and factorization, instead of field extensions by adjoining new variables. However they use a very complicated primality test, involving mappings to polynomial rings over field extensions of the ground field. [Gianni *et al.* 1986]

In our method we compute the prime ideals by first decomposing the ideal according to Schrader. An algorithm has been published by Böge, Gebauer and Kredel [Schrader 1976, Böge *et al.* 1986]. Next we transform the generated ideals by some algebraic extension of the ground field until we reach a prime basis for the ideals. This gives us an easy primality test, and the primary ideals can now be determined by $q = A + p^{\sigma}$.

At the moment the algorithms are implemented for the rational numbers as ground field, but the method applies also to the rational functions as ground field. Since field extensions and squarefree decompositions are involved, the method is restricted to fields of characteristic zero.

The method uses:

- Gröbner base calculations
- construction of univariate polynomials and univariate polynomial factorization
- algebraic field extensions of the ground field
- an easy to handle ideal primality test

The method produces for a given zero dimensional ideal:

- the associated prime ideals
- the primary ideals
- the exponents of the associated prime ideals

An informal overview of all steps of the algorithm is given in table 8.1.

8.3.1 Algorithms and Examples

The specification of the main algorithm `DECOMPO` is as follows. Given a Gröbner base of multivariate polynomials with rational number coefficients of a zero–dimensional ideal. Then find the primary ideal decomposition over algebraic field extensions of **Q**. This problem is solved using the method of P-ideal decomposition, primary ideal computation by M. Noethers method as described in [Kredel 1987].

The primary ideal decomposition is computed only over the coefficient domains of the rational numbers. The term ordering must be inverse lexicographical. There is a lot of run time information printed.

The specification of the main algorithm `DECOMPOA` is as follows. Given a Gröbner base of multivariate polynomials with rational number coefficients of a zero–dimensional ideal.

Input: F = Basis for an ideal $A \subset \mathbf{K}[X_1, \ldots, X_r]$.

Output: All tuples (P, Q, e) such that $ideal(P)$ is an associated prime ideal of A with primary ideal $ideal(Q)$ and exponent e.

Step 0. Compute a Gröbner base G for A. **if** dimension $ideal(G) \neq 0$ **then stop**.

Step 1. Compute a set of so called U–bases $\{U_1, \ldots, U_l\}, l \geq 1$, such that

$$radical(ideal(G)) = \bigcap_{i=1,\ldots,l} radical(ideal(U_i))$$

and all univariate polynomials of minimal degree in $ideal(U_i)$ are irreducible, using univariate polynomial construction (for details see [Böge *et al.* 1986]).

Step 2. Compute a set of P–bases $\{P'_1, \ldots, P'_m\}, m \geq l$, such that all polynomials in P'_i are at most bivariate with linear head term except one univariate polynomial with (possibly) non linear head term using primitive element construction. Compute ideal bases P_i by

$$ideal(P_i) = ideal(P'_i) \cap \mathbf{K}[X_1, \ldots, X_r].$$

Then $ideal(P_i)$ is a prime ideal associated to A.

Step 3. Compute minimal $e \in \mathbf{N}$, such that $ideal(P^e) \subset A + ideal(P^{e+1})$. Then e is the exponent of $ideal(Q)$ and $ideal(Q) = A + ideal(P^e)$ is a primary component of A.

Table 8.1: Overview of primary ideal docomposition

Find the primary ideal decomposition over **Q**. This problem is solved using the method of P-ideal decomposition over algebraic extension fields over **Q**. Back transformation of the prime ideals to prime ideals over **Q** by Gröbner bases computation with a different term ordering and primary ideal computation by M. Noethers method. This method is also described in [Kredel 1987].

The primary ideal decomposition is computed only over the coefficient domains of the rational numbers. The term ordering must be inverse lexicographical. There is a lot of run time information printed. The programs are contained in the Modula-2 libraries `DIPIDEAL` and `DIPDEC0`.

In the following example we compute the decomposition of the ideal generated by $(x^2 - 2, y^2 - 2, (z - xy)^2)$ over **Q** and over $\mathbf{Q}(\alpha)$ for some $\alpha \notin \mathbf{Q}$. The input consists of a polynomial list in the usual syntax as described in section 7.4.3. Then the statement '`q:=DIRPGB(p,1)`' requests the computation of the Gröbner base. Next the statements '`DECOMP0(q)`' and '`DECOMP0A(q)`' request the primary decomposition of the ideal over **Q** respectively over $\mathbf{Q}(\alpha)$ and write the results on the actual output stream.

```
(* Ideal decomposition of zero-dimensional polynomial ideals. *)

(* Polynomial list: *)
p:=PREAD().
(x,y,z) L
(
( ( z**2 - y x)**2  ),
( y**2 - 2 ),
( x**2 - 2 ),
)
PWRITE(p).

(* Groebner base: *)
q:=DIRPGB(p,0).
PWRITE(q).

(* decomposition : *)
DECOMP0(q).

(* decomposition over Q(alpha) : *)
DECOMP0A(q).
```

The following output shows the computed Gröbner base.

```
MAS: Polynomial in the variables: (x,y,z)

Term ordering:  inverse lexicographical.

Polynomial list:
          (  z**4 -2 x y z**2 + x**2 y**2 )
          (  y**2 -2 )
          (  x**2 -2 )

Time: read = 17, eval = 17, print = 0, gc = 0.
MAS: Polynomial in the variables: (x,y,z)

Term ordering:  inverse lexicographical.

Polynomial list:
          (  x**2 -2 )
```

```
          (  y**2 -2 )
          (  z**4 -2 x y z**2 +4 )
```

The following output shows the computed primary decomposition over $\mathbf{Q}$. There are three primary components, which are listed in sequence. The information on each primary component consists at first of the corresponding prime ideal, at second of the primary ideal itself and at last of the exponent of the prime ideal. The computing time is slightly more than 21 seconds. The intermediate output in not shown, except the string '`Introduction of the new variable exz`', which indicates, that a field extension was required to make the variables `x` and `z` linear in terms of the new variable `exz`. After a prime base over this extension field has been computed, the new variable is removed using ideal intersections. In the next example this variable and the corresponding polynomials are kept.

```
Introduction of the new variable exz

Introduction of the new variable exz

The given ideal
          (  x**2 -2 )
          (  y**2 -2 )
          (  z**4 -2 x y z**2 +4 )

The decomposition of the ideal

The prime ideal
          (  x**2 -2 )
          (  y - x )
          (  z + x )

The primary ideal
          (  x**2 -2 )
          (  y - x )
          (  z**2 +2 x z +2 )

The exponent is 2

The prime ideal
          (  x**2 -2 )
          (  y - x )
          (  z - x )

The primary ideal
          (  x**2 -2 )
          (  y - x )
          (  z**2 -2 x z +2 )

The exponent is 2

The prime ideal
          (  x**2 -2 )
          (  y + x )
          (  z**2 +2 )

The primary ideal
          (  x**2 -2 )
          (  y + x )
          (  z**4 +4 z**2 +4 )
```

```
The exponent is 2

Time: read = 0, eval = 21400, print = 0, gc = 0.
```

The following output shows the computed primary decomposition over $\mathbf{Q}(\alpha)$. As before there are three primary components, which are listed in sequence. The information on each primary component consists at first of the corresponding prime ideal, at second of the primary ideal itself and at last of the exponent of the prime ideal. The computing time is nearly 28 seconds. The intermediate output in not shown, except the string '`Introduction of the new variable exz`', which indicates, that a field extension was required to make the variables `x` and `z` linear in terms of the new variable `exz`. Contrary to the previous example, this new variable and the corresponding polynomials are kept after a prime base over this extension field has been computed. So the displayed bases of the prime ideals are in fact prime bases, e.g. they consist of exactly one irreducible univariate polynomial and all other polynomials are bivariate and linear in the second variable. The computing time is slightly more than in the previous example mostly despite of the computation of the primary ideals and exponents in a polynomial ring with 4 instead of 3 variables. But there are also examples where the computation over $\mathbf{Q}(\alpha)$ is faster than the computation over $\mathbf{Q}$ despite of the fact that no ideal intersections need to be computed.

```
Introduction of the new variable exz

Introduction of the new variable exz

The given ideal
          (  exz**2 -2 )
          (  x**2 -2 )
          (  y**4 -2 exz x y**2 +4 )

The decomposition of the ideal

The prime ideal
          (  exz**4 +16 )
          (  x -1/8 exz**3 +1/2 exz )
          (  y +1/8 exz**3 -1/2 exz )
          (  z -1/8 exz**3 -1/2 exz )

The primary ideal
          (  exz**4 +16 )
          (  x -1/8 exz**3 +1/2 exz )
          (  y +1/8 exz**3 -1/2 exz )
          (  z**2 -1/4 exz**3 z - exz z -2 )

The exponent is 2

The prime ideal
          (  exz**2 -18 )
          (  x +1/3 exz )
          (  y +1/3 exz )
          (  z -1/3 exz )

The primary ideal
          (  exz**2 -18 )
          (  x +1/3 exz )
          (  y +1/3 exz )
          (  z**2 -2/3 exz z +2 )
```

```
The exponent is 2

The prime ideal
          (  exz**2 -2 )
          (  x + exz )
          (  y + exz )
          (  z + exz )

The primary ideal
          (  exz**2 -2 )
          (  x + exz )
          (  y + exz )
          (  z**2 +2 exz z +2 )

The exponent is 2

Time: read = 17, eval = 27933, print = 0, gc = 0.
```

In the next example, taken from [Weinberger *et al.* 1976], we want to factor a polynomial over an algebraic number field. Precisely, the polynomial $X^3 - 3$ should be factored in $\mathbf{Q}(\alpha)$ where α is a root of $T^6 + 3T^5 + 6T^4 + T^3 - 3T^2 + 12T + 16$. Therefore we consider the ideal generated by these two polynomials in $\mathbf{Q}[T, X]$.

```
(
( X**3 - 3 ),
( T**6 + 3 T**5 + 6 T**4 + T**3 - 3 T**2  + 12 T + 16 )
)
```

The given ideal base is already a Gröbner Base, and the univariate polynomials are irreducible. One field extensions is necessary to get linear headterms in 'T' and 'X' (denoted by 'ETX'). The field extension step yields a polynomial of degree 18, which can be factored into 3 parts.

```
( ETX**18 -18 ETX**17 +180 ETX**16 -1122 ETX**15
 + 4734 ETX**14 -13860 ETX**13 +37887 ETX**12
 - 163908 ETX**11 +608328 ETX**10 -680564 ETX**9
 - 5061348 ETX**8 +26631000 ETX**7 -46572249 ETX**6
 - 9583146 ETX**5 +231202116 ETX**4 -612614634 ETX**3
 + 1302509286 ETX**2 -2140997364 ETX +1595283481 )
```

From its factors we obtain 3 extension ideals:

```
Extension ideal no 1

( ETX**6 -6 ETX**5 +24 ETX**4 -50 ETX**3 +78 ETX**2
  -132 ETX +121 )
( T -4/165 ETX**5 +37/330 ETX**4 -74/165 ETX**3 +
  112/165 ETX**2 -151/330 ETX +22/15 )
( X -8/165 ETX**5 +37/165 ETX**4 -148/165 ETX**3
  +224/165 ETX**2 -316/165 ETX +44/15 )

Extension ideal no 2

( ETX**6 -6 ETX**5 +24 ETX**4 +4 ETX**3 -570 ETX**2
  +516 ETX +3631 )
( T +8/9345 ETX**5 -23/3738 ETX**4 +302/9345 ETX**3
```

```
  -716/9345 ETX**2 +2593/18690 ETX +6802/9345 )
( X +16/9345 ETX**5 -23/1869 ETX**4 +604/9345 ETX**3
  -1432/9345 ETX**2 -6752/9345 ETX +13604/9345 )

Extension ideal no 3

( ETX**6 -6 ETX**5 +24 ETX**4 +4 ETX**3 +402 ETX**2
 -1428 ETX +3631 )
( T -8/16863 ETX**5 +3/1022 ETX**4 -6/803 ETX**3
   +92/16863 ETX**2 +323/1606 ETX +4926/5621 )
( X -16/16863 ETX**5 +3/511 ETX**4 -12/803 ETX**3
  +184/16863 ETX**2 -480/803 ETX +9852/5621 )
```

From the prime ideals over the field extension over **Q** we compute the prime ideals over **Q**. All Gröbner bases are containing the polynomial:

```
( T**6 +3 T**5 +6 T**4 + T**3 -3 T**2 +12 T +16 )

Prime ideal no 1

( X -1/12 T**5 -1/12 T**4 -1/6 T**3 +7/12 T**2 -11/12 T -4/3 )

Prime ideal no 2

( X -1/12 T**5 -1/4 T**4 -1/2 T**3 -5/12 T**2 +1/4 T -1 )

Prime ideal no 3

( X +1/6 T**5 +1/3 T**4 +2/3 T**3 -1/6 T**2 +2/3 T +7/3 )
```

The computing time on an IBM 3090 was 52090 ms.

This is the end of the primary decomposition examples.

8.4 Zero-dimension ideal real roots

We quote from the introduction of the package developed by [Kredel 1989]. For a complete treatment of the theory see [Becker, Weispfenning 1993].

One of the application problems of computational mathematics is the determination of the solutions (roots, zeros) of systems of algebraic equations. If the system of algebraic equations generates a zero dimensional ideal in a polynomial ring over the rational numbers, then the coordinates of the roots are algebraic numbers. The problem is to find a constructive method for the computation of the **real** roots of an ideal $A = (g_1, \ldots, g_k)$ in $\mathbf{Q}[X_1, \ldots, X_r]$:

$$\begin{aligned} Zero(A) &= \{\alpha \in E^r \ : \ f(\alpha) = 0, \ \forall \ f \in A\} \\ &= \{(\omega_1, \ldots, \omega_r) \in E^r \ : \ g_i(\omega_1, \ldots, \omega_r) = 0, \ i = 1, \ldots, k\}. \end{aligned}$$

where $E \subseteq \mathbf{R}$ is a real algebraic extension field of **Q**.

This problem was fully realized during the last century and the beginning of the 20-th century. The early attempts relied mainly on the resultant calculus, and construction of so-called elimination ideals involving Kroneckers resultant systems. The computational complexity of this approach was to high to be treated satisfactorily in these times

[Kronecker 1882, Macaulay 1916]. Even published methods tend to become forgotten, so in older editions of V.d.Waerdens '(Modern) Algebra' from 1931 we find a lot about resultant calculus which parts are removed in current editions e.g. 1971.

With the rise of computers the problem was mainly attacked by numerical mathematics using iterative methods, such as homotopy methods.

But all these methods give no warranty that all solutions are found, or that a special coordinate of the solution is exactly equal to a certain algebraic number. So there is an increasing need of exact methods in theoretical investigations. We will just mention the article of Morgan which tries to describe 'exact' numerical methods [Morgan 1983], or the article of Butcher for finding the exact coefficients of some Runge-Kutta iteration equations [Butcher 1984] and so proving their existence; in theoretical economics too, there is a need of reliable mathematical methods.

One of the most famous approaches to solve this problem by computer has been published by Collins and Loos based on resultant calculus and has been applied to quantifier elimination over real closed fields [Collins 1974, Collins, Loos 1982, Loos 1982]. The method uses projection sets (a special set of generators for elimination ideals in this case), primitive element construction, real root isolation of algebraic number field polynomials and univariate polynomial construction. It inductively reduces the problem to the bivariate case by the mappings $\mathbf{Q}[X_1,\ldots,X_r] \longrightarrow \mathbf{Q}[X_1,\ldots,X_i] \longrightarrow \mathbf{Q}(\beta_1,\ldots,\beta_{i-1})[X_i] \longrightarrow \mathbf{Q}(\alpha_{i-1})[X_i]$ and then solving the bivariate case.

One of the first attempts to apply Gröbner Bases to this problem has been undertaken by Buchberger and Trinks, see Methods 6.10 and 6.12 in [Buchberger 1985]. Pohst and Yun combined the computation of resultants and pseudo division with Gröbner bases for the computation of elimination ideals [Pohst, Yun 1981]. The next step towards the solution of this problem was the application of univariate polynomial construction and ideal decomposition yielding a finite inclusion for the zeros of the ideal [Böge *et al.* 1986]. Gianni discussed a method also based on Gröbner bases; it seems however to be impractical to construct the desired Gröbner bases due to the posibly large degrees for the polynomials generated (their bases are defined like P-bases, but with one squarefree polynomial instead of an irreducible polynomial) [Gianni 1986]. Gianni and Kalkbrenner gave also methods using specialization in algebraic field extensions and gcd-computations in algebraic field extensions, respectively [Gianni 1987, Kalkbrenner 1987].

Our approach continues the one described by Böge *et al.* and solves the most important open problem left over: the selection of the 'valid' solutions. We use following (sub)methods:

- Gröbner base calculations
- construction of univariate polynomials
- univariate polynomial factorization
- primitive element computation using Gröbner bases
- univariate polynomial real root isolation
- algebraic number field element sign

Using these tools, our method produces for a given zero dimensional ideal A:

- a decomposition of A into radical ideals
- all tuples of coordinates for the real roots of A

An informal overview of all steps of the algorithm is given in table 8.2.

Input: F = Basis for an ideal $A \subset \mathbf{K}[X_1, \ldots, X_r]$.

Output: All tuples of minimal polynomials and isolating intervals for the real roots of A.

Step 0. Compute a Gröbner base G for A. **if** dimension $ideal(G) \neq 0$ **then stop**.

Step 1. Compute a set of so called U–bases $\{U_1, \ldots, U_l\}, l \geq 1$, such that

$$Zero(ideal(G)) = \bigcup_{i=1,\ldots,l} Zero(ideal(U_i))$$

and all univariate polynomials of minimal degree in $ideal(U_i)$ are irreducible, using univariate polynomial construction (for details see [Böge *et al.* 1986]).

Step 2. Compute a set of so called R–bases $\{R_1, \ldots, R_m\}, m \geq l$, such that

$$Zero(ideal(G)) = \bigcup_{i=1,\ldots,m} Zero(ideal(R_i))$$

and all polynomials in R_i are at most bivariate, using primitive element construction (for details see also [Kredel 1987]).

Step 3. Compute isolating intervals for the real roots of univariate polynomials (see [Loos 1982]), and select valid roots of the elimination ideals of $ideal(R_i)$.

Table 8.2: Overview of real root computation

8.4.1 Algorithms and Examples

The specification of the main algorithm `ROOTS` is as follows. Given a Gröbner base of *multivariate* polynomials with rational number coefficients of a zero–dimensional ideal. Find the isolating intervals of the real roots of the ideal generated be the set of polynomials. This problem is solved using the method of R-ideal decomposition, univariate polynomial real root isolation and combination of valid tuples of isolating intervals of the real roots. The method is described in [Kredel 1989].

The real roots are computed only over the coefficient domains of the rational numbers. The term ordering must be inverse lexicographical. There is a lot of run time information printed. The isolating intervals will be refined to any desired precision specified in the second parameter. The programs are contained in the Modula-2 libraries `DIPIDEAL`, `DIPDECO` and `DIPROOT`.

In the following example we compute the real roots of the ideal generated by $(x^2 - 2, y^2 - 3, z^2 - xy)$ to a precision of 20 decimal digits. The components of the real roots should

appear among $\pm\sqrt{2}$ for the x component, $\pm\sqrt{3}$ for the y component and $\pm\sqrt{\sqrt{2}\sqrt{3}}$ for the z component.

The input consists of a polynomial list in the usual syntax as described in section 7.4.3. Then the statement 'q:=DIRPGB(p,0)' requests the computation of the Gröbner base. Next the statement 'ROOTS(q,20)' request the computation of the isolating intervals for real roots to a precision of 20 decimal places and writes the results on the actual output stream.

```
(* Real roots of zero-dimensional polynomial ideals. *)

(* Polynomial list: *)
p:=PREAD().
(x,y,z) L
(
( z**2 - y x ),
( y**2 - 3 ),
( x**2 - 2 ),
)
PWRITE(p).

(* Groebner base: *)
q:=DIRPGB(p,0).
PWRITE(q).

(* Real roots: *)
ROOTS(q,20).
```

The following output shows the computed Gröbner base, which in fact show, that the input already was a Gröbner base.

```
MAS: Polynomial in the variables: (x,y,z)

Term ordering:  inverse lexicographical.

Polynomial list:
          (  z**2 - x y )
          (  y**2 -3 )
          (  x**2 -2 )

MAS: Polynomial in the variables: (x,y,z)

Term ordering:  inverse lexicographical.

Polynomial list:
          (  x**2 -2 )
          (  y**2 -3 )
          (  z**2 - x y )
```

The next output shows the result of step 1 in the method of table 8.2, the so called 'normalized tuples', which happens to be just a single tuple. Such a tuple consists first of the respective Gröbner base and then of the univariate polynomials of minimal degree in the ideal in each variable. The tuples are are characterized *first* by the respective ideals generated by the bases beeing radicals and their intersection beeing a decomposition of the radical of the original ideal and *second* by the fact that the univariate polynomials of minimal degree in each variable are irreducible.

```
The normalized tupels

Zero set tupel no 1

Characterising groebner basis
          (  x**2 -2 )
          (  y**2 -3 )
          (  z**2 - x y )

Characterising polynomial for the real roots is
p(x) = (  x**2 -2 )

Characterising polynomial for the real roots is
p(y) = (  y**2 -3 )

Characterising polynomial for the real roots is
p(z) = (  z**4 -6 )
```

The next output shows the result of step 2 in the method of table 8.2, the so called 'refined tuples', which happens to be just a single tuple. Such a tuple consists first of the respective Gröbner base and then of the univariate polynomials of minimal degree in the ideal in each variable. As the normalized tuples before the refined tuples are characterized *first* by the respective ideals generated by the bases beeing radicals and their intersection beeing a decomposition of the radical of the original ideal and *second* by the fact that the univariate polynomials of minimal degree in each variable are irreducible. Additionally this tuples have the *third* property, that the ideal basis polynomials are at most bivariate (so called R-bases). This property is enforced by a field extension, making the variables `x` and `y` linear with respect to the new variable `eyx` and thus removing the trivariate polynomial $z^2 - xy$ from the base. This event is indicated by the string '`Introduction of the new variable eyx`'. Note, that the ideal base is not jet a prime base (P-base) because the polynomial in the variable `z` is still not linear as it would be required by a P-base.

```
Introduction of the new variable eyx

The refined tupels

Zero set tupel no 1

Characterising groebner basis
          (  eyx**4 -10 eyx**2 +1 )
          (  x -1/2 eyx**3 +9/2 eyx )
          (  y -1/2 eyx**3 +11/2 eyx )
          (  z**2 +1/2 eyx**2 -5/2 )

Characterising polynomial for the real roots is
p(eyx) = (  eyx**4 -10 eyx**2 +1 )

Characterising polynomial for the real roots is
p(x) = (  x**2 -2 )

Characterising polynomial for the real roots is
p(y) = (  y**2 -3 )

Characterising polynomial for the real roots is
p(z) = (  z**4 -6 )
```

The next output shows the result of step 3 in the method of table 8.2, the so called 'zero set tuples', which now happen to be four tuples. Each zero set tuple characterizes one real root of the given zero-dimensional ideal. Such a tuple consists for each variable first of the univariate polynomial of minimal degree in the ideal second of the isolating interval of the real root in this component and third of the decimal approximation of the real root with the requested precision. The components introduced by the field extensions can by ignored to obtain a real root of the original ideal (without changing the number of real roots).

```
The zero set for dim 0 tupels


Zero set tupel no 1

Characterising polynomial for the real root is
p(eyx) = (  eyx**4 -10 eyx**2 +1 )
The isolating interval for the real root is
eyx = (0, 1/2)
The decimal approximation for the real root is
eyx = 0.317837245195782244726

Characterising polynomial for the real root is
p(x) = (  x**2 -2 )
The isolating interval for the real root is
x = (-2, -1)
The decimal approximation for the real root is
x = -1.414213562373095048802

Characterising polynomial for the real root is
p(y) = (  y**2 -3 )
The isolating interval for the real root is
y = (-2, -1)
The decimal approximation for the real root is
y = -1.732050807568877293527

Characterising polynomial for the real root is
p(z) = (  z**4 -6 )
The isolating interval for the real root is
z = (-2, -1)
The decimal approximation for the real root is
z = -1.565084580073287316584


Zero set tupel no 2

Characterising polynomial for the real root is
p(eyx) = (  eyx**4 -10 eyx**2 +1 )
The isolating interval for the real root is
eyx = (-1/2, -1/4)
The decimal approximation for the real root is
eyx = -0.317837245195782244726

Characterising polynomial for the real root is
p(x) = (  x**2 -2 )
The isolating interval for the real root is
x = (0, 2)
The decimal approximation for the real root is
x = 1.414213562373095048802
```

```
Characterising polynomial for the real root is
p(y) = (  y**2 -3 )
The isolating interval for the real root is
y = (0, 2)
The decimal approximation for the real root is
y = 1.732050807568877293527

Characterising polynomial for the real root is
p(z) = (  z**4 -6 )
The isolating interval for the real root is
z = (-2, -1)
The decimal approximation for the real root is
z = -1.565084580073287316584


Zero set tupel no 3

Characterising polynomial for the real root is
p(eyx) = (  eyx**4 -10 eyx**2 +1 )
The isolating interval for the real root is
eyx = (0, 1/2)
The decimal approximation for the real root is
eyx = 0.317837245195782244726

Characterising polynomial for the real root is
p(x) = (  x**2 -2 )
The isolating interval for the real root is
x = (-2, -1)
The decimal approximation for the real root is
x = -1.414213562373095048802

Characterising polynomial for the real root is
p(y) = (  y**2 -3 )
The isolating interval for the real root is
y = (-2, -1)
The decimal approximation for the real root is
y = -1.732050807568877293527

Characterising polynomial for the real root is
p(z) = (  z**4 -6 )
The isolating interval for the real root is
z = (0, 2)
The decimal approximation for the real root is
z = 1.565084580073287316584


Zero set tupel no 4

Characterising polynomial for the real root is
p(eyx) = (  eyx**4 -10 eyx**2 +1 )
The isolating interval for the real root is
eyx = (-1/2, -1/4)
The decimal approximation for the real root is
eyx = -0.317837245195782244726

Characterising polynomial for the real root is
p(x) = (  x**2 -2 )
The isolating interval for the real root is
x = (0, 2)
The decimal approximation for the real root is
x = 1.414213562373095048802
```

```
Characterising polynomial for the real root is
p(y) = (  y**2 -3 )
The isolating interval for the real root is
y = (0, 2)
The decimal approximation for the real root is
y = 1.732050807568877293527

Characterising polynomial for the real root is
p(z) = (  z**4 -6 )
The isolating interval for the real root is
z = (0, 2)
The decimal approximation for the real root is
z = 1.565084580073287316584

Time: read = 0, eval = 7116, print = 0, gc = 7734.
```

The total computing time for the real root isolation including the ideal decompositions is slightly more than 7 seconds (8 seconds in a different run). The computing time strongly depends on the requested precision as it is shown in table 8.3.

Precision in decimal digits	Computing time in milliseconds
0	4400
20	8833
50	16250
100	37566

Table 8.3: Computing Time Summary: Real Roots

This is the end of the real root isolation example.

8.5 Comprehensive Gröbner bases

The main point of comprehensive Gröbner bases is that the construction of a Gröbner base is not performed over a field, but over a ring (with parameters) such that the specialization of the parameters to elements of **any** field leads to a Gröbner base over this field. In this sense such an ideal base is a *comprehensive Gröbner base*. In general the property of being a Gröbner base is lost under specialization of the coefficients as the following example from [Weispfenning 1990] shows. Let $S = \mathbf{Q}[U][X, Y]$ be a polynomial ring in X, Y with parameter U and with $X < Y$. Let

$$F = \{X + 1, UY + X\},$$

then F is a Gröbner base in $\mathbf{Q}[U, X, Y]$ wrt. $<$. Let $\sigma : \mathbf{Q}[U] \longrightarrow \mathbf{K}$ be a specialization, which embeds $\mathbf{Q}$ into $\mathbf{K}$ and with $\sigma(U) \in \mathbf{K}$. Then $\sigma(F)$ (defined by applying σ to the coefficients) is a Gröbner base in $\mathbf{K}[X, Y]$ for any σ with $\sigma(U) \neq 0$. But for a specialization with $\sigma(U) = 0$ we have $\sigma(F) = \{X + 1, X\}$ and we see that $1 \in \text{ideal}(\sigma(F))$ but 1 is not reducible with respect to $\sigma(F)$. So $\sigma(F)$ can not be a Gröbner base. To obtain a comprehensive Gröbner base one would consider also the case when $U = 0$ and under this condition the polynomial $X + 1 - (UY + X) = -UY + 1 \in \text{ideal}(F)$. So in this example

$$G = \{X + 1, UY + X, -UY + 1\}$$

would be a comprehensive Gröbner base, since now also under the specialization $\sigma(U) = 0$ we see that $\sigma(G) = \{X + 1, X, 1\}$ is a Gröbner base.

The construction of a comprehensive Gröbner base is performed by the usual process of building S-polynomials and reductions. But now the conditions under which the steps are performed are recorded in a set of conditions. This leads first to a tree of ideal bases where the nodes are labeled by the set of conditions under which the step has been performed. This tree of ideal bases is called a Gröbner system and a comprehensive Gröbner base is afterwards obtained by taking the union of all ideal bases at the leaves of the tree. A condensed coding of the conditions applied to the coefficients of the polynomials under consideration is called a *colouring*. A coefficient is coloured *red* if it is non-zero under the current set of conditions, it is coloured *green* if it is zero under the current set of conditions, otherwise it is coloured *white*. A determined set of polynomials is a set of polynomials together with a set of conditions such that the first non-green term of a polynomial is coloured red. This term then serves as a head term during the following steps of the reduction and S-polynomial construction. Using these constructions the algorithms for the construction of Gröbner systems are developed.

8.5.1 The new implementation and interface

The comprehensive Gröbner basis package of MAS 0.7 has been changed in many respects. There is a new user interface making better use of the MAS interpreter. The old user interface is still present but will no longer be supported. New features have been added. These include: Conditions can now be evaluated by using reduced sets or Gröbner bases. Comprehensive Gröbner bases for coefficient fields of arbitrary characteristic can be computed. Computation can be restricted to the generic case. This section is intended to explain the *interactive usage* of the new comprehensive Gröbner basis package.

A first example

In case you just want to compute a comprehensive Gröbner basis and do not want to know about options you may want to change the following example input file `example.in` to suit your needs and start computation by entering `mas -f example.in`. Note that most of the blanks are mandatory (including one after `BEGIN`)!

```
BEGIN
p:=CDPREAD();
CGBOPT(LIST(0,1,0,2,0,4,4));
g:=GSYS(p);
c:=CGBFGSYS(g);
CDPWRITE(p);
CGBOPTWRITE();
GSYSWRITE(g);
CGBWRITE(c);
END.

IP (a,b,c,d) (X,Y,Z)

0
```

```
()
1
()
.

(
( ( a**2 b**2 + a b ) X Y Z + ( c**3 d ) Y + 1 )
( X + ( c**2 d**2 + b ) Y )
).

EXIT.
```

8.5.2 Features

This package allows to

- compute Gröbner systems and
- compute comprehensive Gröbner bases.

Gröbner systems can be

- written to the output stream,
- reduced and
- converted to comprehensive Gröbner bases.

Comprehensive Gröbner bases can be

- written to the output stream,
- globally reduced and
- converted to a quantifier free formula.

The computations depend on some options.

- The amount of output generated during computation can be selected.
- Factorization of coefficients can be used.
- Top reduction can be used instead of “normal” reduction.
- Conditions can be evaluated in three different ways:
  - by just comparing polynomials,
  - by using reduced sets and
  - by using Gröbner bases.

- Computations can be done in arbitrary characteristic instead of characteristic zero. Note that in this case d-reduction is used instead of reduction when evaluating conditions with reduced sets. Note that evaluating conditions using Gröbner bases makes no sense in arbitrary characteristic.
- The term order for polynomials and coefficients can be selected.
- The term order for polynomials and coefficients can be selected.
- Computation can be restricted to the generic case only.

8.5.3 Functions available through the interpreter

The following functions supplied by this package are available through the MAS interpreter:

```
PROCEDURE CDPREAD():LIST;

PROCEDURE CDPWRITE(CDP: LIST);

PROCEDURE CGBOPT(O: LIST);

PROCEDURE CGBOPTWRITE();

PROCEDURE GSYS(CDP: LIST): LIST;

PROCEDURE GSYSRED(GS: LIST): LIST;

PROCEDURE GSYSWRITE(S: LIST);

PROCEDURE CGBFGSYS(S: LIST): LIST;

PROCEDURE CGBGLOBRED(CGB: LIST): LIST;

PROCEDURE CGBWRITE(CGB: LIST);

PROCEDURE CGBQFF(CGB: LIST): LIST;
```

To compute a comprehensive Gröbner basis you first read the polynomial system and the initial case distinction with `CDPREAD`. This can be printed by `CDPWRITE`. You can change the options with `CGBOPT` and write them with `CGBOPTWRITE`. Then you can compute a Gröbner system using `GSYS`. This can be reduced with `GSYSRED` and written to the output stream with `GSYSWRITE`. The next step is computing a comprehensive Gröbner basis with `CGBFGSYS`. It can be written by `CGBWRITE` and globally reduced by `CGBGLOBRED`. The corresponding quantifier free formula can be computed by `CGBQFF`.

8.5.4 User selectable Options

All functions in this package depend on some global options.

Writing Options

The current state of the options can be printed out by

```
CGBOPTWRITE().
```

The output (if the default options are in effect) will be

```
Options for computation of Groebner systems are: ( 1,1,0,0,0,4,4,0 )
Some output during computation.
With factorization of coefficients.
Top-reduction only.
Conditions are evaluated by comparing.
Characteristic is 0.
Term order: Total degree inverse lexicographical
Coefficient term order: Total degree inverse lexicographical
```

Setting Options

All options can be set by calling

```
CGBOPT(O).
```

`O` is a list with an arbitrary number of elements. The elements of `O` (if existent) are interpreted as follows:

1st element: if 0 no output during computation, otherwise chatty.

2nd element: if 1 factorize coefficients, otherwise do not.

3rd element: if 0 use top reduction only, if 1 use "'normal"' reduction.

4th element: evaluate conditions using: if 0: simple method, if 1: reduced sets, if 2: Gröbner bases.

5th element: if 0: characteristic 0, otherwise arbritrary characteristic.

6th element: term order for polynomials

7th element: term order for coefficients

8th element: if 1 only the generic case is considered, otherwise all cases

8.5.5 Case Distinction and Polynomial Set

A *case distinction and polynomial set* is the data structure needed as input to compute comprehensive Gröbner bases. It contains a domain descriptor, an initial case distinction and a list of polynomials.

Reading Case Distinction and Polynomial Sets

A *case distinction and polynomial set* can be read in by entering

```
CDP:=CDPREAD().

IP (<list of coeff. variables>) (<list of main variables>)

0
(<list of polynomials = 0>)
1
(<list of polynomials <> 0>)
.

(<list of polynomials>)
.
```

Writing Case Distinction and Polynomial Sets

A case distinction and polynomial set `CDP` can be written by

```
CDPWRITE(CDP).
```

8.5.6 Gröbner Systems

Computing Gröbner Systems

A Gröbner system can be computed by calling

```
S:=GSYS(CDP).
```

where `CDP` is a case distinction and polynomial set (see 8.5.5).

Computing Factorized Gröbner Systems

This function is experimental! A factorized Gröbner system can be computed from a case distinction and polynomial set `CDP` by

```
S:=GSYSF(CDP).
```

Reducing Gröbner Systems

A Gröbner system `S` can be reduced by

```
S:=GSYSRED(S).
```

Writing Gröbner Systems

A Gröbner system `S` can be printed by entering

```
GSYSWRITE(S).
```

8.5.7 Comprehensive Gröbner Bases

Computing Comprehensive Gröbner Bases

A comprehensive Gröbner basis can be computed from a Gröbner System `S` by entering

```
CGB:=CGBFGSYS(S).
```

Writing Comprehensive Gröbner Bases

A comprehensive Gröbner basis `S` can be written to the output stream by

```
CGBWRITE(CGB).
```

Globally Reducing Comprehensive Gröbner Bases

A comprehensive Gröbner basis `CGB` can be globally reduced by

```
CGB:=CGBGLOBRED(CGB).
```

8.5.8 Quantifier Elimination

Computing quantifier free formulas

A quantifier free formula containing a condition for the existence of common zeroes of the polynomials in a comprehensive Gröbner basis `CGB` can be computed by

```
QFF:=CGBQFF(CGB).
```

8.5.9 Writing the actual state of computation

This feature is experimental! While MAS computes a Gröbner system sending `SIGUSR1` to MAS will cause the actual state of computation to be written on the terminal (even if output is redirected). This can be done from the command line by entering

```
kill -USR1 <pid of MAS>
```

8.5.10 A Sample Session

In this section we give a sample interactive session. Note that some uninteresting parts of the output (e.g. blank lines) have been omitted. First we start MAS.

```
{pesch@alice}[~]1: /mas

** Storage initialization ...
** ... completed.

Modula-2 Algebra System, Version 1.0

Copyrights:
(c) 1989 - 1996, MAS: H. Kredel, Uni Passau.
(c) 1982, SAC-2:      G. E. Collins, Uni Ohio, R. Loos, Uni Tuebingen.

(non-profit redistribution is permitted)
```

Next a set of polynomials and an initial (empty) case distinction is read.

```
MAS: CDP:=CDPREAD().

IP (a,b,c,d) (X,Y,Z)

0

()
1
()
.

(
( ( a**2 b**2 + a b ) X Y Z + ( c**3 d ) Y + 1 )
( X + ( c**2 d**2 + b ) Y )
).

ANS: [...]
```

We write this to the output stream.

```
MAS: CDPWRITE(CDP).

Case distinction:
Condition: Empty.

Polynomial set:
Ring: IP(a,b,c,d) (* Integral Polynomial *) (X,Y,Z)
( (  a**2 b**2 + a b ) X Y Z +  c**3 d  Y + 1  )

( (  c**2 d**2 + b ) Y + X )


ANS: ()
```

We set the options, and write them to the output stream.

```
MAS: CGBOPT(LIST(0,1,0,1,0,4,4,0)).
```

```
ANS: ()

MAS: CGBOPTWRITE().

Options for computation of Groebner systems are: ( 0,1,0,1,0,4,4,0,)
No output.
With factorization of coefficients.
Top-reduction only.
Conditions are evaluated using reduced sets.
Characteristic is 0.
Term order: Total degree inverse lexicographical
Coefficient term order: Total degree inverse lexicographical

ANS: ()
```

We compute a Gröbner system, reduce it and write the result to the output stream.

```
MAS: GS:=GSYS(CDP).

ANS: [...]

MAS: GSR:=GSYSRED(GS).

ANS: [...]
MAS: GSYSWRITE(GSR).


Groebner system:
Condition:
(  c**2 d**2 + b )= 0
(  a b +1 )<> 0
  b <> 0
  a <> 0

1 Condition.

Basis:
( (  c**2 d**2 + b ) Y + X )

( (  a**2 b**2 c**2 d**2 + a b c**2 d**2 + a**2 b**3 + a b**2 ) Y**2 Z -  c**3
d  Y - 1  )

Condition:
(  c**2 d**2 + b )<> 0
(  a b +1 )<> 0
  b <> 0
  a <> 0

1 Condition.

Basis:
( (  c**2 d**2 + b ) Y + X )

( (  a**2 b**2 c**2 d**2 + a b c**2 d**2 + a**2 b**3 + a b**2 ) X**2 Z +(  c**5
 d**3 + b c**3 d ) X -(  c**4 d**4 +2 b c**2 d**2 + b**2 ) )

Condition:
  c = 0
  d = 0
  b = 0
  a = 0
```

```
Condition:
  d = 0
  b = 0
  a = 0
  c <> 0

Condition:
  c = 0
  b = 0
  a = 0
  d <> 0

Condition:
  c = 0
  d = 0
  a = 0
  b <> 0

Condition:
  d = 0
  a = 0
  c <> 0
  b <> 0

Condition:
  c = 0
  a = 0
  d <> 0
  b <> 0

Condition:
  c = 0
  d = 0
  b = 0
  a <> 0

Condition:
  d = 0
  b = 0
  c <> 0
  a <> 0

Condition:
  c = 0
  b = 0
  d <> 0
  a <> 0

Condition:
  c = 0
  d = 0
(  a b +1 )= 0
  b <> 0
  a <> 0

Condition:
  d = 0
(  a b +1 )= 0
  c <> 0
  b <> 0
```

```
  a <> 0

Condition:
  c = 0
(  a b +1 )= 0
  d <> 0
  b <> 0
  a <> 0

12 Conditions.

Basis:
( (  a**2 b**2 + a b ) X Y Z +  c**3 d  Y + 1  )

Condition:
(  a b +1 )= 0
(  c**2 d**2 + b )<> 0
  c <> 0
  d <> 0
  b <> 0
  a <> 0

Condition:
  b = 0
  c**2 d**2 <> 0
  c <> 0
  d <> 0
  a <> 0

Condition:
  a = 0
(  c**2 d**2 + b )<> 0
  c <> 0
  d <> 0
  b <> 0

Condition:
  b = 0
  a = 0
  c**2 d**2 <> 0
  c <> 0
  d <> 0

4 Conditions.

Basis:
( (  a**2 b**2 c**2 d**2 + a b c**2 d**2 + a**2 b**3 + a b**2 ) X Y Z +(  c**5
d**3 + b c**3 d ) Y +(  c**2 d**2 + b ) )

( (  a**2 b**2 c**2 d**2 + a b c**2 d**2 + a**2 b**3 + a b**2 ) X Y Z -  c**3 d
  X +(  c**2 d**2 + b ) )

Condition:
(  c**2 d**2 + b )= 0
(  a b +1 )= 0
  c <> 0
  d <> 0
  b <> 0
  a <> 0

Condition:
```

```
(  c**2 d**2 + b )= 0
  c**2 d**2 = 0
  b = 0
  c <> 0
  d <> 0
  a <> 0

Condition:
(  c**2 d**2 + b )= 0
  a = 0
  c <> 0
  d <> 0
  b <> 0

Condition:
(  c**2 d**2 + b )= 0
  c**2 d**2 = 0
  b = 0
  a = 0
  c <> 0
  d <> 0

4 Conditions.

Basis:
( (  a**2 b**2 + a b ) X Y Z +  c**3 d  Y + 1  )

( (  c**2 d**2 + b ) Y + X )

ANS: ()
```

We compute a comprehensive Groebner basis and write it to the output stream.

```
MAS: CGB:=CGBFGSYS(GSR).

ANS: [...]

MAS: CGBWRITE(CGB).



Comprehensive-Groebner-Basis:
( (  c**2 d**2 + b ) Y + X )

( (  a**2 b**2 c**2 d**2 + a b c**2 d**2 + a**2 b**3 + a b**2 ) X**2 Z +(  c**5
 d**3 + b c**3 d ) X -(  c**4 d**4 +2 b c**2 d**2 + b**2 ) )

( (  a**2 b**2 c**2 d**2 + a b c**2 d**2 + a**2 b**3 + a b**2 ) X Y Z -  c**3 d
  X +(  c**2 d**2 + b ) )

( (  a**2 b**2 + a b ) X Y Z +  c**3 d  Y + 1  )

( (  a**2 b**2 c**2 d**2 + a b c**2 d**2 + a**2 b**3 + a b**2 ) X Y Z +(  c**5
d**3 + b c**3 d ) Y +(  c**2 d**2 + b ) )

( (  a**2 b**2 c**2 d**2 + a b c**2 d**2 + a**2 b**3 + a b**2 ) Y**2 Z -  c**3
d  Y - 1  )

22 Conditions.
```

```
ANS: ()
```

This is the end of the comprehensive Gröbner base example.

8.6 Non-commutative Gröbner bases and centers

In this section we discuss the background for a package which implements the theory of Kandri-Rody and Weispfenning [Kandri-Rody, Weispfenning 1988] (abbreviated by KW in the sequel). The implementation is based on an earlier implementation in the ALDES / SAC-2 system [Collins, Loos 1980]. For a treatment of the theory of solvable polynomial rings, including the case of non-commutative coefficient domains, and Gröbner bases see [Kredel 1992].

A solvable polynomial ring is an ordinary commutative polynomial ring $R = \mathbf{K}[X_1, \ldots, X_n]$ equipped with a new non-commutative multiplication $*$. The field $\mathbf{K}$ is assumed to be commutative and to commute with the indeterminates $X_1, \ldots, X_n$. The set T of terms (power-products of indeterminates) is supposed to be linearly ordered by an admissible order $<_T$ which is compatible with the new multiplication $*$. For a fixed term order $<_T$, $(R, *)$ is a solvable polynomial ring if the following axioms are satisfied:

Axioms: (KW 1.2.)

1. $(R, 0, 1, +, -, *)$ is an associative ring with 1.
2. For all $a, b \in \mathbf{K}$, $1 \leq h \leq i \leq j \leq k \leq n$, $t \in T(X_i, \ldots, X_j)$,
   (a) $a * bt = bt * a = abt$,
   (b) $X_h * bt = bX_h t$,
   (c) $bt * X_k = btX_k$.
3. For all $1 \leq i \leq j \leq n$ there exist $0 \neq c_{ij} \in \mathbf{K}$ and $p_{ij} \in R$ such that
$$X_j * X_i = c_{ij} X_i X_j + p_{ij}$$
   and $p_{ij} <_T X_i X_j$.

For commutator relations Q as in axiom (3), solvable polynomial rings will be denoted by

$$R = \mathbf{K}\{X_1, \ldots, X_n; Q\}.$$

By the following lemmas, the computation of the $*$-product is extended to arbitrary polynomials in R. For the proofs and further details see the original work and [Kredel 1990], [Kredel 1992].

Lemma 8.6.1 *(KW 1.3.) Let $R = \mathbf{K}\{X_1, \ldots, X_n, Q\}$ be a solvable polynomial ring, let $1 \leq i \leq n$ and let $f \in \mathbf{K}[X_1, \ldots, X_i]$, $g \in \mathbf{K}[X_i, \ldots, X_n]$. Then*

$$f * g = f \cdot g.$$

Lemma 8.6.2 *(KW 1.4.) Let $R = \mathbf{K}\{X_1, \ldots, X_n, Q\}$ be a solvable polynomial ring, let $<_T$ be a $*$-compatible admissible term order, and let $f, g \in R$. Then there exists an $h \in R$ such that*

$$f * g = c \cdot f \cdot g + h$$

and $h <_T f \cdot g$. Moreover, c and h are uniquely determined by f and g.

In the implementation we use an ordinary commutative distributive polynomial representation. Actually the Distributive Polynomial System of [Gebauer, Kredel 1983], implemented in the SAC-2 / ALDES system is used. The non-commutative product $*$ is defined via relations, which are elements of a free associative algebra. These relations are represented as triples (u, v, p) of (commutative) terms u, v and a (commutative) polynomial p, such that $u * v = p$ and p is of the form $c \cdot u \cdot v + p'$. Besides the defining relations between variables of the non-commutative product, many relations between powers of variables and terms are derived during computation. These relations are incrementaly stored in a so called relation table. Each time a product of terms is to be computed the relation table is scanned for an applicable relation. Missing relations between variables are treated as if the two variables commute.

Using the non-commutative product algorithm, the input-routines for polynomials can be setup to respect the order of variables in the products. For the rest of the Gröbner base algorithms the existing ones from the Buchberger algorithm system of [Gebauer, Kredel 1983a] could be used as a starting point. However great care was inorder to assure that for no algorithm the input parameters where interchanged. Further the non-commutative product may modify the leading coefficients of the product polynomials, so the order of computation steps had also to be checked. It is known, that not all criteria derived by Buchberger for the detection of unnecessary reductions are valid in the non-commutative case. The valid criterion BBEC is implemented as in the commutative case and leads to similar improvements of computing time. The computation of two-sided Gröbner bases uses an improved way of including right variable multiples during the main Buchberger loop instead of iterating the left Buchberger algorithm on bases given by right variable multiples of polynomials.

8.6.1 Relation Tables

The non-commutative polynomials are represented as ordinary commutative polynomials. A polynomial in distributive representation is a list of so called exponent vectors and so called base coefficients. The coefficient field $\mathbf{K}$ is the field of rational numbers $\mathbf{Q}$ in the current implementation.

The commutator relations from 8.6(3) are implemented as triples of commutative polynomials. More precisely a relation $X_j * X_i = c_{ij} X_i X_j + p_{ij}$ for some $1 \leq i \leq j \leq n$ is represented as triple

$$(X_j, X_i, c_{ij} X_i X_j + p_{ij})$$

of distributive polynomials. Missing relations between variables are treated as if the relation $X_j * X_i = X_i X_j$ was specified, that means as if a relation with $c_{ij} = 1$ and $p_{ij} = 0$ was defined.

The set of all commutator relations is stored in a list called *relation table*. So that in order to compute the product $X_j * X_i$ one has to look for a triple starting with (X_j, X_i, p) and the

take the third polynomial in this triple. The relation table must be maintained through recursive applications of the $*$-product algorithm and we want to use all computed relations to be accessible at any time during further recursive calls. This table is implemented as a list of distributive polynomials:

$$T = (u_1, v_1, p_1, \ldots, u_t, v_t, p_t)$$

where the $u_i = X_{j_i}^{e_i}$, $v_i = X_{k_i}^{l_i}$ and $p_i = c_i \cdot X_{k_i}^{l_i} \cdot X_{j_i}^{e_i} + p_i'$. The table entries are partially ordered with respect to divisibility of the relation heads (u_i, v_i).

Definition: If T is a relation table, then the following condition holds:

> for all $1 \leq i \leq t$ there does not exist $1 \leq i < j \leq t$ such that $u_i \mid u_j$ and $v_i \mid v_j$.

This means that heads which are 'later' in the table may divide relation heads, which come 'earlier' in the table. If T is empty at the beginning, in this case **all** variables commute, then no non-commuting relation will ever be computed during `DINPPR` and T will remain empty. The search for a product relation goes from left to right in the list so one finds a relation with maximal exponents. The search is successful, if both exponents of u_i and v_i divide the exponents of the relation we look for. If no relation matches, we assume the variables to commute, i.e. we assume $c = 1$ and $p = 0$.

8.6.2 Left and Two-sided Gröbner Bases

With the definition of a suitable (left/right) reduction most concepts of commutative Gröbner bases carry over to solvable polynomial rings.

Definition: (Left Reduction) Let R be a solvable polynomial ring. Let $p \in R$, $t \in T$. Then the *left reduction* $\longrightarrow_{t,p} \subseteq R \times R$ is defined as follows:

> For $f, f' \in R$, $t \in T(f)$, $f \longrightarrow_{t,p} f'$ iff there exists $u \in T$ such that $t = u \cdot \mathrm{HT}(p) = \mathrm{HT}(u * p)$ and
>
> $$f' = f - a_u * u * p,$$
>
> where $a_u \in \mathbf{K}^*$ is the unique element of $\mathbf{K}^*$ such that $\mathrm{coeff}(t, f) = a_u * \mathrm{coeff}(t, u * p)$.

By construction $t \notin T(f')$. If for certain f, t no such u exists, then t in $T(f)$ is called *irreducible wrt. p.*

If the left reduction relation $\longrightarrow_G$ (for some finite subset $G \subset R$) is confluent, then G is called a *left Gröbner base.* And the following theorem indicates the construction of left Gröbner bases using left S-polynomials.

Theorem 8.6.3 *Let G be a finite subset of R, then the following assertions are equivalent.*

1. *G is a left Gröbner base.*
2. *For all $f \in \mathrm{ideal}_l(G)$, $f \longrightarrow_G^* 0$.*
3. *For all $h \in H = \{LSP(f, g) \ : \ f, g \in G, f \neq g\}$, $h \longrightarrow_G^* 0$.*

Let $\text{ideal}_t(P)$ denote the two-sided ideal generated by $P \subseteq R$. The main key to the construction of two-sided Gröbner bases of two-sided ideals is contained in the following lemma.

Lemma 8.6.4 *Let G be a left Gröbner base in R, then* $\text{ideal}_l(G) = \text{ideal}_t(G)$ *implies* $\text{ideal}_r(G) = \text{ideal}_t(G)$.

The construction of two-sided Gröbner bases using left Gröbner bases is indicated in the following theorem.

Theorem 8.6.5 *Let G be a finite subset of R. Then the following assertions are equivalent:*

1. *G is a left Gröbner base and* $\text{ideal}_l(G) = \text{ideal}_t(G)$,
2. *For all $f \in R$ with $f \in \text{ideal}_t(G)$: $f \longrightarrow_G^* 0$,*
3. *G is a left Gröbner base and for all $1 \leq i \leq n$, $p \in G$: $p * X_i \longrightarrow_G^* 0$,*

Examples

The algorithms have already been discussed in section 7.7 and we continue with the discussion of examples. The listing of an example is given in table 8.4. The input of non-commutative polynomials consists of two steps:

1. the input of the commutator relations together with the list of variables of the polynomial ring and the desired term order,
2. the input of the non-commutative polynomials itself.

The commutator relations are a list of commuting polynomials which are read by the MAS function `PREAD`. `PREAD` reads from the current input stream and returns a list of distributive rational polynomials in internal representation. See section 7.4.3 for more details on `PREAD`. `PREAD` expects the following items:

The variable list: '`(a, x, y)`'. A variable name may consist of a alpha-numerical character sequence starting with a letter. All variables occurring in the polynomials must be specified.

The desired term order: '`L`'. The term order may be one of the following:

L for the inverse lexicographical term order

G for the inverse graduated term order

polynomial list a list of univariate (integral) polynomials in the variable `T` (this name is fixed). See 7.4.3 for details.

There is no check if the term order is compatible with the commutator relations or if the linear form defines an admissible term order.

The commutator relations themselves: '`( y ), ( x ), ( x y + a )`'. This relation is interpreted as $y * x = xy + a$. Not specified relations are interpreted as if the variables

commute. In this example a commutes both with x and y. The relations must be given as a list (!) of polynomial triples.

The second input consists of the non-commutative polynomials. `NPREAD` takes as input a relation table and reads a list (!) of polynomials from the current input stream. The output is a list of distributive rational polynomials. `**` denotes exponentiation, the multiplication operator `*` may be omitted. That means `x y` denotes `x * y`. All multiplications of variables mean the non-commutative $*$-product. It is possible to specify also more complex polynomial expressions. See 7.4.3 for the accepted syntax. Be sure to include enough parenthesis to avoid ambiguities.

```
(* Commutator relations: *)
t:=PREAD().
(a,x,y) L
(
( y ), ( x ), ( x y + a ),
)
PWRITE(t).

(* Non-commutative polynomials: *)
p:=NPREAD(t).
(
( y**3 + x**2 y + x y ),
( x**2 + x )
)
PWRITE(p).

c:=LIRRSET(t,p). (* Left Normalform *)
PWRITE(c).

c:=LGBASE(t,p,1). (* Left G-base *)
PWRITE(c).

c:=TSGBASE(t,p,1). (* Two sided G-base *)
PWRITE(c).
```

Table 8.4: Computing example input

Next three of the above discussed algorithms are called. The produced output is shown in tables 8.5, 8.6 and 8.7. In any case the output polynomials are printed to the current output stream with the procedure `PWRITE`. `PWRITE` prints the actual variable list, the actual term order and the list of polynomials, each polynomials starting on a new line.

`LIRRSET`: The input parameters are '`t`' the relation table and '`p`' the polynomial list. The output '`c`' is the left irreducible set of the input, it is listed in table 8.5.

`LGBASE`: The input parameters are '`t`' the relation table, '`p`' the polynomial list and '`1`' a trace flag. The output '`c`' is the left Gröbner base of the input, it is listed in table 8.6.

`TSGBASE`: The input parameters are '`t`' the relation table and '`p`' the polynomial list. The output '`c`' is the two-sided Gröbner base of the input, it is listed in table 8.7.

Next we reproduce a summary table 8.8 of computing times for several variants of the above algorithms on various machines. The systems and machines are ALDES on IBM 9370/VM, MAS on an Atari 1040 ST (8 Mhz), an PC AT/386SX (16 Mhz) and an IBM RS600/520 (20 Mhz). The timings are obtained with a fresh list of commutator relations

```
Polynomial in the variables: (a,x,y)
Term ordering:  inverse lexicographical.
Polynomial list:
          (  y**3 -2 a x - a )
          (  x**2 + x )
```

Table 8.5: Computing example left irreducible set

```
Polynomial in the variables: (a,x,y)
Term ordering:  inverse lexicographical.
Polynomial list:
            a**2
          (  x**2 + x )
            a y**2
          (  y**3 -2 a x - a )
```

Table 8.6: Computing example left Gröbner base

```
Polynomial in the variables: (a,x,y)
Term ordering:  inverse lexicographical.
Polynomial list:
            a
          (  x**2 + x )
            y**3
```

Table 8.7: Computing example two-sided Gröbner base

in each case and not in the sequence suggested by the above input listing.

'DINLGB, - irred.' means that the algorithm did not compute a reduced (irreducible) left GB. 'DINLGB, + irred.' means that the algorithm computed a reduced (irreducible) left GB. 'DINLGB, BBEC, + irr.' means that the algorithm computed a reduced (irreducible) left GB and used the condition 'BBEC' (Buchberger's criterion 2) to avoid unnecessary reductions. In the later case 22 polynomials have not been reduced according to the criterion from a total of 34 S-polynomials.

The algorithms 'DINCGB' and 'DIN1GB' both compute two-sided reduced Gröbner Bases. 'DIN1GB' denotes the MAS algorithm corresponding to the algorithm 'GROEBNER' of [Kandri-Rody, Weispfenning 1988]. 'DINCGB' is superior to 'DIN1GB' due to the fact, that the polynomials $p * X_i$ are added to the base at the beginning of the computation, so much more polynomials will be reducible later on. 'DINCGB, BBEC' means that the algorithm computed a reduced (irreducible) two-sided GB and used the condition 'BBEC' to avoid unnecessary reductions. In the later case 9 polynomials have not been reduced according to the criterion from a total of 14 S-polynomials.

Algorithm	IBM 9370/VM ALDES/SAC-2	Atari ST MAS	AT/386sx MAS	IBM RS6000 MAS
DINLIS	0.03	< 1.0	< 1.0	0.03
DINLGB, - irred.	1.47			
DINLGB, + irred.	1.47	18.0	13.0	1.30
DINLGB, BBEC + irr		6.0		0.47
DIN1GB	1.93			
DINCGB	0.58	8.0	5.0	0.45
DINCGB, BBEC		4.0		0.23

Computing time in seconds.

Table 8.8: Computing Time Summary: Gröbner Bases

8.6.3 Center of solvable polynomial rings

The center of a solvable polynomial ring consists exactly of the polynomials which commute with all variables. Further elements in the center may be computed and non-commuting variables have only trivial centralizer in case the underlying ring has characteristic zero.

Let $\mathbf{R}$ be a (commutative) ring with 1 over a field $\mathbf{K}$. Let $S = \mathbf{R}\{X_1, \ldots, X_n; Q\}$ be a solvable polynomial ring over $\mathbf{R}$ in the variables $X_1, \ldots, X_n$, such that the coefficients commute with the variables (i.e. $c_{ai} = 1$ and $p_{ai} = 0$ for $1 \leq i \leq n$, $0 \neq a \in \mathbf{R}$).

Definition: Let S be a ring. The *center of* S is the set of all elements of S which commute with all elements of S:

$$\mathrm{Cen}(S) = \{a \in S \ : \ ab = ba \text{ for all } b \in S\}.$$

Let I be a subset of S. The *centralizer of* I *in* S is the set of all elements of S which commute with all elements of I:

$$\mathrm{Cen}_S(I) = \{a \in S \ : \ ab = ba \text{ for all } b \in I\}.$$

Proposition 8.6.6 *Let $S = \mathbf{R}\{X_1, \ldots, X_n; Q\}$ be a solvable polynomial ring over a commutative field $\mathbf{R}$ in the variables $X_1, \ldots, X_n$. Let $X = \{X_1, \ldots, X_n\}$. Then*

$$\mathrm{Cen}(S) = \mathrm{Cen}_S(X).$$

Proposition 8.6.6 provides a means to determine elements in the center up to any degree bound. One takes a polynomial f with indeterminate coefficients, then a necessary and sufficient condition for $f \in \mathrm{Cen}(S)$ is that f must commute with all variables X_i, $i = 1, \ldots, n$. This gives a system of linear equations for the coefficients of f. Solving it answers the question if for some values of the coefficients $f \in \mathrm{Cen}(S)$.

Proposition 8.6.7 *Let $S = \mathbf{R}\{X_1, \ldots, X_n; Q\}$ be a solvable polynomial ring over $\mathbf{R}$ in the variables $X_1, \ldots, X_n$. Let $X = \{X_1, \ldots, X_n\}$.*
Given a finite set of terms $T' = \{t_1, \ldots, t_k\}$ $k \in \mathbf{N}$ in $T(X_1, \ldots, X_n)$, then there is an algorithm, which decides if there is a polynomial

$$f = \sum_{i=1}^{k} a_i t_i \ \in \ \mathrm{Cen}(S)$$

for some $a_i \in \mathbf{R}$, $1 \leq i \leq k$. Moreover the algorithm determines all such polynomials.

Example: Centers of Enveloping Algebras

We include some examples of the computation of centers of enveloping algebras of some finite dimensional Lie algebras over the rational numbers. The examples are taken from [Patera *et al.* 1976] and compared to their results. The enumeration of the Lie algebras is as follows: $A_{i,j}$ denotes the j-th Lie algebra of dimension i. The examples are contained in tables 8.9, 8.10, 8.11, 8.12 and 8.13. Note that we only present examples which have polynomial invariants. Actually there are also rational functions and analytical functions which are invariant under the commutator product of the Lie algebra (see example 8.9).
In the examples we list

1. the defining commutator relations of the enveloping algebra of a Lie algebra,
2. the statement for the generation of the terms,
3. the center polynomial with parametric coefficients,
4. the specialized center polynomials,
5. the computing time on an Atari 1040 ST.

The input syntax is as described in subsection 8.6.2. `CenterPol` denotes the 'driver' algorithm, which calls `DINCCP` and then specializes the coefficients to obtain a **Q**-vectorspace basis of the center. `EVLGIL` takes a list of exponents $(e_1, \ldots, e_n)$ and delivers a set of terms with exponents $(a_1, \ldots, a_n)$ with $0 \leq a_i \leq e_i$ for $1 \leq i \leq n$. `EVLGTD` has inputs (n, d, E), where n is the number of variables, d is the total degree and E is a list of already computed terms (used for internal recursion). It returns a list $(E_0, \ldots, E_d)$ where each E_i is a list of exponents of terms in n variables and of total degree exactly i.

The output further shows the names of the parameter variables in the coefficients, then the full center polynomial with parametric coefficients and then the specialized polynomials. The computing times are splitted into the time for input '`read =`', time for evaluation '`eval =`', time for output '`print =`' and time spend in garbage collection '`gc =`'. The first three times do not include garbage collection times. The computing times are summarized also in table 8.14.

```
(* Commutator relations: *) t:=PREAD().
(e1,e2,e3) G
(
( e3 ), ( e1 ), ( e1 e3 - e1 ),
( e3 ), ( e2 ), ( e2 e3 - e2 ),
)
(*generate terms. *) e:=EVLGIL(LIST(1,1,1)).
(*compute polynomial in the center. *) x:=CenterPol(t,e).

Parameters: (X1)

Center polynomial:
           X1

Specialized center polynomials:
           1

Time: read = 0, eval = 4, print = 0, gc = 0.
```

Table 8.9: Lie Algebra: $A_{3,2}$

Note, that in example 8.9 we do not find a polynomial in the center. But Patera *et. al.* show that there are analytic functions, which are invariant under the commutator product in the Lie algebra. Namely $e_1 \exp(-\frac{e_2}{e_1})$.

In example 8.10 we do not obtain the constants of $\mathbf{Q}$ respectively the specialized polynomial 1, since we did not ask for it. We only asked for polynomials in the center of homogeneous total degree 2.

In example 8.13 we obtain more polynomials than [Patera *et al.* 1976]. But observe, that the first polynomial is the product of polynomials 4 and 5 and the second polynomial is the product of the polynomials 5 and 6. This raises the question of canonical bases for subrings.

A summary of the above examples including computing times for MAS on an Atari 1040 ST is contained in the next table 8.14. We list the respective Lie algebra in column one, the dimension of the Lie algebra in column 2, then the total degree and the exponents as input to the term generating algorithm. The last column contains the computing times. The column entitled 'polynomials' contains before the slash '/' the number of specialized polynomials as produced by the algorithms and after the slash the number of polynomials as listed in [Patera *et al.* 1976].

```
(* Commutator relations: *) t:=PREAD().
(e1,e2,e3) G
(
( e3 ), ( e1 ), ( e1 e3 - e1 ),
( e3 ), ( e2 ), ( e2 e3 + e2 ),
)
(*generate terms. *)
e:=EVLGTD(3,2,NIL). e:=INV(e). e:=FIRST(e).
(*compute polynomial in the center. *) x:=CenterPol(t,e).

Parameters: (X1)

Center polynomial:
            X1 e1 e2

Specialized center polynomials:
            e1 e2

Time: read = 0, eval = 4, print = 0, gc = 0.
```

Table 8.10: Lie Algebra: $A_{3,4}$

```
(* Commutator relations: *) t:=PREAD().
(e1,e2,e3) G
(
( e2 ), ( e1 ), ( e1 e2 - e3 ),
( e3 ), ( e2 ), ( e2 e3 - e1 ),
( e3 ), ( e1 ), ( e1 e3 + e2 ),
)
(*generate terms. *) e:=EVLGIL(LIST(2,2,2)).
(*compute polynomial in the center. *) x:=CenterPol(t,e).

Parameters: (X1,X2)

Center polynomial:
            (  X2 e3**2 + X2 e2**2 + X2 e1**2 + X1 )

Specialized center polynomials:
            (  e3**2 + e2**2 + e1**2 )
            1

Time: read = 0, eval = 64, print = 0, gc = 16.
```

Table 8.11: Lie Algebra: $A_{3,9}$

```
(* Commutator relations: *) t:=PREAD().
(e1,e2,e3,e4) G
(
( e4 ), ( e2 ), ( e2 e4 - e1 ),
( e4 ), ( e3 ), ( e3 e4 - e2 ),
)
(*generate terms. *) e:=EVLGIL(LIST(0,1,2,1)).
(*compute polynomial in the center. *) x:=CenterPol(t,e).

Parameters: (X1,X2,X3)

Center polynomial:
            ( -2 X3 e1 e3 + X3 e2**2 + X2 e1 + X1 )

Specialized center polynomials:
            ( -2 e1 e3 + e2**2 )
            e1
            1

Time: read = 0, eval = 6, print = 0, gc = 8.
```

Table 8.12: Lie Algebra: $A_{4,1}$

```
(* Commutator relations: *) t:=PREAD().
(e1,e2,e3,e4,e5,e6) G
(
( e2 ), ( e1 ), ( e1 e2 - e3 ),
( e3 ), ( e1 ), ( e1 e3 - e4 ),
( e5 ), ( e1 ), ( e1 e5 - e6 ),
)
(*generate terms. *) e:=EVLGIL(INV(LIST(0,1,2,1,1,1))).
(*compute polynomial in the center. *) x:=CenterPol(t,e).

Parameters: (X1,X2,X3,X4,X5,X6,X7)

Center polynomial:
            ( -2 X7 e2 e4 e6 + X7 e3**2 e6 + X6 e4 e6 -
               X5 e3 e6 + X5 e4 e5 -2 X4 e2 e4 + X4 e3**2 +
               X3 e6 + X2 e4 + X1 )

Specialized center polynomials:
            ( -2 e2 e4 e6 + e3**2 e6 )
            e4 e6
            ( - e3 e6 + e4 e5 )
            ( -2 e2 e4 + e3**2 )
            e6
            e4
            1

Time: read = 0, eval = 140, print = 0, gc = 24.
```

Table 8.13: Lie Algebra: $A_{6,1}$

Lie Algebra	dim	total degree	exponents	polynomials	time
$A_{3,2}$	3	≤ 3	$(1,1,1)$	1/2	4
$A_{3,4}$	3	$= 2$		1/2	4
$A_{3,9}$	3	≤ 6	$(2,2,2)$	2/2	64
$A_{4,1}$	4	≤ 4	$(0,1,2,1)$	3/3	6
$A_{6,1}$	6	≤ 6	$(0,1,2,1,1,1)$	7/5	140

Computing time in seconds.

Table 8.14: Computing Time Summary: Center

8.7 Linear Equations and Modules over Polynomial rings

This section we discuss the problem of solving linear polynomial equations, respectively systems of linear polynomial equations. These methods can be used to solve the ideal intersection and the ideal quotient problem. Finally we discuss the problem of finding canonical bases for submodules of modules over polynomial rings using the method of partial Gröbner bases. We will describe only the commutative case, but the problem can also be solved in the non-commutative solvable type polynomial ring case for the corresponding left and right problems. For a complete treatment of the commutative case see [Becker, Weispfenning 1993], for the non-commutative case see [Kredel 1992].

Let $R = \mathbf{K}[X_1, \ldots, X_n]$ be a polynomial ring over a field $\mathbf{K}$ with respect to an admissible term order $<$. For the non-commutative cases let $R = \mathbf{K}\{X_1, \ldots, X_n; Q\}$ be a solvable polynomial ring over a field $\mathbf{K}$ with respect to a $*$-compatible admissible term order $<$ and commutator relations Q. By R^m we denote the free module over R, i.e. the m fold cartesian product of R together with a module structure.

8.7.1 Linear Equations and Syzygies

A system of linear equations over $\mathbf{R}$ in the variables $Y_1, \ldots, Y_m$ (disjoint to $X_1, \ldots, X_n$) with coefficients f_{ij}, $g_i \in R$, $1 \leq i \leq k$, $1 \leq j \leq m$ is a system of equations of the following form:

$$\begin{array}{ccccc} f_{11}Y_1 & + \ldots + & f_{1m}Y_m & = & g_1 \\ \vdots & \ddots & \vdots & = & \vdots \\ f_{k1}Y_1 & + \ldots + & f_{km}Y_m & = & g_k \end{array} \qquad (*)$$

By a solution of the system of equations $(*)$ we mean any m tuple $(h_1, \ldots, h_m) \in R^m$, such that

$$\begin{array}{ccccc} f_{11}h_1 & + \ldots + & f_{1m}h_m & = & g_1 \\ \vdots & \ddots & \vdots & = & \vdots \\ f_{k1}h_1 & + \ldots + & f_{km}h_m & = & g_k \end{array}$$

holds.

A method to find a solution is the following.

1. Find a solution for a single *homogeneous* equation, i.e. let $k = 1$ and $g_1 = 0$ in $(*)$:

$$f_1Y_1 + \ldots + f_mY_m = 0.$$

   Solutions to this problem are called *syzygies*. The set of all such solutions forms a submodule of R^m. Using Gröbner bases technics one can find a generating set for this submodule (see below). If $\{f_1, \ldots, f_m\}$ does not already form a Gröbner base we first solve the problem for the corresponding Gröbner base and the use some transformation to obtain the solution for the original problem.

2. Find a solution for a single *inhomogeneous* equation, i.e. let $k = 1$ and $g_1 \neq 0$ in $(*)$:

$$f_1Y_1 + \ldots + f_mY_m = g.$$

One particular solution to this problem is found by the ideal membership test $g \in \text{ideal}(f_1, \ldots, f_m)$ and the representation of g as element of the ideal generated by $f_1, \ldots, f_m$. There is a solution to this problem, iff g is contained in this ideal. The set of all solutions is then the set of sums of solutions of the homogeneous system and the particular solution of the inhomogeneous system.

3. In the general case we reduce the problem to a single equation problem, using an embeding of the problem to a polynomial ring in $Z_1, \ldots, Z_k$ new variables and by multiplication of the i-th equation with Z_i and summing up the rows.

$$(f_{11}Z_1 + \ldots + f_{k1}Z_k)Y_1 + \ldots + (f_{1m}Z_1 + \ldots + f_{km}Z_k)Y_m = (g_1Z_1 + \ldots + g_kZ_k) \quad (**)$$

Then we can solve this equation with the methods described in 1) and 2) above in a way such that the solutions are free of the variables $Z_1, \ldots, Z_k$. The later method is the method of partial Gröbner bases and is discussed in the next section. Then it can be shown that $(h_1, \ldots, h_m)$ is a solution of $(*)$ iff $(h_1, \ldots, h_m)$ is a solution of $(**)$.

In more detail the first problem is solved as follows.

Let S be a ring. $P = \{p_1, \ldots, p_m\}$ be a finite subset of S. Let

$$M_P = \{(h_1, \ldots, h_m) \in S^m \ : \ h_1p_1 + \ldots + h_mp_m = 0\}.$$

M_P is called *module of syzygies for* P. The elements of M_P are called *syzygies of* P and M_P is a submodule of the free module R^m.

Let R be a polynomial ring with respect to an admissible term order. A set of polynomials is called *monic* if the head coefficients of all polynomials in the set are 1. Let $P = \{p_1, \ldots, p_m\}$ be a monic Gröbner base in R. For $1 \leq i < j \leq m$ let

$$f_{ij} = SPol(p_i, p_j) = a_{ij}u_{ij}p_i - b_{ij}v_{ij}p_j$$

be the S-polynomials of all distinct pairs of elements of P; with $0 \neq a_{ij}, b_{ij} \in \mathbf{K}$, $u_{ij}, v_{ij} \in T$ and $\text{HT}(f_{ij}) < \text{HT}(u_{ij}p_i) = u_{ij}\text{HT}(p_i) = v_{ij}\text{HT}(p_j) = \text{HT}(v_{ij}p_j)$.

Since $f_{ij} \in \text{ideal}(P)$ and P is a Gröbner base we have $f_{ij} \longrightarrow_P^* 0$. This reduction determines a representation of f_{ij} from which terms belonging to the same p_i can be combined to a polynomial q_{ijk}:

$$f_{ij} = \sum_{k=1}^{m} q_{ijk}p_k$$

with $q_{ijk} \in R$ and $\text{HT}(q_{ijk}p_k) = \text{HT}(q_{ijk}p_k) \leq \text{HT}(f_{ij})$. Subtracting both representations of f_{ij} we obtain a syzygy of P. More precisely let $b_{ij} = (r_{ij1}, \ldots, r_{ijm}) \in R^m$ with

$$r_{ijk} = \begin{cases} q_{ijk} & k \neq i, j \\ q_{ijk} - a_{ij}u_{ij} & k = i \\ q_{ijk} + b_{ij}v_{ij} & k = j \end{cases}$$

then $B_P = \{b_{ij} \ : \ 1 \leq i < j \leq m\}$ is a set of syzygies of P.

The next theorem says, that if this construction is applied to a Gröbner base G, then the B_G is already a generating set for the module of syzygies.

Theorem 8.7.1 *Let R be a polynomial ring with respect to an admissible term order. Let G be a monic Gröbner base in R and let B_G be the set of syzygies as defined before. Then B_G generates M_G as an R-module.*

For arbitrary ideal bases, there is a 'transformation' lemma for syzygies with respect to a Gröbner base to syzygies for an arbitrary ideal base. The fact was reported by [Zacharias 1978] for commutative polynomial rings and was reported in [Apel, Lassner 1988] for enveloping algebras of Lie algebras.

Proposition 8.7.2 *Let I be an ideal in a ring R and let $F, G \subset R$, $F = \{f_1, \ldots, f_m\}$, $G = \{g_1, \ldots, g_l\}$ such that $I = \text{ideal}(F) = \text{ideal}(G)$. Let $X = (q_{ij})$ with $q_{ij} \in R$ for $1 \leq i \leq m$, $1 \leq j \leq l$ and $Y = (p_{ij})$ with $p_{ij} \in R$ for $1 \leq i \leq m$, $1 \leq j \leq l$ be the transformation matrices between the F and G:*

$$\begin{aligned} f_i &= \sum_{j=1,\ldots,l} p_{ij} * g_j \quad 1 \leq i \leq m \\ g_j &= \sum_{i=1,\ldots,m} q_{ij} * f_i \quad 1 \leq j \leq l \end{aligned}$$

If we consider the F and G also as vectors and denote matrix transposition by ^{t}, we can write more compactly: $G^t = XF^t$ and $F^t = YG^t$. Let I_m denote the $m \times m$ unit matrix. Let B_G be a generator of M_G, then B_F (in block matrix representation) defined by

$$B_F = \begin{pmatrix} I_m - YX \\ B_G X \end{pmatrix}$$

is a generator of M_F.

Note that for a Gröbner base G, Y can be computed during the construction of G and X can be computed by reduction of the $f_i \in F$ wrt. G.

By this we have the main ingredients for the solution of step 1) of the method above. Step 2) only requires some linear algebra to be seen to be correct and step 3) needs some results on partial Gröbner bases discussed later. We turn now to some applications, which can be solved using the methods in 1) above.

Ideal Intersection

In this subsection we are going to reformulate the ideal intersection problem as syzygy problem. Another way to solve the ideal intersection problem is by the so called 'tag variable' method. The main fact, using syzygies, is contained in the following lemma.

Lemma 8.7.3 *Let R be a polynomial ring and let $F_1 = \{f_1, \ldots, f_r\} \subset R$, $F_2 = \{g_1, \ldots, g_s\} \subset R$. Let $F = F_1 \cup F_2$ and let B_F be a generating set for the module of syzygies of F. Then*

$$\text{ideal}(F_1) \cap \text{ideal}(F_2) = \text{ideal}(P)$$

where $P = \{p_1, \ldots, p_k\} \subset R$ with $p_j = \sum_{i=1}^{r} h_{ij} f_{ij}$ for $1 \leq j \leq k = |B_F|$ and $(h_{1j}, \ldots, h_{rj}, h'_{1j}, \ldots, h'_{sj}) \in B_F$.

A special case of the ideal intersection problem is the question of the existence of left common multiples for two elements in a non-commutative ring. That means, given $a, b \in R$ (ring), do there exist $b', a' \in R$ such that

$$b'a = a'b \qquad (*)$$

It is known that this problem is always positively solvable in a Noetherian domain. If the ideal membership problem is solvable in such a ring, then the proof of this fact can be adapted to obtain an algorithm for the computation of such left common multiples.

An other method to determine left common multiples is to consider the equation $(*)$ as an ideal intersection problem $Sa \cap Sb = \emptyset$? or directly as a syzygy problem $b'a - a'b = 0$. The last way is persued and discussed e.g. in [Apel, Lassner 1988].

Ideal Quotient

In this subsection we are going to define ideal quotients and show how ideal quotients can be computed using syzygy methods.

Let S be a ring. Let $I \subseteq S$ be an ideal, and let $J \subseteq S$ be a subset of S. Then the set

$$I : J \ = \ \{h \in S \ : \ hg \in I \text{ for all } g \in J\}$$

is called the *ideal quotient of I by J*.

$I : J$ is an ideal of S, since for $h_1 g \in I$, $h_2 g \in I$ and $f \in S$ also $(h_1 - h_2)g \in I$ and $f h_1 g \in I$ (because I is an ideal) for all $g \in J$. If I is an ideal generated by a finite subset $F = \{f_1, \ldots, f_k\}$ of S, i.e. $I = \text{ideal}(F)$, then $\text{ideal}(F) : J = \{h \in S \ : \ \text{exists } h_1, \ldots, h_k \in S, \text{ with } hg = h_1 f_1 + \ldots + h_k f_k \text{ for all } g \in J\}$. If moreover S is commutative and $J = \text{ideal}(G)$ is an ideal in S generated by a set G in S then $I : J = I : G$. This holds because $hg \in I$ for all $g \in J$ iff $hg_j \in I$ for all $g_j \in G$, since all $g \in J$ have a representation $g = \sum_j p_j g_j$ by elements of G and since S is commutative we also have $hg = h(\sum_j p_j g_j) = \sum_j p_j (h g_j)$.

If G is a subset of S and I is an ideal in S, then

$$I : G = \bigcap_{g \in G} I : \{g\}.$$

This identity holds, since for ideals $hg \in I$ for all $g \in G$ if and only if $h \in I : \{g\}$ for all $g \in G$. In particular for finite $G = \{g_1, \ldots, g_k\}$ we have $I : G = \bigcap_{i=1,\ldots,k} I : \{g_i\}$.

Having reduced the problem of determination of ideal quotients to $I : \{g\}$ (which we will simply denote by $I : g$) we now solve this problem using syzygies for polynomial rings R.

Lemma 8.7.4 *Let R be a polynomial ring. Let $I = \text{ideal}(F)$ be an ideal in R generated by a finite set $F = \{f_1, \ldots, f_k\} \subseteq R$ and let $g \in R$. Let $F' = \{g, f_1, \ldots, f_k\}$ and let $B_{F'}$ be a generating set for the module of syzygies of F'. If we let*

$$H = \{h \in S \ : \ \text{ exists } h_1, \ldots, h_k \in S, \text{ with } (h, h_1, \ldots, h_k) \in B_{F'}\},$$

then $\text{ideal}(H) = \text{ideal}(F) : g$.

8.7.2 Gradings and Partial Gröbner Bases

By a *grading* γ of a polynomial ring $\mathbf{K}[X_1, \ldots, X_n]$ with set of terms T we mean a monoid homomorphism

$$\gamma : (T, 1, \cdot) \longrightarrow (\mathbf{N}, 0, +).$$

This means that γ is a mapping from T to $\mathbf{N}$ such that $\gamma(1) = 0$ and $\gamma(s \cdot t) = \gamma(s) + \gamma(t)$ for $s, t \in T$. For polynomials $0 \neq f \in \mathbf{K}[X_1, \ldots, X_n]$ we define the γ-degree of f, which will also be denoted by $\gamma(f)$, as $\gamma(f) = \max\{\gamma(t) \ : \ t \in T(f)\}$. For solvable polynomial rings $R = \mathbf{K}\{X_1, \ldots, X_n; Q, Q'\}$ we define the γ-degree of $0 \neq f \in R$ by the γ-degree of f as element of $\mathbf{K}[X_1, \ldots, X_n]$.

A grading γ on T with *weights* $a_1, \ldots, a_n \in \mathbf{N}$ can be defined as

$$\gamma(t) = \gamma(X_1^{\nu_1} \ldots X_n^{\nu_n}) = a_1\nu_1 + \ldots + a_n\nu_n,$$

where $t = X_1^{\nu_1} \ldots X_n^{\nu_n} \in T$. Moreover since γ is a homomorphism between $(T, 1, \cdot)$ and $(\mathbf{N}, 0, +)$ it can be shown, that any grading on T arises from a linear form. Moreover is holds, that $s \mid t$ implies $\gamma(s) \leq \gamma(t)$ for all $s, t \in T$. And since a polynomial ring over a domain is a domain we have: $\gamma(fg) = \gamma(f) + \gamma(g)$ for all $0 \neq f, g \in S$. By definition of the degrees of polynomials it holds also that $\gamma(f + g) \leq \max\{\gamma(f), \gamma(g)\}$ for all $0 \neq f, g \in S$.

We call an element f of R *homogeneous of degree* d if for all $t \in T(f)$ we have $\gamma(t) = d$ and we call it *homogeneous* if for all $t, s \in T(f)$ we have $\gamma(t) = \gamma(s)$. So every polynomial f of R can be represented as a (finite) sum of its homogeneous components: $f = \sum_{i \in \mathbf{N}} f_i$, where each f_i is homogeneous of degree i. An ideal I in R is called *homogeneous* if it is generated by homogeneous elements. One can prove, that I is homogeneous, *iff* with each $f \in I$ it contains every homogeneous component of f. Homogeneous ideals are denoted by $\text{ideal}^h(F)$. A representation of a polynomial $f \in R$ with respect to a set of homogeneous polynomials P defines a *homogeneous representation* of f with respect to P.

Note that contrary to commutative polynomial rings homogeneity is in general not preserved under the $*$-product in solvable polynomial rings. This restricts the possible gradings to those which are so called *homogeneity compatible with* $*$. But it can be shown, that for the purpose of submodule Gröbner bases there exist gradings γ which are homogeneity compatible with $*$.

Let R be polynomial ring with term order $\leq$, then a grading γ on R is called *compatible with* $\leq$ if for all $s, t \in T$ $\gamma(s) \leq \gamma(t) \implies s \leq t$. Then it can be shown, that there exists a term order $\leq'$ on R such that γ is compatible with $\leq'$. But note that $\leq'$ may not be $*$-compatible.

We are new prepared to state the main results of partial Gröbner bases.

Let $F \subset R$ be a subset of a polynomial ring with a term order $\leq$ and a grading γ which is compatible with $<$. For $d, e \in \mathbf{N}$ define

$$F(d, e) = \{f \in F \ : \ d \leq \gamma(f) < e\}$$

and $F(d, \infty) = \{f \in F \ : \ d \leq \gamma(f) \ \}$. For finite $F \subseteq R$ let the *d-restriction* of the reduction relation be defined as

$$\longrightarrow_{d,F} \ = \ \longrightarrow_F \ \cap \ R(0, d)^2.$$

The definition of partial Gröbner bases is as follows. Let $R = \mathbf{K}[X_1, \ldots, X_n]$ be a polynomial ring, with respect to an admissible ordering, over a field $\mathbf{K}$ and with a grading γ on

R which is compatible with $<$. Let $G \subset R$ be a finite subset of homogeneous polynomials of R and let $d \in \mathbf{N}$. If the reduction relation $\longrightarrow_{d,G}$ is confluent, then G is a *d-Gröbner base.*

Theorem 8.7.5 *Let $R = \mathbf{K}[X_1, \ldots, X_n]$ be a polynomial ring, with respect to an admissible ordering, over a field $\mathbf{K}$ and with a grading γ on R which is compatible with $<$. Let $G \subset R$ be a finite subset of homogeneous polynomials of R and let $d \in \mathbf{N}$. Then the following assertions are equivalent.*

1. *G is a d-Gröbner base.*
2. *For all $f \in \text{ideal}(G)(0,d)$, $f \longrightarrow^*_{d,G} 0$.*
3. *For all $0 \neq f \in \text{ideal}(G)(0,d)$, $f \longrightarrow_{d,G} f'$.*
4. *For all $h \in H(0,d) = \{SP(f,g) \ : \ f,g \in G, f \neq g\}(0,d)$, $h \longrightarrow^*_{d,G} 0$.*

By a theorem we know, that for any finite $F \subset R$ of homogeneous polynomials one can construct a d-Gröbner base G of $\text{ideal}(F)(0,d)$. And the proof of this theorem presents the Buchberger algorithm for constructing d-Gröbner bases.

8.7.3 Modules over Polynomial Rings

In this subsection we consider submodules of free modules over polynomial rings. Let $R = \mathbf{K}[X_1, \ldots, X_n]$ be a polynomial ring, with respect to an admissible ordering, over a field $\mathbf{K}$. Let $M = R^m$ be a free R-module with canonical basis $u_1, \ldots, u_m$. First we need to introduce some notation about generating sets of submodules.

Let M be an R-module. We say a sub module is *generated by a set* N, $N \subseteq M$ if it is of the form:

$$\text{modul}(N) = \{ \sum_{i \in \Lambda} r_i a_i \ : \ r_i \in R, a_i \in N, \Lambda \text{ finite } \},$$

If $N = \{a_1, \ldots, a_n\} \subseteq M$ is finite, then we write also $\text{modul}(a_1, \ldots, a_n)$ for $\text{modul}(N)$.

We can ask the questions: Let $N = \text{modul}(a_1, \ldots, a_k)$ be a submodule of M generated by $a_1, \ldots, a_k$,

> given $a \in M$, is $a \in N$?
> given a finite generating set of a submodule N, is there a canonical basis for N ?

The questions are answered using the method of partial Gröbner bases. To apply these we need some preparations.

Let $R = \mathbf{K}[X_1, \ldots, X_n]$, then we can embed the free module $R^m = \text{modul}(u_1, \ldots, u_m) = M$ into a polynomial ring with m additional variables:

$$\begin{aligned} R^m &\hookrightarrow R[Y_1, \ldots, Y_m] = \mathbf{K}[X_1, \ldots, X_n, Y_1, \ldots, Y_m] = R_{nm} \\ u_i &\mapsto Y_i \quad 1 \leq i \leq m. \end{aligned}$$

With the restriction of the multiplication in R_{nm} to polynomials from R with polynomials from R_{nm}. The restriction of the multiplication can be accomplished by a grading γ on

R_{nm}. We do this by defining $\gamma(X_i) = 0$ for $1 \leq i \leq n$, $\gamma(a) = 0$ for $0 \neq a \in \mathbf{K}$ and $\gamma(Y_j) = 1$ for $1 \leq j \leq m$. Then $\gamma(fg) = 0 = \gamma(f) + \gamma(g)$ for all $0 \neq f, g \in R$.

With this embbeding all results of partial reductions are available for free modules over polynomial rings. In particular for finite subsets N of R^m there exists a *submodule Gröbner base* G of modul(N), since by a theorem for homogeneous ideals there exists a partial 1-Gröbner base for N in R_{nm}.

Moreover there are several ways to order the variables, e.g. $X_i < Y_j$, $Y_j < X_i$ or according to some 'weights'. This way has been persued e.g. by [Möller, Mora 1986]. There is also a way proposed by [Armbruster, Kredel 1986] where the embedding into an extended polynomial ring is not required.

Furthermore we can find bases for the modules of syzygies for a submodule of R^m, since we can apply a partial version of the algorithm which generates the module of *syzygies* for a polynomial ring.

8.7.4 Algorithms and Examples

The main procedure `SYHC` computes a set of generators of the module of syzygies for a set of homogeneous commutative linear polynomial equations as sketched before. The main procedure `SYHNL` computes a set of generators of the left module of syzygies for set of non-commutative linear polynomial equations. The main procedure `MGB` computes a submodule Gröbner base from a set of generators of a submodule. The algorithms have been implemented by [Philipp 1991].

The computation takes place over the coefficient domain of the rational numbers. The term ordering may be any implemented term order. In case of a system of equations the problem is automatically reduced to a single equation problem by introducing new varaibles as sketched in step 3) before. In case of the non-commutative versions of the procedures, the commutator relations must be supplied as additional input. The programs are contained in the Modula-2 libraries `SYZHLP`, `SYZFUNC`, `SYZGB` and `SYZMAIN`.

Commutative Problems

In the following example we compute a generating set for the solutions to the single homogeneous equation

$$(3x^3 - w)Y_1 + (-1/23wz - x)Y_2 + (4wy^2 - x^2)Y_3 = 0.$$

The input consists first of the defining statements of the variables w, x, y, z, namely `VL:=LIST("w","x","y","z")` and `xxx:=DIPVDEF(VL)`. Then the input equation is read by the procedure `PM := MREAD(VL)`. It consists of a list of polynomial lists (defining the coefficient matrix) of distributive rational polynomials. The polynomial syntax is as described in section 7.4.3. The statement `PMWR(PM, VL)` prints the polynomial matrix to the current output stream. Then the statement `SY:=SYHC(PM, 1, 0)` requests the computation of the syzygies. The second argument `1` requests the printing of intermediate output and the third argument `0` indicates, that no reduced (intermediate) Gröbner base and no reduced set of syzygies is to be computed. Then again `PMWR(SY, VL)` prints the list of polynomial lists to the current output stream. Finally the statement `SYTHC(SY, PM, VL)` tests whether all elements of `SY` are indeed a solution to the original problem.

```
(* Linear homogeneous polynomial equation. *)

(* Polynomial matrix *)
VL:=LIST("w","x","y","z").  xxx:=DIPVDEF(VL).
PM := MREAD(VL).
(
 ( 3 x**3- w, -1/23 w z - x, 4 w y**2 - x**2 )
)
PMWR(PM, VL).

(* Syzygies: *)
SY:=SYHC(PM, 1, 0).
PMWR(SY, VL).

(* Test Syzygies *)
SYTHC(SY, PM, VL).
```

The following output shows first the input polynomial lists, then a generating set for the module of syzygies as computed by SYHC with a computing time of 0.484 seconds. Finally the result (0) of the test is shown.

```
MAS: (
(( 3 x**3 - w ), ( -1/23 w z - x ), ( 4 w y**2 - x**2 ))
)

Time: read = 16, eval = 484, print = 116, gc = 0.
MAS: (
(( -1/3 w z -23/3 x ), ( -23 x**3 +23/3 w ), 0),
(( -1/3 w y**2 +1/12 x**2 ),  0, ( 1/4 x**3 -1/12 w )),
( 0,  0, 0),
(( -1/276 x z -1/3 y**2 ), ( - x**2 y**2 +1/12 x ),
 ( -1/92 x**2 z -1/12 )),
(( 1/23 x**2 z**2 +4 x y**2 z ), ( 12 x**3 y**2 z - x**2 z ),
 ( 3/23 x**3 z**2 + x z )),
( 0,  0, 0),
( 0, ( -92 w y**2 +23 x**2 ), ( - w z -23 x )),
(( -1/276 w z**2 -1/12 x z ), ( - w x y**2 z +1/12 w z ),
 ( -1/92 w x z**2 -1/4 x**2 z )),
(( 1/23 w z**2 + x z ),
 ( 12 w x y**2 z - w z -1104 w y**4 +276 x**2 y**2 ),
 ( 3/23 w x z**2 -12 w y**2 z +3 x**2 z -276 x y**2 )),
( 0, ( -92 w y**4 +23 x**2 y**2 ), ( - w y**2 z -23 x y**2 )),
(( -1/276 w z -1/12 x ), ( - w x y**2 +1/12 w ),
 ( -1/92 w x z -1/4 x**2 )),
(( 1/23 w y**2 z**2 + x y**2 z ),
 ( 12 w x y**4 z - w y**2 z -1104 w y**6 +276 x**2 y**4 ),
 ( 3/23 w x y**2 z**2 -12 w y**4 z +3 x**2 y**2 z -276 x y**4 )),
( 0,  0, 0),
(( 1/23 x z**2 +4 y**2 z ), ( 12 x**2 y**2 z - x z ),
 ( 3/23 x**2 z**2 + z )),
(( 1/23 x y**2 z**2 +4 y**4 z ), ( 12 x**2 y**4 z - x y**2 z ),
 ( 3/23 x**2 y**2 z**2 + y**2 z ))
)

SYZYGIEN-TEST:  (0)
Time: read = 0, eval = 183, print = 0, gc = 0.
```

The next example is taken from [Buchberger 1985] (example number 6.17). It askes for a

generating set for the solutions to the single homogeneous equation

$$(x^2y - xy)Y_1 + (xy^2 - x^2)Y_2 + (x^3y - x^2y + x^3 - x^2)Y_3 = 0.$$

The input is as described in the previous example.

```
(* Linear homogeneous polynomial equation. *)
(* Example 6.17 from Buchberger (ch 6). ------------ *)

(* Polynomial matrix *)
VL:=LIST("x","y").  xxx:=DIPVDEF(VL).
PM := MREAD(VL).
(
 ( x**2 y - x y, x y**2 - x**2, x**3 y - x**2 y + x**3 - x**2 )
)
PMWR(PM, VL).

(* Syzygies: *)
SY:=SYHC(PM, 1, 0).
PMWR(SY, VL).

(* Test Syzygies *)
SYTHC(SY, PM, VL).
```

The following output shows first the input polynomial lists, then a generating set for the module of syzygies as computed by SYHC with a computing time of 0.134 seconds. Finally the result (0) of the test is shown.

```
MAS: (
((  x**2 y - x y ), (  x y**2 - x**2 ),
 (  x**3 y - x**2 y + x**3 - x**2 ))
)

Time: read = 0, eval = 134, print = 33, gc = 0.
MAS: (
(( - y - x ), (  x -1 ),  1 ),
( 0,  0, 0),
(( - x y - x ),  0,   y ),
((  y + x**2 ), ( - x**2 +1 ), (  y - x -1 )),
(( - x y**2 + y + x**2 + x ), ( - x**2 +1 ), (  y**2 - x -1 )),
(( - x y - x ),  0,   y )
)

SYZYGIEN-TEST:  (0)
Time: read = 0, eval = 50, print = 0, gc = 0.
```

Non-commutative Problems

In the following example from [Apel, Lassner 1988] a generating set for the solutions of the single homogeneous equation

$$Y_1 * (x_1x_2) + Y_2 * (x_1) = 0$$

is asked for in an enveloping algebra R of a Lie algebra defined by the commutator relations

$$\begin{aligned} x_2x_1 - x_1x_2 &= +x_3, \\ x_3x_2 - x_2x_3 &= +x_1, \\ x_3x_1 - x_1x_3 &= -x_2. \end{aligned}$$

The input consists first of the defining statements of the variables x_1, x_2, the commutator relations `T` and then the polynomial matrix. The commutator relation syntax is as described in section 7.7.1. The statement `SY:=SYHNL(PM, 1, 0, T)` requests the computation of a set of generators of the module of syzygies for a non-commutative linear polynomial equation. The first, second and third input is as described in the examples before and the fourth input `T` is the relation table of the commutator relations. The statement `OREC(p1, p2, q1, q2, T)` requests the computation of elements $q1$ and $q2 \in R$ such that $q1 * p1 = q2 * p2$.

```
(* Linear homogeneous polynomial equation. *)
(* Example 1 from Apel Lassner (1988). -------------- *)

(* Commutator relations: *)
VL:=LIST("x1","x2","x3").  xxx:=DIPVDEF(VL).
T := DIRLRD(VL).
(
 ( x2 ), ( x1 ), ( x1 x2 - x3 ),
 ( x3 ), ( x2 ), ( x2 x3 - x1 ),
 ( x3 ), ( x1 ), ( x1 x3 + x2 ),
)
PLWR(T, VL).

(* Polynomial matrix: *)
PM := NMREAD(VL, T).
(
 ( x1 x2, x1 )
)
PMWR(PM, VL).

(* Syzygies: *)
SY:=SYHNL(PM, 1, 0, T).
PMWR(SY, VL).

(* Test Syzygies *)
SYTHNL(SY, PM, VL, T).


(* The same example with the Ore condition algorithm. *)
P:=FIRST(PM). p1:=FIRST(P). p2:=FIRST(RED(P)).

(* Ore condition: *)
OREC(p1,p2, q1, q2, T).
Q:=LIST(LIST(q1,q2)).
PMWR(Q,VL).

(* Test Syzygies *)
SYTHNL(Q, PM, VL, T).
```

The corresponding output for the first problem shows first the input polynomial lists, then a generating set for the module of syzygies as computed by `SYHNL` with a computing time of 0.3 seconds. Finally the result `(0)` of the test is shown.

```
MAS: (  x2 ,   x1 , ( - x3 + x1 x2 ),   x3 ,   x2 ,
       (  x2 x3 - x1 ),   x3 ,   x1 , (  x1 x3 + x2 ))

MAS: (
(  x1 x2 ,   x1 )
)
```

```
Time: read = 0, eval = 300, print = 34, gc = 0.
MAS: (
( 0, 0),
(( 4 x1**2 x3 -4 x1**3 x2 -8 x1 x2 ),
 ( 4 x1**3 x2**2 +4 x1 x2**2 +4 x1**3 +4 x1 )),
(( - x1**2 -1 ), (  x1 x3 + x1**2 x2 )),
( 0, 0),
((  x1**2 +1 ), ( - x1 x3 - x1**2 x2 )),
(( -17/3 x1 x3 +8/3 x1**2 x2 -2/3 x2 ),
 ( 3 x3**2 +3 x1 x2 x3 -8/3 x1**2 x2**2 +10/3 x2**2 -17/3 x1**2 +1/3 ))
)

NEW RELATION =   x3**2  .*.   x1  = (  x1 x3**2 +2 x2 x3 - x1 )
NEW RELATION =   x2**2  .*.   x1  = ( -2 x2 x3 + x1 x2**2 + x1 )

N-SYZYGIEN-TEST:  (0)
Time: read = 0, eval = 133, print = 0, gc = 0.
```

The corresponding output for the second problem shows first two elements $q1, q2$ as computed by `OREC` with a computing time of 0.35 seconds. Finally the result `(0)` of the test is shown.

```
Time: read = 17, eval = 350, print = 0, gc = 0.
MAS: (
(( 4 x1**2 x3 -17/3 x1 x3 -4 x1**3 x2 +8/3 x1**2 x2 -8 x1 x2 -2/3 x2 ),
 ( 3 x3**2 +3 x1 x2 x3 +4 x1**3 x2**2 -8/3 x1**2 x2**2 +4 x1 x2**2
   +10/3 x2**2 +4 x1**3 -17/3 x1**2 +4 x1 +1/3 ))
)

N-SYZYGIEN-TEST:  (0)
Time: read = 0, eval = 116, print = 0, gc = 0.
```

Submodule Gröbner base

The following example is from Armbruster and is used in the classification of bifurkation problems [Armbruster, Kredel 1986]. It asks for a submodule Gröbner base in a $\mathbf{Q}[u, v, l]$ module given by the following set of generators.

$$
\begin{array}{rrrrr}
(& 1, & 2, & 0, & l^2 &), \\
(& 0, & l+3v, & 0, & u &), \\
(& 1, & 0, & 0, & l^2 &), \\
(& l+v, & 0, & 0, & u &), \\
(& l^2, & 0, & 0, & v &), \\
(& u, & 0, & 0, & vl+v^2 &), \\
(& 1, & 0, & l+3v, & 0 &), \\
(& l^2, & 0, & 2u, & v &), \\
(& 0, & 1, & l+v, & 0 &), \\
(& 0, & l^2, & u, & 0 &), \\
(& 0, & v, & ul^2, & 0 &), \\
(& 0, & vl+v^2, & u^2, & 0 &)
\end{array}
$$

The input consists first of the defining statements of the variables u, v, l: `VL:=LIST("u","v","l")` and `xxx:=DIPVDEF(VL)`. Then the input submodule base is read

by the procedure `PM := MREAD(VL)`. It consists of a list of polynomial lists (defining the elements which generate the submodule) of distributive rational polynomials. The polynomial syntax is as described in section 7.4.3. The statement `PMWR(PM, VL)` prints the polynomial matrix to the current output stream. The statement `QM:=MGB(PM, 0)` requests the computation of a submodule Gröbner base. The second argument `0` requests that no intermediate output is to be printed. Then again `PMWR(QM, VL)` prints the list of polynomial lists to the current output stream.

```
(* Submodule Groebner bases. *)
(* Example by Armbruster, bifurcation. *)

(* Polynomial matrix: *)
VL:=LIST("u","v","l").  xxx:=DIPVDEF(VL).
PM := MREAD(VL).
(
 (     ( 1 ),          ( 2 ),           0,       ( l**2 ) ),
 (         0,    ( l + 3 v ),           0,          ( u ) ),
 (     ( 1 ),              0,           0,       ( l**2 ) ),
 ( ( l + v ),              0,           0,          ( u ) ),
 (  ( l**2 ),              0,           0,          ( v ) ),
 (     ( u ),              0,           0, ( v l + v**2 ) ),
 (     ( 1 ),              0, ( l + 3 v ),              0 ),
 (  ( l**2 ),              0,     ( 2 u ),          ( v ) ),
 (         0,          ( 1 ),   ( l + v ),              0 ),
 (         0,       ( l**2 ),       ( u ),              0 ),
 (         0,          ( v ),  ( u l**2 ),              0 ),
 (         0, ( v l + v**2 ),    ( u**2 ),              0 )
)
PMWR(PM, VL).

(* Submodule Groebner base: *)
QM:=MGB(PM, 0).
PMWR(QM, VL).
```

The output shows first the repeated input tuples and then the computed module Gröbner base. The computing time of 28 seconds (IBM RS6000) is much more, than the computing time of 8 seconds (IBM 3081) reported by Armbruster. This might be due to the fact that different reduction strategies are used in both algorithms.

```
MAS: (
( 1 ,  2 ,  0,   l**2 ),
( 0, (  l +3 v ),  0,   u ),
( 1 ,  0,  0,   l**2 ),
((  l + v ),  0,  0,   u ),
(  l**2 ,  0,  0,   v ),
(  u ,  0,  0, (  v l + v**2 )),
( 1 ,  0, (  l +3 v ), 0),
(  l**2 ,  0,  2 u ,   v ),
( 0,  1 , (  l + v ), 0),
( 0,   l**2 ,   u , 0),
( 0,   v ,   u l**2 , 0),
( 0, (  v l + v**2 ),   u**2 , 0)
)

Time: read = 0, eval = 27967, print = 17, gc = 0.
MAS: (
( 0,  0,  0,   u ),
( 0,  0,  0, (  v**4 - v )),
```

```
( 0,  0,  0, (  v l + v**2 )),
( 0,  0,  0, (  l**3 + v**3 )),
( 0,  0,   u , 0),
( 0,  0,   v ,  -1/2 l**2 ),
( 0,  0,   l ,  1/2 l**2 ),
( 0,  1 ,  0, 0),
( 1 ,  0,  0,   l**2 )
)
```

8.8 Universal Gröbner Bases

Universal Gröbner bases are ideal bases, which are Gröbner bases with respect to **any** admissible term order. The problem is here to find algorithmically all different term orders on a given set of terms. We qoute from the introduction of [Weispfenning 1987a] who developed the theory. The implementation is based on an ALDES/SAC-2 program developed by [Belkahia 1992].

A *universal Gröbner basis* is a finite basis for a polynomial ideal that has the Gröbner property with respect to all admissible term-orders. Let R be a commutative polynomial ring over a field K, or more generally a non-commutative polynomial ring of solvable type over K (see [Kandri-Rody, Weispfenning 1988]). We show, how to construct and characterize left, right, two-sided, and reduced universal Gröbner bases in R. Moreover, we extend the upper complexity bounds in [Weispfenning 1986] to the construction of *universal* Gröbner bases. Finally, we prove the stability of universal Gröbner bases under specialization of coefficients. All these results have counterparts for polynomial rings over commutative regular rings (compare [Weispfenning 1987b]).

The construction of *Gröbner bases* initiated in [Buchberger 1970] has proved to be of great importance in the algorithmic theory of commutative polynomial ideals (compare [Buchberger 1985, Möller, Mora 1986]). Recently, the method and its applications have been extended to one- and two-sided ideals in certain non-commutative polynomial rings (see [Apel, Lassner 1988, Kandri-Rody, Weispfenning 1988]; compare also [Mora 1985, Mora 1986] for an analysis of the problem in general non-commutative polynomial rings).

In all these cases, the construction of a Gröbner basis depends on the choice of an *admissible order* on the set T of *terms*. In many considerations – and in all present implementations – this choice is restricted to a few basic types of admissible orders, such as *pure lexicographic* and *total degree orders*. The set of *all* such orders is, however, huge: It has the cardinality of the continuum. An algebraic characterization of all these orders has been given in [Robbiano 1985]; more recently, the author has given an alternative characterization in terms of linear forms with coefficients in a finitely generated ordered field (see [Weispfenning 1987]). It is particularly suitable for computational purposes.

As a consequence, only countably many admissible orders are decidable (as binary relations), while the majority (continuum many) are undecidable. So strictly speaking, the Buchberger method for constructing Gröbner bases is *algorithmic* only *relative to the admissible order* under consideration.

This raises the following natural *question*: What is the nature of the dependence of a Gröbner basis on an admissible order? In a recent paper [Schwartz 1988], N. Schwartz proves some remarkable facts concerning this question: He shows the existence of *universal*

Gröbner bases, i.e. of ideal bases that have the Gröbner property (*confluence* of the induced reduction relation) for *all* admissible orders. Moreover, he shows that the Gröbner property is locally constant under the Zariski topology on the space of admissible orders. This means that if G is a Gröbner basis with respect to the admissible order $<$ then it is also a Gröbner bases for all admissible orders $<'$ in some neighborhood of $<$. His results are *totally non-constructive, and apply to commutative polynomials only.*

The *purpose of this note* is to prove corresponding facts for the much more general situation of non-commutative polynomial rings of solvable type studied in [Apel, Lassner 1988] and [Kandri-Rody, Weispfenning 1988], and also for commutative polynomial rings over commutative regular rings (see [Weispfenning 1987b]). Moreover, our proofs yield *constructions* of universal Gröbner bases together with *upper complexity bounds* for these constructions. Our arguments are based on a use of *König's tree lemma* in combination with the results in [Weispfenning 1975] and [Weispfenning 1987], which in turn employ *logical compactness, a decision procedure for algebraically closed fields, the Tarski principle for real closed fields*, and the *polynomial time solvability of the restricted open hemisphere problem* (see [Garey, Johnson 1978], pp. 246, 339).

The most important *consequence* concerning the question posed above is the following: Given a finite ideal basis F(in any of the polynomial rings mentioned above); then all reduced (one- and two-sided) Gröbner bases G for the ideal generated by F, as well as a (reduced) universal Gröbner basis for this ideal, can be constructed using only finitely many decidable admissible orders, that can be found effectively.

For *commutative polynomials over fields* this fact was proved by a different method also in a recent preprint by [Mora, Robbiano 1988]; the special case of *two* variables was solved independently by [Schemmel 1987] as well. Both papers have came to the author's attention only at the EUROCAL'87 conference, June 1987, – well after completion of the research for of the present note and the submission of its results. (The note was first presented at a mathematical colloquium, University of München, January 1987.)

Our last result in this very general setting concerns the *stability* of Gröbner bases and universal Gröbner bases *under variation or specialization of the coefficients* in the given ideal basis: We provide a 'universal' construction of (universal) Gröbner bases for ideal bases with indeterminate coefficients, and show that the (universal) Gröbner bases resulting from specializations of these coefficients in a field will be the same within a Zariski-constructible set. For a very special case, a more specific result in this direction has been presented at EUROCAL'87 by [Gianni 1987] and [Kalkbrenner 1987].

This ends the authors intruduction and we turn now to the discussion of an example.

8.8.1 Example

The following is an example of an input data set for universal Gröbner bases. It can be read in with the known input command

```
IN("data set name").
```

from the MAS prompt.

The statement

```
UGBBIN().
```

calls the main routine of the universal Gröbner base package of Tijani Belkahia. All options and inputs are exactly the same as described in his "Diplomarbeit" [Belkahia 1992].

The input consists of several parts, which are discussed in the sequel.

1. The first part consists of the variable declaration '`(u0,u1,u2)`'. Note, that no term order is defined after the variable list.

2. The next input part consists of a list of polynomials which constitute the ideal base, for which an universal Gröbner base is requested. The syntax of such a list is that of an distributive rational polynomial list as described in section 7.4.3, 7.4.3, algorithm `DIRLRD`.

3. Then follows an optional part, which controls printing of intermediate results.

   Y Print information and intermediate results.

   N Do not print information and intermediate results.

4. The last input is the string '`EXEC UGB ..`'. Note, the required double period at the end of the string. This string requestes the computations and options as described in the following complete list.

   UGB compute an universal Gröbner base.

   PUGB compute an universal Gröbner base using precomputed linear forms of term orders.

   LF compute all linear froms, i.e. all term orders for the set of terms of the given input polynomials.

   PLF compute all linear froms, i.e. all term orders for the set of terms of the given input polynomials using precomputed linear forms of term orders.

   Further explanations of the options are described in [Belkahia 1992].

   The precomputed linear forms are contained in some subdirectory of the `examples` directory and are named `LINFORM.LADEi`, where `i` $= 1, 2, 3, 4$. The algorithm tries to predict the number of required linear forms which are read and used. But the prediction is in some cases too pessimistic, i.e. there are possibly much more precomputed linear forms than required for the construction of an universal Gröbner base. So the usage of this option can slowdown the computation. However, if the numbers of precomputed and required linear forms agree the computation will be much faster using the precomputed term orders (linear forms).

```
UGBBIN().
      (u0,u1,u2)

      (
      ( 2 u1**2 + u0**2 - u0 ),
      ( 2 u1 u2 + 2 u0 u1 - u1 ),
      ( 2 u2 + 2 u1 + u0 - 1 )
      )

      Y.
      EXEC UGB ..
```

The output is the same (except for letter case) as of the ALDES/SAC-2 version.

```
MAS: Die eingegebenen Polynome sind

          ( 2 u1**2 + u0**2 - u0 )
          ( 2 u1 u2 +2 u0 u1 - u1 )
          ( 2 u2 +2 u1 + u0 -1 )

Zwischenausgaben ... JA

Option ... UGB

Die Liste der Terme als ganzzahlige Tupel ist

((0,2,0),(0,0,2),(0,0,1),(1,1,0),(0,1,1),(0,1,0),(1,0,0),(0,0,0))

Projektionen ...
Dimension ... 2
Gradschranke dieser Dimension ist 8
Dimension ... 1
Gradschranke dieser Dimension ist 4
Die berechneten Linearformen sind 31
Ordne die Polynome nach den Linearformen
Reduktionsschritt
es sind 2 neue Terme entstanden

Projektionen ...
Dimension ... 2
Gradschranke dieser Dimension ist 6
Dimension ... 1
Gradschranke dieser Dimension ist 6
Die disjunkten Linearformen sind 59 Linearformen
Neue Linearformen entstanden
Ordne die Polynome nach den neuen Linearformen
Es ist nur ein neuer Term entstanden

Projektionen ...
Dimension ... 2
Gradschranke dieser Dimension ist 10
Dimension ... 1
Gradschranke dieser Dimension ist 6
Die disjunkten Linearformen sind 89 Linearformen
Neue Linearformen entstanden
Ordne die Polynome nach den neuen Linearformen
Es ist nur ein neuer Term entstanden

Projektionen ...
Dimension ... 2
Gradschranke dieser Dimension ist 8
Dimension ... 1
Gradschranke dieser Dimension ist 8
Die disjunkten Linearformen sind 96 Linearformen
Neue Linearformen entstanden
Ordne die Polynome nach den neuen Linearformen
Es ist nur ein neuer Term entstanden

Projektionen ...
Dimension ... 2
Gradschranke dieser Dimension ist 6
Dimension ... 1
Gradschranke dieser Dimension ist 8
```

```
Die disjunkten Linearformen sind 111 Linearformen
Neue Linearformen entstanden
Ordne die Polynome nach den neuen Linearformen

       ***********************************
              Universelle Groebnerbasis
       ***********************************

          ( 4 u0**2 u2 -4 u0 u2 +2/3 u0**2 -2/3 u0 )
          ( 2 u0 u1 +2 u0**2 -2 u0 )
          ( -6 u0**3 +10 u0**2 -4 u0 )
          ( 4 u0 u1 +4 u0**2 -4 u0 )
          ( 2 u2 +2 u1 + u0 -1 )
          ( 2 u1 u2 +2 u0 u1 - u1 )
          ( 2 u1**2 + u0**2 - u0 )
          ( -4 u2**2 -6 u0 u2 +4 u2 -2 u0**2 +3 u0 -1 )
          ( -2 u0 u2 + u0**2 - u0 )
          ( -12 u0**3 +20 u0**2 -8 u0 )
          ( 6 u0**2 u1 -4 u0 u1 )

7 garbage collections, 1615139 cells reclaimed, in 1068 milliseconds.
1699287 cells used, in 6023 milliseconds.
256512 cells allocated. Total time 7091 milliseconds.
```

This example needed less than 60 seconds on a PC 486/33/8 running OS/2 2.1. This is the end of the universal Gröbner base example.

8.9 Polynomial rings over Euclidean domains

The standard theory of Gröbner bases relies heavily on the fact, that the polynomial ring is a ring over a field. If this field condition is dropped to only euclidean domains or even principal ideal domains it can be shown that there still exist ideal bases with properties like that of Gröbner bases. For an introduction into the theory and further references see [Becker, Weispfenning 1993] chapter 10.1. The programs were implemented by [Mark 1992].

8.9.1 Examples

The algorithms for D- and E-Gröbner bases are located in the program modules '`DIPIDGB`' and '`DIPDDGB`'. The main programs for the computation of E-Gröbner bases are '`DIIPEGB`' for the base coefficient ring of the integers and '`DIDPEGB`' for the base coefficient ring of rational univariate polynomials. Both procedures take two inputs: first the polynomial list and second a trace flag. If the trace flag is equal to `0`, then no intermediate output is generated; if the trace flag is greater than `0`, say `n`, then after the `n`-th G- or S-polynomial has been computed statistics output is generated.

The first example shows the computation of an E-Gröbner base over the coefficient domain of integers, i.e. the computation takes place in the polynomial ring $\mathbf{Z}[X, Y]$. The integral polynomial list read function is '`PREADI`', the syntax is similar to the syntax of the '`PREAD`' function as described in 7.4.3. The input looks as follows.

```
BEGIN CLOUT("Computing integer E-Groebner base ... ");
      BLINES(1);
```

```
      P:=PREADI();
      CLOUT("The input polynomials are: ");
      BLINES(1);
      PWRITEI(P);
      Q:=DIIPEGB(P,1000);
      CLOUT("The E-Groebner base is: ");
      BLINES(1);
      PWRITEI(Q);
      CLOUT(" ... finished.");
      BLINES(1);
      END.

(X,Y) L
(
    ( Y**6 + X**4 Y**4 - X**2 Y**4 - Y**4 - X**4 Y**2
      + 2 X**2 Y**2 + X**6 - X**4 ),
    ( 2 X**3 Y**4 - X Y**4 - 2 X**3 Y**2 + 2 X Y**2
      + 3 X**5 - 2 X**3 ),
    ( 3 Y**5 + 2 X**4 Y**3 - 2 X**2 Y**3 - 2 Y**3
      - X**4 Y + 2 X**2 Y )
)
```

The following output shows first the input polynomials. Then it shows the intermediate statistics after each 1000 G- respectively S-polynomials, together with the effect of several criterions used to avoid unnecessary reductions. Finaly the resulting E-Gröbner base is displayed.

```
Computing integer E-Groebner base ...
The input polynomials are:
Polynomial in the variables: (X,Y)
Term ordering:  inverse lexicographical.
Polynomial list:
          (  Y**6 + X**4 Y**4 - X**2 Y**4 - Y**4 - X**4 Y**2
           +2 X**2 Y**2 + X**6 - X**4 )
          ( 2 X**3 Y**4 - X Y**4 -2 X**3 Y**2 +2 X Y**2 +3 X
          **5 -2 X**3 )
          ( 3 Y**5 +2 X**4 Y**3 -2 X**2 Y**3 -2 Y**3 - X**4
          Y +2 X**2 Y )

Number of computed G-polynomials: 1000
Cancelled due to criterion 1: 651
Cancelled due to top-D-reducibility: 332
New polynomials added to G: 17
Number of critical pairs in D: 47
Number of critical pairs in C: 81

Number of computed G-polynomials: 2000
Cancelled due to criterion 1: 1134
Cancelled due to top-D-reducibility: 845
New polynomials added to G: 21
Number of critical pairs in D: 129
Number of critical pairs in C: 16

Number of computed S-polynomials: 1000
Cancelled due to criterion 2: 0
Cancelled due to criterion 3: 441
New polynomials added to G: 43
Number of critical pairs in B: 1346
Number of critical pairs in C: 0
```

```
...

Number of computed S-polynomials: 7000
Cancelled due to criterion 2: 53
Cancelled due to criterion 3: 4455
New polynomials added to G: 75
Number of critical pairs in B: 260
Number of critical pairs in C: 0

Number of computed G-polynomials: 7875
Cancelled due to criterion 1: 5415
Cancelled due to top-D-reducibility: 2416
Number of computed S-polynomials: 7875
Cancelled due to criterion 2: 139
Cancelled due to criterion 3: 4956

Number of polynomials in G before reduction: 126
Number of polynomials in G after reduction: 8

The E-Groebner base is:

Polynomial in the variables: (X,Y)
Term ordering:  inverse lexicographical.
Polynomial list:
          ( 2 X Y**2 + X**13 -2 X**11 + X**9 +2 X**7 -2 X**3
           )
          (  X**15 -2 X**11 +4 X**9 )
          ( 4 X**11 Y -8 X**9 Y +8 X**7 Y )
          (  X**13 Y -2 X**9 Y +4 X**7 Y )
          (  Y**4 + X**12 -3 X**10 +3 X**8 - X**4 )
          ( 2 Y**3 -2 X**10 Y +3 X**8 Y -2 X**6 Y -2 X**2 Y
          )
          (  X**8 Y**2 + X**10 -2 X**8 )
          ( 2 X**12 -4 X**10 +4 X**8 )

Time: read = 3, eval = 11303, print = 1, gc = 1816.
```

The computing time is 113 seconds on an IBM RS6000-520.

The second example shows the computation of an E-Gröbner base over the coefficient domain of rational univariate polynomials. More precisely the computation takes place in the polynomial ring

$$\mathbf{Q}[B][S,T,Z,P,W].$$

The domain descriptor read and polynomial read functions '`ADDDREAD`' and '`DILRD`' are used as described in the section 7.8.1 on the arbitrary domain system. The input looks as follows.

```
BEGIN CLOUT("Computing domain rational polynomial E-Groebner base ...");
      BLINES(1);
      DP:=ADDDREAD(); ADDDWRIT(DP);
      V:=LIST("S","T","Z","P","W"); t:=DIPTODEF(2);
      P:=DILRD(V,DP);
      CLOUT("The input polynomials are: ");
      BLINES(1);
      DILWR(P,V);
      Q:=DIDPEGB(P,DP,10);
      CLOUT("The E-Groebner base is: "); BLINES(1);
      DILWR(Q,V);
```

```
      CLOUT(" ... finished.");
      BLINES(1);
      DIPTODEF(t);
      END.

RP(B)
(
( (45) P + (35) S - (165 B) - (36) ),
( (35) P + (40) Z + (25) T - (27) S ),
( (15) W + (25) S P + (30) Z - (18) T - (165 B**2) ),
( - (9) W + (15) T P + (20) S Z ),
( P W + (2) T Z - (11 B**3) ),
( (99) W - (11 B) S + (3 B**2) )
( (B**2 + 33/50 B + 2673/10000) )
)
```

The output is as follows. For an explanation see the previous example.

```
Computing domain rational polynomial E-Groebner base ...

RP(B) (* Rational Polynomial *)

The input polynomials are:

(  45  P + 35  S -( 165 B +36 ) )
(  35  P + 40  Z + 25  T - 27  S )
(  15  W + 25  S P + 30  Z - 18  T - 165 B**2  )
(  -9  W + 15  T P + 20  S Z )
(  P W + 2  T Z - 11 B**3  )
(  99  W - 11 B  S + 3 B**2  )
 (  B**2 +33/50 B +2673/10000 )

Number of computed G-polynomials: 10
Cancelled due to criterion 1: 10
Cancelled due to top-D-reducibility: 0
New polynomials added to G: 0
Number of critical pairs in D: 0
Number of critical pairs in C: 11

...

Number of computed S-polynomials: 100
Cancelled due to criterion 2: 75
Cancelled due to criterion 3: 0
New polynomials added to G: 6
Number of critical pairs in B: 5
Number of critical pairs in C: 0

Number of computed G-polynomials: 105
Cancelled due to criterion 1: 102
Cancelled due to top-D-reducibility: 1
Number of computed S-polynomials: 105
Cancelled due to criterion 2: 80
Cancelled due to criterion 3: 0

Number of polynomials in G before reduction: 15
Number of polynomials in G after reduction: 6

The E-Groebner base is:
```

```
 (  B**2 +33/50 B +2673/10000 )
(  15  W +( 19/8 B +3969/4000 ) )
(  45  P -( 155/2 B +1377/40 ) )
(  -1800  Z -( 2450 B +10287/10 ) )
(  -43659/20  T +( 538461/100 B -1178793/5000 ) )
(  S -( 5/2 B +9/200 ) )

Time: read = 7, eval = 471, print = 0, gc = 355.
```

The computing time is approximately 5 seconds on an IBM RS6000-520.

8.10 Buchberger algorithm with sugar strategy

The programs were implemented by [Rose 1995], we cite from this technical report:

"The basic ideas for the implementation of the Buchberger algorithm in the MAS module DIPAGB originate from the algorithms GRÖBNERNEW2 ([Becker, Weispfenning 1993], p. 232) and its subalgorithm UPDATE ([Becker, Weispfenning 1993], p. 230). Given a finite subset F of a multivariate polynomial ring $K[X_1, ..., X_n]$ over a computable field K GRÖBNERNEW2 finds a Gröbner basis G of the ideal in $K[X_1, ..., X_n]$ generated by F. The algorithm eliminates superfluous critical pairs according to Buchberger's criteria as implemented by Gebauer and Möller ([Gebauer, Möller 1988]). A call of the function `DIPAGB` with parameter F in the DIPAGB module performs the computation of G as given by GRÖBNERNEW2 with the exception that the algorithm stops with the ideal basis $G = \{1\}$ if a constant non-zero polynomial would be added to the computed ideal basis.

Besides the choice of the term order relative to which the Gröbner basis G is computed one of the main points of big influence on the time complexity of the algorithm is the *selection strategy*, i.e. the strategy to choose one of the critical pairs for the process of reduction. In the DIPAGB module Rev. 1.3 there are two selection strategies supported: the *normal strategy* that chooses a critical pair (f, g) if the least common multiple of the leading terms of f and g is minimal in the current term order, and the *normal with sugar strategy* that chooses a critical pair (f, g) if the sugar of the S-polynomial of f and g is minimal and, in order to break ties, if the least common multiple of the leading terms of f and g is minimal in the current term order among all critical pairs with the same sugar of their S- polynomial. The *sugar* S_f of a polynomial $f \in K[X_1, ..., X_n]$ is defined as follows: Let $W = (w_1, ..., w_n)$ be an n-tuple of rational numbers where w_i, $i = 1, ..., n$, is the weight of the variable X_i, and $f = \sum_{j=0}^{k} a_j X_1^{\nu_{j,1}} \cdot ... \cdot X_n^{\nu_{j,n}} \in K[X_1, ..., X_n]$. Then the *rational-weighted total degree* $\deg_W(f)$ of f is defined by

$$\deg_W(f) := \max_{0 \le j \le k} \sum_{i=1}^{n} w_i \, \nu_{j,i}. \tag{8.1}$$

Furthermore, for the initial $f \in F$, let $S_f = \deg_W(f)$.
If $g, h \in K[X_1, ..., X_n]$ are polynomials with sugar S_g, S_h and $t \in K[X_1, ..., X_n]$ is a term, then $S_{t \cdot g} = \deg_W(t) + S_g$ and $S_{g+h} = \max\{S_g, S_h\}$.

The motivation for the sugar definition of polynomials and a comprehensive description in the special case $W = (1, ..., 1)$ can be found in [Giovini *et al.* 1991].

8.10.1 DIPAGB procedures in the interpreter

The main data structures used in the DIPAGB module are given by the MAS system. The `DIPAGB` function takes a list F of polynomials in distributive representation and produces a list G of polynomials in distributive representation. Whenever the normal strategy is used, this data structure for the polynomials handled during the Gröbner basis computation is sufficient. The normal with sugar strategy option, however, makes it necessary to carry along the sugar with each polynomial. This fact leads to a new data structure called *extended distributive polynomial*. In case of the normal strategy an extended distributive polynomial simply is the same as a distributive polynomial; in case of the normal with sugar strategy an extended distributive polynomial is a pair (f, S_f) of a polynomial f in distributive representation and its sugar S_f.

Furthermore it is advantageous in practice to store frequently used data of a critical pair of polynomials together with each critical pair. Accordingly in case of the normal strategy an *extended critical pair* is a triple (f, g, lcm) of two (extended) distributive polynomials f, g and the least common multiple lcm of the leading terms of f and g, whenever (f, g) defines a critical pair of polynomials.

Since in case of the normal with sugar strategy it is also necessary to keep the sugar of the S-polynomial of each critical pair in mind, an *extended critical pair* is then given by a quadruple $((f, S_f), (g, S_g), lcm, S_{f,g})$ where (f, g) is a critical pair, lcm is the least common multiple of the leading terms of f and g and $S_{f,g}$ is the sugar of the S-polynomial of f and g.

In the DIPAGB module Rev. 1.3 tuples of data are represented by lists of appropriate length. The names of procedures which handle with extended distributive polynomials, start with "`EDIP`", procedures which take extended critical pairs are denoted "`ECP`...".

The function `DIPAGB`

The main function in the DIPAGB module is the `DIPAGB` function. As described above, it takes one input parameter, namely a finite list F of polynomials in distributive representation. Since `DIPAGB` works over arbitrary domains, the coefficients of these polynomials are allowed to be taken from every domain supported in the arbitrary domain system of MAS. Look at the MAS documentation for details on the arbitrary domain system. The output of `DIPAGB` is a list G of polynomials in distributive representation such that G is a Gröbner basis of the ideal generated by F. Whenever the reduced and sorted Groebner basis is to be calculated, the user should subsequently apply the `DIPLIR` function from the DIPADOM module.

The option set procedures

The following option set procedures are at the user's disposal:

```
SetDIPAGBStrategy
SetDIPAGBVariableWeight
SetDIPAGBTraceFlag
SetDIPAGBOptions
```

`SetDIPAGBStrategy(`st`)`, $st \in \{0,1\}$, chooses the selection strategy ($st = 0$: normal, $st = 1$: normal with sugar). The selection strategy is set to 0 by default. The MAS system will use the chosen strategy for each run of `DIPAGB` until a new selection strategy is set by the user.

With the command `SetDIPAGBVariableWeight(`W`)` the variable weight list used during any computation of the rational-weighted total degree of a polynomial f in n variables in case of the normal with sugar strategy is set. $W = (W_1, ..., W_n)$ is a list of rational numbers where n is the length of the variable list `VALIS` and W_i, $1 \leq i \leq n$, is the weight of the ith variable in `VALIS`. If no variable weight list is set by the user, the default weight list $(1, ..., 1)$ is used in all degree calculations while the Gröbner basis G is computed. The variable weight list remains W until it is reset by the user. If the actual variable weight list is different from the default weight list and of a length different to the length of `VALIS` the message

```
DIPAGB: no valid variable weight list
```

will occur.

The *trace flag*, a non-negative integer level for interactive documentations, is very useful for solving diagnostic problems. With the `SetDIPAGBTraceFlag(`tf`)` command, $tf \in \{0, 1, 2, ...\}$, you can set the trace flag to tf. The following operations are documented in the output stream:

- $tf = 0$: No computation steps
- $tf \geq 1$: The total execution time and the number of cells needed to perform the Gröbner basis computation
- $tf \geq 2$: The number and execution time of the normalform computations
- $tf \geq 3$: The changes of selection strategy, variable weight and trace flag
- $tf \geq 4$: The polynomials to be reduced to normalform
- $tf \geq 5$: The results of the normalform computations
- $tf \geq 6$: The results of the rational-weighted total degree calculations
- $tf \geq 7$: The insert operations of an extended critical pair into the extended critical pair list
- $tf \geq 8$: The extensions of the distributive polynomials and the critical pairs of distributive polynomials
- $tf \geq 9$: The results of all S-polynomial computations

The default value of the trace flag is 0. The trace flag is unchanged until it is reset by the user.

Finally, all above options can be set with the single `SetDIPAGBOptions(`O`)` command where O is a list of options. Let O be the list $(O_1, ..., O_L)$. Then `SetDIPAGBOptions(`O`)` works as a macro of the following operations:

if $L \geq 1$ **then** SetDIPAGBTraceFlag(O_1) **end**;
if $L \geq 2$ **then** SetDIPAGBStrategy(O_2) **end**;
if $L \geq 3$ **then** SetDIPAGBVariableWeight(O_3) **end**;

The option write procedures

In analogy to section 8.10.1 the following option write procedures exist:

```
WriteDIPAGBStrategy
WriteDIPAGBVariableWeight
WriteDIPAGBTraceFlag
WriteDIPAGBOptions
```

With these commands the appropriate current options can be shown in the output stream. Each of these procedures takes no parameters.

Further procedures

One procedure of the DIPAGB module available in the MAS interpreter remains to be mentioned. It's the operation LRNWRIT(L) (list of rational numbers write) where L is a finite list of rational numbers. With this procedure for example a variable weight list can be shown in the output stream.

8.10.2 Examples

The following example demonstrates the interactive documentation handling during a run of DIPAGB when using trace flag 9.

```
DomainDescriptor := ADDDREAD(). RN.
VariableList := LIST("X","Y").
V := DIPVDEF(VariableList).
TermOrder := DIPTODEF(2).

F := DILRD(VariableList,DomainDescriptor).
     ( ( 3/2 X**2 Y - 4 ) ( - X Y + 2/5 X ) )
DILWR(F,VariableList).

SetDIPAGBTraceFlag(9).
   WX:=RNREAD(). 1/2.
   WY:=RNREAD(). 1.
SetDIPAGBVariableWeight(LIST(WX,WY)).
SetDIPAGBStrategy(1).

DILWR(DIPAGB(F),VariableList).

   ANS: (8 0 -1)

   ANS: ((57) (59))

   ANS: ((57) (59))

   ANS: 2
```

```
ANS: (((1 2) (8 (3 2) -1) (0 0) (8 (-4 1) -1)) ((1 1)
     (8 (-1 1) -1) (0 1) (8 (25) -1)))

( 3/2 X**2 Y -4 )
( -1 X Y +2/5 X )

New documentation level: 9
...
New variable weight list: (1/2,1)

New strategy: 1  (= normal with sugar)

The rational-weighted total degree of the polynomial
  ( 3/2 X**2 Y -4 )
w.r.t. the variable list
  (X,Y)
and the variable weight list
  (1/2,1)
is 2.

Extending the polynomial
    ( 3/2 X**2 Y -4 )
with sugar 2.

The rational-weighted total degree of the polynomial
  ( -1 X Y +2/5 X )
w.r.t. the variable list
  (X,Y)
and the variable weight list
  (1/2,1)
is 3/2.

Extending the polynomial
    ( -1 X Y +2/5 X )
with sugar 3/2.

Extending the following critical pair:
1st polynomial: ( -1 X Y +2/5 X )
    with sugar 3/2
2nd polynomial: ( 3/2 X**2 Y -4 )
    with sugar 2
LCM of their leading terms:  X**2 Y
Sugar of their S-polynomial: 2

Inserting the following extended critical pair:
1st polynomial: ( -1 X Y +2/5 X )
    with sugar 3/2
2nd polynomial: ( 3/2 X**2 Y -4 )
    with sugar 2
LCM of their leading terms:  X**2 Y
Sugar of their S-polynomial: 2

==> New extended critical pair list:
1st critical pair:
1st polynomial: ( -1 X Y +2/5 X )
    with sugar 3/2
2nd polynomial: ( 3/2 X**2 Y -4 )
    with sugar 2
LCM of their leading terms:  X**2 Y
Sugar of their S-polynomial: 2
```

```
Computing the S-polynomial of the following two distributive polynomials:
f = ( -1 X Y +2/5 X )
with sugar 3/2.

g = ( 3/2 X**2 Y -4 )
with sugar 2.

==> Spol(f,g) = ( 3/5 X**2 -4 )
with sugar 2.

1st polynomial to reduce to normalform:
  ( 3/5 X**2 -4 )

The normalform is
  ( 3/5 X**2 -4 )
with sugar 2.

...

Extending the following critical pair:
1st polynomial: ( 4 Y -8/5 )
    with sugar 3
2nd polynomial: ( -1 X Y +2/5 X )
    with sugar 3/2
LCM of their leading terms:  X Y
Sugar of their S-polynomial: 7/2

Inserting the following extended critical pair:
1st polynomial: ( 4 Y -8/5 )
    with sugar 3
2nd polynomial: ( -1 X Y +2/5 X )
    with sugar 3/2
LCM of their leading terms:  X Y
Sugar of their S-polynomial: 7/2

==> New extended critical pair list:
1st critical pair:
1st polynomial: ( 4 Y -8/5 )
    with sugar 3
2nd polynomial: ( -1 X Y +2/5 X )
    with sugar 3/2
LCM of their leading terms:  X Y
Sugar of their S-polynomial: 7/2

2nd critical pair:
1st polynomial: ( 4 Y -8/5 )
    with sugar 3
2nd polynomial: ( 3/5 X**2 -4 )
    with sugar 2
LCM of their leading terms:  X**2 Y
Sugar of their S-polynomial: 4

Computing the S-polynomial of the following two distributive polynomials:
f = ( 4 Y -8/5 )
with sugar 3.

g = ( -1 X Y +2/5 X )
with sugar 3/2.

==> Spol(f,g) = 0
```

```
with sugar 7/2.

3rd polynomial to reduce to normalform:
  0

The normalform is
  0
with sugar 7/2.

3 normalform computations in 60 milliseconds.

Total time: 770 milliseconds.
     Cells: 781.

( 3/5 X**2 -4 )
( 4 Y -8/5 )
```

Hairer, Runge-Kutta

This example is taken from [Böge *et al.* 1986].

Domain: `RN`
Variable list: `LIST("C2","C3","B3","B2","B1","A21","A32","A31")`

Polynomial list:
```
( ( + C2 - A21 )
  ( + C3 - A31 - A32 )
  ( + B1 + B2 + B3 - 1)
  ( + B2 C2 + B3 C3 - 1/2 )
  ( + B2 C2**2 + B3 C3**2 - 1/3 )
  ( + B3 A32 C2 - 1/6 ) )
```

Results with the inverse lexicographical term order `INVLEX`:

Strategy	Normalform computations		Time	Space
	total number	time in ms	in ms	in cells
normal	14	430	620	17015
normal with sugar, variable weight list $(0,...,0)$	14	550	750	19723
normal with sugar, variable weight list $(1,...,1)$	17	670	970	26591
DIPGB			640	13889

Results with the Buchberger term order `REVITDG`:

Strategy	Normalform computations		Time	Space
	total number	time in ms	in ms	in cells
normal	15	180	360	14541
normal with sugar, variable weight list $(0,...,0)$	15	190	480	16626
normal with sugar, variable weight list $(1,...,1)$	15	220	540	19409
DIPGB			620	15880

For the complete set of examples with many more comparisons see the technical report [Rose 1995].

8.11 Buchberger algorithm with polynomial factorization

The programs were implemented by [Pfeil 1994]. The module DIPDCGB has been extented to allow several ways to compute factored Gröbner bases.

8.11.1 Optionen

In diesem Abschnitt werden vier Optionen erläutert, die beiden Varianten gemeinsam sind, eine Option für die Variante nach [Melenk *et. al.* 1989] und zwei Optionen für die Variante nach [Gräbe, Lassner]. Diese Optionen können durch die Prozeduren *SetTraceLevel, SetDecompProc, SetUpdateProc, SetVarOrdOpt, SetFacSugar, SetReduceExp, SetBranchProc und SetDCGBopt* gesetzt werden (vergl. Abschnitt 8.11.2).

Trace-Level

Das Trace-Level gibt an, wieviel Information während der Berechnung ausgegeben wird. Der Wert kann zwischen 0 und 4 (jeweils einschließlich) liegen, wobei ein höherer Wert mehr Ausgaben bedeutet und jeweils die Ausgaben der niedrigeren Trace-Level-Werte mit einschließt.

Faktorisierungs-Prozedur

Statt der in Definition [Pfeil 1994] eingeführten Faktorisierung – im folgenden als vollständige Faktorisierung bezeichnet – kann auch die quadratfreie Zerlegung (vgl. [Becker, Weispfenning 1993], S.100) benutzt werden. Die quadratfreie Zerlegung (Prozedur *DIPSFF*, vgl. [Pfeil 1994], Anhang A.1) hat gegenüber der vollständigen Faktorisierung (Prozedur *DIPFAC*, vgl. [Pfeil 1994], Anhang A.1) i.a. den Vorteil der geringeren Rechenzeit welcher durch den Nachteil der geringeren Zerlegung des Problems relativiert wird.

Aktualisierungs-Prozedur

Durch diese Option wird die Prozedur zur Aktualisierung der kritischen Paare und der Polynommenge gewählt. Wird diese Option nicht angegeben, so wird standardmäßig die Prozedur *UPDATE* aus dem Modul *DIPAGB* von K. Rose benutzt. Soll hierzu eine andere Prozedur benutzt werden können, muß diese in dem Modul *DIPDCGB* importiert werden und die Prozeduren *SetUpdateProc* und *SetDCGBopt* müssen dementsprechend erweitert werden.

Optimierung der Variablenordnung

Die Variablenordnung gibt die Ordnung auf den Variablen an und ist von der Termordnung zu unterscheiden. Zur Optimierung wird die Variablenordnung so geändert, daß die Faktorisierung möglichst schnell berechnet werde kann. Um die ursprüngliche Variablenordnung wiederherzustellen muß dann die Variablenordnung zurück geändert werden. Es wurden zwei Arten der Optimierung der Variablenordnung implementiert:

1. Optimierung der Variablenordnung zu Beginn des Algorithmus und Wiederherstellung der ursprünglichen Variablenordnung am Ende des Algorithmus, im folgenden mit VOOB bezeichnet.
2. Optimierung der Variablenordnung unmittelbar vor jeder Faktorisierung und Wiederherstellung der ursprünglichen Variablenordnung unmittelbar nach der Faktorisierung, im folgenden mit VOOF bezeichnet.

Bei VOOB ist der Verwaltungsaufwand gegenüber VOOF wesentlich geringer, aber die erhaltenen Polynommengen bilden i.a. keine Gröbner-Basis bzgl. der ursprünglichen Variablenordnung. Die Option VOOB wurde in der Prozedur *GroebnerBases1* bzw. *GroebnerBases2* implementiert, während die Option VOOF in den Prozeduren zur Faktorisierung implementiert wurde. Da sich VOOB und VOOF nicht gegenseitig beeinflußen, können auch beide gleichzeitig benutzt werden.

Faktor-Sugar

Diese Option bezieht sich nur auf die Prozedur *GroebnerBases1* und gibt an, ob die Sugar-Strategie (vergl. [Giovini *et al.* 1991]) benutzt wird und wenn ja, welchen Sugar die bei einer Faktorisierung entstehenden Faktoren bekommen. Entweder ist der neue Sugar der des Ausgangspolynoms oder der Totalgrad des Faktors.

Reduktions-Exponent

Diese Option bezieht sich nur auf die Prozedur *GroebnerBases2* und gibt an, von welcher Potenz u^e in Prozedur *TEII* eine Normalform berechnet wird (vergl. [Pfeil 1994], Abschnitt 4.2). Wenn $e = 1$ ist, wird nur die Optimierung nach [Pfeil 1994], Satz 3.2.11 verwendet, für $e > 1$ wird entsprechend [Pfeil 1994], Satz 3.2.13 verfahren.

Verzweigungs-Prozedur

Diese Option bezieht sich nur auf die Prozedur *GroebnerBases2* und gibt an, welche Prozedur benutzt wird, um die neuen Zweige aus den Faktoren zu konstruieren. Die Prozedur *SSCO* konstruiert für jede nichtleere Teilmenge der Faktoren einen neuen Zweig (vergl. [Pfeil 1994], Abschnitt 3.2), während die Prozedur *EQIEQ* nur einen neuen Zweig je Faktor konstruiert (vergl. [Gräbe, Lassner]).

In *EQIEQ* werden die Gleichungen G_i und Ungleichungen U_i für einen Zweig i aus den Faktoren $F = \{f_1, \ldots, f_r\}$ folgendermaßen konstruiert:

1. $G_1 := \{f_1\},\ U_1 := \emptyset$
2. $G_i := \{f_i\},\ U_i := \{f_1, \ldots, f_{i-1}\},\ 2 \leq i \leq r$

8.11.2 Benutzung der Prozeduren im MAS-Interpreter

In diesem Abschnitt wird die Benutzung der vorgestellten Prozeduren erläutert. Im weiteren sind mit Polynomen stets Polynome in distributiver Darstellung gemeint. Für einen Benutzer des MAS-Interpreters sind die folgenden Prozeduren zugänglich:

Prozedur *GroebnerBases1*

Diese Prozedur implementiert, wie bereits in den vorangegangenen Kapiteln erläutert, die Variante nach [Melenk *et. al.* 1989] und wird im MAS-Interpreter mit *GB1* aufgerufen. Diese Prozedur hat als Eingabeparameter eine Liste von Polynomen F und liefert als Ergebnis eine Liste von Polynommengen $\{F_1, \ldots, F_k\}$.

Prozedur *GroebnerBases2*

Diese Prozedur implementiert, wie bereits in den vorangegangenen Kapiteln erläutert, die Variante nach [Gräbe, Lassner] und wird im MAS-Interpreter mit *GB2* aufgerufen. Diese Prozedur hat als Eingabeparameter zwei Listen von Polynomen F und U und liefert als Ergebnis eine Liste von Paaren von Polynommengen $\{(F_1, U_1), \ldots, (F_k, U_k)\}$.

Prozedur *SetTraceLevel*

Diese Prozedur setzt die in Abschnitt 8.11.1 erläuterte Option. Hierzu wird als Parameter eine ganze Zahl angegeben, deren Wert zwischen 0 und 4 (jeweils einschließlich) liegt. Dabei bedeutet:

0 : (default) keine Ausgabe, außer mit Option VOOB, um darauf hinzuweisen, daß das Ergebnis i.a. keine Gröbner-Basis darstellt

> 0 : Ausgabe der Rechenzeit und der Ergebnisse nach der Berechnung

> 1 : Ausgabe von Meldungen über den Berechnungsbaum:
Anzahl der abgeschnittenen Zweige, "cancel factor", "cancel branch", "groebner base", "branch w.o. zeros".

> 2 : Ausgabe von S-Polynomen und Normalformen

> 3 : Ausgabe der Parameter der Prozeduren *RECGB1, RECGB2, REC1, REC2*, der Rechenzeit und der Gröbner-Basen während der Berechnung

Prozedur *SetDecompProc*

Diese Prozedur setzt die in Abschnitt 8.11.1 erläuterte Option. Hierzu wird als Parameter eine ganze Zahl angegeben, deren Wert zwischen 1 und 2 (jeweils einschließlich) liegt. Dabei bedeutet:

1 : (default) vollständige Faktorisierung (mit Prozedur *DIPFAC*)

2 : quadratfreie Zerlegung (mit Prozedur *DIPSFF*)

Prozedur *SetUpdateProc*

Diese Prozedur setzt die in Abschnitt 8.11.1 erläuterte Option. Hierzu wird als Parameter eine ganze Zahl angegeben, deren Wert 1 ist. Dabei bedeutet:

1 : (default) Update nach K. Rose (mit Prozedur *UPDATE*)

Prozedur *SetVarOrdOpt*

Diese Prozedur setzt die in Abschnitt 8.11.1 erläuterte Option. Hierzu wird als Parameter eine ganze Zahl angegeben, deren Wert zwischen 0 und 3 (jeweils einschließlich) liegt. Dabei bedeutet:

0 : (default) keine Optimierung der Variablenordnung

1 : VOOB

2 : VOOF

3 : VOOB und VOOF

Prozedur *SetFacSugar*

Diese Prozedur setzt die in Abschnitt 8.11.1 erläuterte Option. Hierzu wird als Parameter eine ganze Zahl angegeben, deren Wert zwischen 0 und 2 (jeweils einschließlich) liegt. Dabei bedeutet:

0 : (default) keine Sugar-Strategie

1 : Faktoren erhalten als Sugar ihren Totalgrad

2 : Faktoren erhalten als Sugar den Sugar des faktorisierten Polynoms

Prozedur *SetReduceExp*

Diese Prozedur setzt die in Abschnitt 8.11.1 erläuterte Option. Hierzu wird als Parameter eine ganze Zahl angegeben, deren Wert größer gleich 1 ist. Dabei bedeutet:

1 : (default) Optimierung nach [Pfeil 1994], Satz 3.2.11

> 1 : Optimierung nach [Pfeil 1994], Satz 3.2.13 mit diesem Wert als Exponent e

Prozedur *SetBranchProc*

Diese Prozedur setzt die in Abschnitt 8.11.1 erläuterte Option. Hierzu wird als Parameter eine ganze Zahl angegeben, deren Wert zwischen 1 und 2 (jeweils einschließlich) liegt. Dabei bedeutet:

1 : (default) ein Zweig für jede nichtleere Teilmenge der Faktoren (mit Prozedur *SSCO*)

2 : ein Zweig für jeden Faktor (mit Prozedur *EQIEQ*)

Prozedur *SetDCGBopt*

Mit dieser Prozedur können alle, in den vorangegangenen Abschnitten erläuterten Optionen gesetzt werden. Hierzu wird als Parameter eine Liste von ganzen Zahlen angegeben. Diese Liste muß mindestens ein Element haben und kann höchstens sieben Elemente haben. Hierbei werden diese Zahlen in folgender Reihenfolge den Optionen zugeordnet:

1. Trace-Level
2. Nr. der Faktorisierungs-Prozedur
3. Nr. der Aktualisierungs-Prozedur
4. Optimierung der Variablenordnung
5. Faktor-Sugar
6. Reduktions-Exponent
7. Nr. der Verzweigungs-Prozedur

Prozedur *WriteDCGBopt*

Mit dieser Prozedur können die aktuellen Werte der Optionen ausgegeben werden. Diese Prozedur hat keinen Parameter.

8.11.3 Beispiel

Das Beispiel und die Beschreibung (mit leichten Anpassungen) sind [Pfeil 1994] entnommen; weitere Beispiele sind ebendort zufinden.

In diesem Abschnitt wird für ein Beispiel die Rechenzeit und Ergebnisse für jeweils mehrere Optionen angegeben. Hierbei gibt die Zahl hinter TO die gewählte Termordnung an (vgl. 7.5). Zum Vergleich ist außerdem jeweils die Zeit für den Buchberger-Algorithmus ohne Faktorisierung (DIPGB, vgl. 7.8.1) angegeben. In der Tabelle gibt die obere Zeile eines Tabelleneintrages die Rechenzeit an. In der unteren Zeile gibt der Kleinbuchstabe an, welche Lösungsmenge zu diesem Eintrag gehört, wobei verschiedene Lösungsmengen, deren Polynome sich nur durch Einheiten unterscheiden und daher dieselben Nullstellen beschreiben, zusammengefaßt wurden. Die Zahl in der unteren Zeile gibt in den Spalten GB1 die Anzahl der durch Algorithmus `GB1` (nach [Melenk *et. al.* 1989]) berechneten F_i, in den Spalten GB2 die Anzahl der durch Algorithmus `GB2` (nach [Gräbe, Lassner]) berechneten (F_i, U_i) und in der Spalte DIPGB die Anzahl der Polynome in der mit DIPGB berechneten Lösungsmenge an. Alle Beispiele wurden mit der Standardeinstellung von 4 MByte für den Speicherplatz berechnet.

Dieses Beispiel wurde [Buchberger 1970] entnommen. Die Polynommenge F_1 besteht aus drei Polynomen in drei Variablen $X > Y > Z$:

$$\begin{aligned} F_1 \quad = \{ \quad & X^3YZ - XZ^2, \\ & XY^2Z - XYZ, \\ & X^2Y^2 - Z \quad \} \end{aligned}$$

Lösungsmengen (vgl. Tabelle 8.11.3):

		vollständige Fakt.		quadratfreie Zerl.		
		GB1	GB2	GB1	GB2	DIPGB
TO=2		690	820	230	290	140
	ohne	a 3	f 5	b 2	d 3	c 5
		140	210	100	120	
	VOOB	a 3	g 5	b 2	e 3	
		180	270	200	220	
	VOOF	a 3	f 5	b 2	d 3	
		100	140	90	120	
	beide	a 3	g 5	b 2	e 3	
TO=8		650	800	240	290	120
	ohne	a 3	f 5	b 2	d 3	c 5
		260	390	180	240	
	VOOB	a 3	f 5	b 2	d 3	
		180	280	190	230	
	VOOF	a 3	f 5	b 2	d 3	
		200	280	190	230	
	beide	a 3	f 5	b 2	d 3	

Table 8.15: Ergebnisse zu F_1, Zeiten in ms

a) $\{X, Z\},\ \{X^2 - Z, Y - 1\},\ \{Z, Y\}$

b) $\{Y - 1, X^2 - Z\},\ \{XY, Z\}$

c) $\{YZ^2 - Z^2, XY^2Z - XYZ, X^2Z^2 - Z^3, X^2YZ - Z^2, X^2Y^2 - Z\}$

d) $(\{Y - 1, X^2 - Z\}, \{Z\}),\ (\{Y - 1, Z, X\}, \{\}),\ (\{XY, Z\}, \{Y - 1\})$

e) $(\{Y - 1, X^2 - Z\}, \{XY\}),\ (\{Y - 1, Z, X\}, \{\}),\ (\{XY, Z\}, \{Y - 1\})$

f) $(\{Y, Z\}, \{X, Y - 1\}),\ (\{X, Y, Z\}, \{Y - 1\}),\ (\{X, Z\}, \{Y, Y - 1\}),$
$(\{Y - 1, X, Z\}, \{Y\}),\ (\{X^2 - Z, Y - 1\}, \{X, Y, Z\})$

g) $(\{Y, Z\}, \{X, Y - 1\}),\ (\{X, Y, Z\}, \{Y - 1\}),\ (\{X, Z\}, \{Y, Y - 1\}),$
$(\{Y - 1, X, Z\}, \{Y\}),\ (\{X^2 - Z, Y - 1\}, \{X, Y\})$

Katsura, Laurent Series case 3

Dieses Beispiel stammt aus [Böge *et al.* 1986].

```
dp:=ADDDREAD(). RN.
V:=LIST("Z","Y","X","W"). xx:=DIPVDEF(V).
```

```
F10:=DILRD(V,dp).
( ( W**2 - W + 2 X**2 + 2 Y**2 + 2 Z**2)
( 2 W X + 2 X Y + 2 Y Z - X )
( 2 W Y + X**2 + 2 X Z - Y )
( W + 2 X + 2 Y + 2 Z - 1 ) )

DILWR(F10,V).
SetTraceLevel(2).
SetFacSugar(1).
SetDecompProc(2).
SetVarOrdOpt(3).
WriteDCGBopt.

gbl:=GB1(F10).
```

Die Ausgabe sieht wie folgt aus.

```
ANS: (8 0 -1)
ANS: ()
ANS: ((61) (59) (57) (55))
ANS: ((61) (59) (57) (55))
ANS: (((2 0 0 0) (8 (1 1) -1) (1 0 0 0) (8 (-1 1) -1) (0 2 0 0)
(8 (2 1) -1) (0 0 2 0) (8 (2 1) -1) (0 0 0 2) (8 (2 1) -1))
((1 1 0 0) (8 (2 1) -1) (0 1 1 0) (8 (2 1) -1) (0 1 0 0)
(8 (-1 1) -1) (0 0 1 1) (8 (2 1) -1)) ((1 0 1 0) (8 (2 1) -1)
(0 2 0 0) (8 (1 1) -1) (0 1 0 1) (8 (2 1) -1) (0 0 1 0)
(8 (-1 1) -1)) ((1 0 0 0) (8 (1 1) -1) (0 1 0 0)
(8 (2 1) -1) (0 0 1 0) (8 (2 1) -1) (0 0 0 1) (8 (2 1) -1)
(0 0 0 0) (8 (-1 1) -1)))

MAS:
(  W**2 - W +2 X**2 +2 Y**2 +2 Z**2 )

( 2 X W +2 Y X - X +2 Z Y )

( 2 Y W + X**2 +2 Z X - Y )

(  W +2 X +2 Y +2 Z -1 )


MAS: TraceLevel : 2
DecompProc : DIPSFF
UpdateProc : UPDATE
VarOrdOpt  : 3
FacSugar   : 1
ReduceExp  : 1
BranchProc : SSCO


= groebner base =

= changed variable order =
Number of canceled branches/factors : 0

Time : 6984 ms with program GB1 :

1. GB with 5 equation(s)
 ( -697038804/54935 X**6 +71721504/54935 X**5 +80905692/54935 X**4
   -14887832/164805 X**3 -1487501/32961 X**2 +400672/164805 X + Y )
 (  X W -28598724/54935 X**6 +15827184/54935 X**5 +3421752/54935 X**4
```

```
  -4828462/164805 X**3 -82840/32961 X**2 +24707/164805 X )
(  X**7 -26/77 X**6 -53/693 X**5 +184/6237 X**4 +2/2079 X**3
  -1/1386 X**2 +1/24948 X )
(  W +1394077608/54935 X**6 -143443008/54935 X**5
  -161811384/54935 X**4 +29775664/164805 X**3 +2975002/32961 X**2
  -471734/164805 X +2 Z -1 )
(  W**2 -4/3 W +550774224/54935 X**6 -136900224/54935 X**5
  -55289632/54935 X**4 +101348336/494415 X**3 +3030680/98883 X**2
  -2285476/494415 X +1/3 )
```

8.12 Polynomial Invariants Package

This package contains functions and procedures for the reduction of G-invariant polynomials for any given group of permutations and substitutions, respectively. The reduction method finds a representation of a given polynomial in terms of a basis of the ring of G-invariant polynomials. The algorithms are based on the the classical theorem of E. Noether [Noether 1916] and on recent results of Göbel [Göbel 1992, Göbel 1995].

The algorithmic approach is a generalization of the classical algorithm for symmetric polynomials presented, for example, in [Becker, Weispfenning 1993], section 10.7, or [Sturmfels 1993], section 1.1. In contrary to the method of Noether, our rewriting technique works independent of the given ground ring.

The algorithms have been implemented by M. Göbel (diploma thesis [Göbel 1992] and DFG-project: Algorithmische Ideal- und Eliminationstheorie) at the university of Passau under the guidance of V. Weispfenning and H. Kredel.

The invariant package consists of three modules for rewriting permutation invariant polynomials ($GSYMFUIN$, $GSYMFURN$, $NOETHER$) and one module for rewriting substitution group invariant polynomials ($SUBST$). The modules contain algorithms for

- information and help for the user,
- input and output of permutation and substitution groups,
- generation of orbit polynomials,
- checking of invariant properties of polynomials,
- unique separation of polynomials in invariant and remainder polynomials,
- computation of representations and reduction of invariant polynomials, and
- checking of representation and reduction results.

The algorithms can be used interactively in the interpreter environment of MAS.

The following sections give a short description of the implemented MAS procedures and show some examples.

8.12.1 Permutation Invariant Polynomials

The modules in this section use for the representation of G-invariant polynomials the following definition of an orbit polynomial:

$$orbit_G(t) = \sum_{s \in \{\pi(t) \mid \pi \in G\}} s$$

A permutation group G is stored as the list of generating permutations. The order of G is computed over the number of terms of $orbit(X_1^{n-1} X_2^{n-2} \ldots X_{n-1})$.

A description of the algorithms in Section 8.12.2 and Section 8.12.3 can be found in [Göbel 1992]. The algorithms in Section 8.12.4 are based on the proof of Noether's theorem [Noether 1916] as described in [Kraft 1984].

8.12.2 The Integer Case (Module GSYMFUIN)

GSYINF() The procedure writes information about this package to the output stream.

GSYPGR(N: GAMMAINT): LIST The procedure reads the generating elements of a permutation group operating on N variables from the input stream.

GSYPGW(G: LIST) The procedure writes the generating elements of the permutation group G to the output stream.

GSYSPG(N: GAMMAINT): LIST The procedure computes the generating elements for the symmetric group operating on N variables.

GSYORD(G: LIST): GAMMAINT The procedure computes the order of the permutation group G.

GSYNSP(G: LIST) The procedure computes the number of special orbits for the permutation group G.

GINORP(G, MO: LIST): LIST The procedure computes the G-invariant orbit of the monomial MO.

GINCUT(G, Pol: LIST; VAR Pol1, Pol2: LIST)
The procedure performs a unique separation of the polynomial Pol w.r.t. the term order in a G-invariant polynomial $Pol1$ and a remainder polynomial $Pol2$.

GINCHK(G, Base, Pol: LIST): LIST The procedure computes the original polynomial from the representation polynomial Pol and the head term list $Base$ of the G-invariant base polynomials. The algorithm could be used to check a representation.

GINRED(G, Pol: LIST; VAR Base, BasePol, RemPol: LIST)
The procedure computes the polynomial $BasePol$ which is the G-invariant polynomial representation w.r.t. the base polynomials and the unique remainder polynomial $RemPol$ from the polynomial Pol. The head term list of the base polynomials are stored in $Base$.

GINBAS(G: LIST): LIST The procedure computes the head term list of the G-invariant base polynomials for the permutation group G.

8.12.3 The Rational Case (Module GSYMFURN)

GRNORP(G, MO: LIST): LIST See GINORP, rational case.

GRNCUT(G, Pol: LIST; VAR Pol1, Pol2: LIST) See GINCUT, rational case.

GRNCHK(G, Base, Pol: LIST): LIST See GINCHK, rational case.

GRNRED(G, Pol: LIST; VAR Base, BasePol, RemPol: LIST)
See GINRED, rational case.

GRNBAS(G: LIST): LIST See GINBAS, rational case.

GRNGGB(G: LIST): LIST The procedure computes the head term list of the G-invariant base polynomials for the permutation group G using Buchberger's algorithm (cf. [Göbel 1993]).

8.12.4 Noether's Theorem (Module NOETHER)

NOEINF() The procedure writes information about this package to the output stream.

NOENSP(G: LIST) The procedure computes the number of base polynomials of Noether for the permutation group G.

NOERED(G, Pol: LIST; VAR Base, BasePol, RemPol: LIST)
The procedure computes the polynomial $BasePol$, which is the G-invariant polynomial representation w.r.t. the base polynomials and the unique remainder polynomial $RemPol$ from the polynomial Pol by Noether's theorem [Noether 1916]. The head term list of the base polynomials are stored in $Base$.

8.12.5 Substitution Invariant Polynomials (Module SUBST)

The modules in this section use for the representation of G-invariant polynomials the following definition of an orbit polynomial:

$$orbit_G(t) = \sum_{\pi \in G} \pi(s)$$

A substitution group G is stored as the list of all substitutions. The order of G is the length of the list of substitutions.

The algorithms are based on the proof of Noether's theorem [Noether 1916] as described in [Kraft 1984].

SUBINF() The procedure writes information about the package to the output stream.

SUBSGR(N: GAMMAINT): LIST The procedure reads the generating elements of a substitution group with N variables from the input stream. Then the list of all substitutions of the specified group will be computed and returned.

SUBSGW(G: LIST) The procedure writes the elements of the substitution group G to the output stream.

SUBORD(G: LIST): GAMMAINT The procedure computes the order of the substitution group G.

SUBORP(G, MO: LIST): LIST The procedure computes the G-invariant orbit of the monomial MO.

SUBSYM(G, Pol: LIST): GAMMAINT The procedure returns 1, if Pol is G-invariant and otherwise 0.

SUBCHK(G, Base, Pol: LIST): LIST See GINCHK, substitution group case.

SUBRED(G, Pol: LIST; VAR Base, BasePol: LIST)
The procedure computes the polynomial $BasePol$, which is the G-invariant polynomial representation w.r.t. the base polynomials from the G-invariant polynomial Pol by Noether's theorem [Noether 1916]. The head term list of the base polynomials are stored in $Base$.

8.12.6 Examples

Basis polynomials

The example computes a complete set of basis polynomials for the given permutation group by using Buchberger's algorithm [Göbel 1993] `GRNGGB`.

```
pg := GSYPGR(6).
(1 2 3 5 6 4)
(2 3 1 4 5 6)
().

base := GRNGGB(pg).
```

`pg` denotes the permutation group PG generated by `(1 2 3 5 6 4)` and `(2 3 1 4 5 6)`. `base` denotes the rational PG-symmetric base polynomials.

```
ANS: ((1 2 3 5 6 4) (2 3 1 4 5 6))

GRNGGB working... (Term 1/1): (0,0,1,0,0,0)
GRNGGB working... (Term 2/2): (0,0,0,0,0,1)
GRNGGB working... (Term 3/3): (0,1,1,0,0,0)
GRNGGB working... (Term 5/4): (0,0,0,0,1,1)
GRNGGB working... (Term 6/5): (1,1,1,0,0,0)
GRNGGB working... (Term 9/6): (0,0,0,1,1,1)
GRNGGB working... (Term 16/7): (1,0,2,0,0,0)
GRNGGB working... (Term 20/8): (0,0,0,1,0,2)

GRNGGB exit (BASE): (((0,0,1,0,0,0),(1,1)),((0,0,0,0,0,1),(1,1)),
((0,1,1,0,0,0),(1,1)),((0,0,0,0,1,1),(1,1)),((1,1,1,0,0,0),(1,1)),
```

```
((0,0,0,1,1,1),(1,1)),((1,0,2,0,0,0),(1,1)),((0,0,0,1,0,2),(1,1)))
Number of special polynomials: 555
Number of base polynomials:    8

ANS: (((0 0 1 0 0 0) (1 1)) ((0 0 0 0 0 1) (1 1)) ((0 1 1 0 0 0)
(1 1)) ((0 0 0 0 1 1) (1 1)) ((1 1 1 0 0 0) (1 1)) ((0 0 0 1 1 1)
(1 1)) ((1 0 2 0 0 0) (1 1)) ((0 0 0 1 0 2) (1 1)))
```

Noether's Theorem

The example computes a representation for the polynomial $5 \cdot orbit_{Z_4}(X_1^2 X_2^2 X_4)$ over the rationals. Both, the theorem of Noether [Noether 1916] and the rewriting technique for permutation invariant polynomials [Göbel 1992] are used.

```
pg := GSYPGR(4).
(2,3,4,1)
().

f := GRNORP(pg,FIRST(GSRREAD())).
(x1,x2,x3,x4) S (x1**2 x2**2 x4).

NOERED(pg,f,a,b,c).
z := GRNCHK(pg,a,b).
PWRITE(LIST(f)).
xl := VLREAD().
(x1,x2,x3,x4).
DIRLWR(a,xl,0).
sl := VLREAD().
(s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11).
DIRPWR(b,sl,5).

GRNRED(pg,f,a,b,c).
z := GRNCHK(pg,a,b).
DIRLWR(a,xl,0).
sl := VLREAD().
(s1,s2,s3,s4,s5,s6,s7).
DIRPWR(b,sl,5).
```

After getting the permutation group `pg` the next example computes in five steps the following objects. *First*, with `GRNORP` the rational PG-symmetric orbit polynomial `f` of the monomial `(x1**2 x2**2 x4)` w.r.t. the permuation group `pg` generated by `(2,3,4,1)`. *Second*, with `NOERED` the PG-symmetric polynomial `b` (which is the PG-symmetric polynomial representation w.r.t. the base polynomials `a`) and the remainder polynomial `c` from the rational polynomial `f` w.r.t. to the permutation group `pg` after the theorem of E. Noether. *Third*, with `GRNCHK` the original polynomial `z` from the polynomial `b` and the PG-symmetric base polynomials `a` with respect to the permutation group `pg`. *Fourth*, with `GRNRED` the PG-symmetric polynomial `b`, which is the PG-symmetric polynomial reconstruction with respect to the base polynomials `a`, and the remainder polynomial `c` are computed from the polynomial `f` with respect to the permutation group `pg`. *Fifth*, with `GRNCHK` the original polynomial `z` from the polynomial `b` and the PG-symmetric base polynomials `a` with respect to the permutation group `pg`.

```
SKP ((0,0,0,5),(-1,24),(0,0,1,3),(5,12),(0,1,0,2),(-5,6),(0,0,2,1),
(-5,8),(1,0,0,1),(5,4),(0,1,1,0),(5,6))
HT = (0,0,0,5)
```

```
HT = (0,0,1,3)
HT = (0,1,0,2)
HT = (0,0,2,1)
HT = (1,0,0,1)
HT = (0,1,1,0)

NOERED exit (BASE): (((0,0,2,2),(1,1)),((1,0,1,2),(1,1)),((1,1,0,2),
(1,1)),((0,0,1,2),(1,1)),((0,1,0,2),(1,1)),((1,0,0,2),(1,1)),((0,0,
0,2),(1,1)),((0,1,1,1),(1,1)),((0,0,1,1),(1,1)),((0,1,0,1),(1,1)),
((0,0,0,1),(1,1)))
NOERED exit (BASEPOL): ((0,0,0,0,0,0,0,0,0,0,5),(-1,24),(0,0,0,0,0,
0,0,0,0,1,3),(1,6),(0,0,0,0,0,0,0,0,1,0,3),(1,4),(0,0,0,0,0,0,1,0,
0,0,3),(1,12),(0,0,0,0,0,0,0,1,0,0,2),(-1,3),(0,0,0,0,0,1,0,0,0,0,2),
(-1,4),(0,0,0,0,1,0,0,0,0,0,2),(-1,12),(0,0,0,1,0,0,0,0,0,0,2),(-1,6),
(0,0,0,0,0,0,0,0,1,1,1),(-1,3),(0,0,0,0,0,0,1,0,0,1,1),(-1,6),
(0,0,0,0,0,0,0,0,2,0,1),(-1,4),(0,0,0,0,0,0,1,0,1,0,1),(-1,12),
(0,0,0,0,0,0,2,0,0,0,1),(-1,24),(0,0,1,0,0,0,0,0,0,0,1),(1,2),(0,1,0,
0,0,0,0,0,0,0,1),(1,2),(1,0,0,0,0,0,0,0,0,0,1),(1,4),(0,0,0,1,0,0,0,
0,0,1,0),(1,3),(0,0,0,0,0,0,0,1,1,0,0),(1,3),(0,0,0,0,0,1,0,0,1,0,0),
(1,6),(0,0,0,0,0,1,1,0,0,0,0),(1,12),(0,0,0,0,1,0,1,0,0,0,0),(1,12))
NOERED exit (REMPOL): 0

ANS: ((1 0 2 2) (1 1) (2 1 0 2) (1 1) (0 2 2 1) (1 1) (2 2 1 0) (1 1))

Polynomial in the variables: (x1,x2,x3,x4)
Term ordering:
Polynomial list:
          (  x1**2 x2**2 x4 + x1**2 x3 x4**2 + x1 x2**2 x3**
          2 + x2 x3**2 x4**2 )

ANS: ((56 1) (56 2) (56 3) (56 4))

MAS:
            x1**2 x2**2
            x1**2 x2 x4
            x1**2 x3 x4
            x1**2 x2
            x1**2 x3
            x1**2 x4
            x1**2
            x1 x2 x3
            x1 x2
            x1 x3
            x1

ANS: ((46 1) (46 2) (46 3) (46 4) (46 5) (46 6) (46 7) (46 8) (46 9)
(46 1 0) (46 1 1))

MAS: ( -0.04167 s1**5 +0.16667 s1**3 s2 +0.25000 s1**3 s3 +0.08333
s1**3 s5 -0.33333 s1**2 s4 -0.25000 s1**2 s6 -0.08333 s1**2 s7
-0.16667 s1**2 s8 -0.33333 s1 s2 s3 -0.16667 s1 s2 s5 -0.25000
s1 s3**2 -0.08333 s1 s3 s5 -0.04167 s1 s5**2 +0.50000 s1 s9 +0.50000
s1 s10 +0.25000 s1 s11 +0.33333 s2 s8 +0.33333 s3 s4 +0.16667 s3 s6
+0.08333 s5 s6 +0.08333 s5 s7 )

GRNRED working... (BASE) (((1,0,2,2),(1,1)))
HT = (1,0,2,2)
C/R = 8/7 XLS = (0,0,(-1,2),0,(1,2),0,0,(1,2))
GRNRED working... (BASE) (((0,2,1,2),(1,1)),((1,1,1,2),(1,1)),
((1,1,0,2),(1,1)),((1,0,0,2),(1,1)),((0,1,1,1),(1,1)),((0,0,1,1),
(1,1)),((0,1,0,1),(1,1)),((0,0,0,1),(1,1)))
```

```
...
GRNRED working... (BASE) (((1,1,0,2),(1,1)),((1,0,0,2),(1,1)),
((1,1,1,1),(1,1)),((0,1,1,1),(1,1)),((0,0,1,1),(1,1)),((0,1,0,1),
(1,1)),((0,0,0,1),(1,1)))
HT = (0,0,0,1)

GRNRED exit (BASE): (((1,1,0,2),(1,1)),((1,0,0,2),(1,1)),((1,1,1,1),
(1,1)),((0,1,1,1),(1,1)),((0,0,1,1),(1,1)),((0,1,0,1),(1,1)),
((0,0,0,1),(1,1)))
GRNRED exit (BASE_POL): ((0,0,1,0,0,0,1),(-1,1),(1,0,0,0,0,0,1),
(1,2),(0,0,0,1,0,1,0),(-1,2),(0,1,0,0,0,1,0),(-1,2),(0,0,0,1,1,0,0),
(1,2))
GRNRED exit (REM_POL): 0

ANS: ((1 0 2 2) (1 1) (2 1 0 2) (1 1) (0 2 2 1) (1 1) (2 2 1 0) (1 1))
MAS:
            x1**2 x3 x4
            x1**2 x4
            x1 x2 x3 x4
            x1 x2 x3
            x1 x2
            x1 x3
            x1

ANS: ((46 1) (46 2) (46 3) (46 4) (46 5) (46 6) (46 7))

MAS: ( - s1 s5 +0.50000 s1 s7 -0.50000 s2 s4 -0.50000 s2 s6 +0.50000 s3 s4 )
```

Substitution Invariant Polynomials

The example computes a representation for the polynomial $orbit_{Z_4}(X_1^2X_2^2X_4)$ over the rationals. The fist reduction algorithm treats Z_4 as permutation group and the second reduction works with Z_4 stored as substitution group.

```
pg := GSYPGR(4).
(2,3,4,1)
().
sg := SUBSGR(4).
((0,1,0,0), (0,0,1,0), (0,0,0,1), (1,0,0,0))
().

f := GRNORP(pg, FIRST(GSRREAD())).
(x1,x2,x3,x4) S (x1**2 x2**2 x4).

NOERED(pg,f,a1,b1,c1).
ff := GRNCHK(pg,a1,b1).

SUBRED(sg,f,a2,b2,c2).
ff := SUBCHK(sg,a2,b2).
```

After getting the permutation group `pg` and a subgroup `sg`, the next example computes in three steps the following objects. *First*, with `GRNORP` the rational PG-symmetric orbit polynomial `f` of the monomial `(x1**2 x2**2 x4)` w.r.t. the permuation group `pg` generated by `(2,3,4,1)`. *Second*, with `NOERED` the PG-symmetric polynomial `b1` (which is the PG-symmetric polynomial representation w.r.t. the base polynomials `a1`) and the remainder polynomial `c2` from the rational polynomial `f` w.r.t. to the permutation group `pg` after

the theorem of E. Noether. *Third*, with GRNCHK the original polynomial ff from the polynomial b1 and the PG-symmetric base polynomials a1 with respect to the permutation group pg. *Fourth*, with SUBRED SG-symmetric polynomial b2 (which is the SG-symmetric polynomial representation w.r.t. the base polynomials a2) and the remainder polynomial c2 from the rational polynomial f w.r.t. to the substitution group sg after the theorem of E. Noether. *Fifth*, with SUBCHK the original polynomial ff from the polynomial b2 and the PG-symmetric base polynomials a2 with respect to the substitution group sg.

```
ANS: ((2 3 4 1))

MAS:
1 2 3
ANS: ((((1 1) 0 0 0) (0 (1 1) 0 0) (0 0 (1 1) 0) (0 0 0 (1 1)))
((0 0 0 (1 1)) ((1 1) 0 0 0) (0 (1 1) 0 0) (0 0 (1 1) 0))
((0 0 (1 1) 0) (0 0 0 (1 1)) ((1 1) 0 0 0) (0 (1 1) 0 0))
((0 (1 1) 0 0) (0 0 (1 1) 0) (0 0 0 (1 1)) ((1 1) 0 0 0)))

Enter polynomial list:
ANS: ((1 0 2 2) (1 1) (2 1 0 2) (1 1) (0 2 2 1) (1 1) (2 2 1 0) (1 1))

SKP ((0,0,0,5),(-1,24),(0,0,1,3),(5,12),(0,1,0,2),(-5,6),(0,0,2,1),
(-5,8),(1,0,0,1),(5,4),(0,1,1,0),(5,6))
HT = (0,0,0,5)
HT = (0,0,1,3)
HT = (0,1,0,2)
HT = (0,0,2,1)
HT = (1,0,0,1)
HT = (0,1,1,0)

NOERED exit (BASE): (((0,0,2,2),(1,1)),((1,0,1,2),(1,1)),((1,1,0,2),
(1,1)),((0,0,1,2),(1,1)),((0,1,0,2),(1,1)),((1,0,0,2),(1,1)),
((0,0,0,2),(1,1)),((0,1,1,1),(1,1)),((0,0,1,1),(1,1)),((0,1,0,1),
(1,1)),((0,0,0,1),(1,1)))
NOERED exit (BASEPOL): ((0,0,0,0,0,0,0,0,0,0,5),(-1,24),(0,0,0,0,
0,0,0,0,0,1,3),(1,6),(0,0,0,0,0,0,0,0,1,0,3),(1,4),(0,0,0,0,0,0,
1,0,0,0,3),(1,12),(0,0,0,0,0,0,0,1,0,0,2),(-1,3),(0,0,0,0,0,1,0,
0,0,0,2),(-1,4),(0,0,0,0,1,0,0,0,0,0,2),(-1,12),(0,0,0,1,0,0,0,
0,0,0,2),(-1,6),(0,0,0,0,0,0,0,0,1,1,1),(-1,3),(0,0,0,0,0,0,1,0,
0,1,1),(-1,6),(0,0,0,0,0,0,0,0,2,0,1),(-1,4),(0,0,0,0,0,0,1,0,
1,0,1),(-1,12),(0,0,0,0,0,0,2,0,0,0,1),(-1,24),(0,0,1,0,0,0,0,
0,0,0,1),(1,2),(0,1,0,0,0,0,0,0,0,0,1),(1,2),(1,0,0,0,0,0,0,0,
0,0,1),(1,4),(0,0,0,1,0,0,0,0,0,1,0),(1,3),(0,0,0,0,0,0,0,1,1,
0,0),(1,3),(0,0,0,0,0,1,0,0,1,0,0),(1,6),(0,0,0,0,0,1,1,0,0,0,
0),(1,12),(0,0,0,0,1,0,1,0,0,0,0),(1,12))
NOERED exit (REMPOL): 0

ANS: ((1 0 2 2) (1 1) (2 1 0 2) (1 1) (0 2 2 1) (1 1) (2 2 1 0) (1 1))

SKP ((0,0,0,5),(-1,24),(0,0,1,3),(5,12),(0,1,0,2),(-5,6),(0,0,2,1),
(-5,8),(1,0,0,1),(5,4),(0,1,1,0),(5,6))
HT = (0,0,0,5)
HT = (0,0,1,3)
HT = (0,1,0,2)
HT = (0,0,2,1)
HT = (1,0,0,1)
HT = (0,1,1,0)
...
SKP ((0,0,0,5),(-1,24),(0,0,1,3),(5,12),(0,1,0,2),(-5,6),
(0,0,2,1),(-5,8),(1,0,0,1),(5,4),(0,1,1,0),(5,6))
```

```
HT = (0,0,0,5)
HT = (0,0,1,3)
HT = (0,1,0,2)
HT = (0,0,2,1)
HT = (1,0,0,1)
HT = (0,1,1,0)
SUBRED exit (BASE): (((0,0,2,2),(1,1)),((1,0,1,2),(1,1)),((1,1,0,2),
(1,1)),((0,0,1,2),(1,1)),((0,1,0,2),(1,1)),((1,0,0,2),(1,1)),((0,
0,0,2),(1,1)),((0,1,1,1),(1,1)),((0,0,1,1),(1,1)),((0,1,0,1),(1,1)),
((0,0,0,1),(1,1)))
SUBRED exit (BASEPOL): ((0,0,0,0,0,0,0,0,0,5),(-128,3),(0,0,0,0,
0,0,0,0,0,1,3),(64,3),(0,0,0,0,0,0,0,0,1,0,3),(64,1),(0,0,0,0,0,0,
1,0,0,0,3),(64,3),(0,0,0,0,0,0,0,1,0,0,2),(-64,3),(0,0,0,0,0,1,0,0,
0,0,2),(-16,1),(0,0,0,0,1,0,0,0,0,0,2),(-16,3),(0,0,0,1,0,0,0,0,0,
0,2),(-32,3),(0,0,0,0,0,0,0,0,0,1,1,1),(-32,3),(0,0,0,0,0,0,1,0,0,1,
1),(-16,3),(0,0,0,0,0,0,0,0,0,2,0,1),(-16,1),(0,0,0,0,0,0,0,1,0,1,0,1),
(-16,3),(0,0,0,0,0,0,2,0,0,0,1),(-8,3),(0,0,1,0,0,0,0,0,0,0,1),(8,1),
(0,1,0,0,0,0,0,0,0,0,1),(8,1),(1,0,0,0,0,0,0,0,0,0,1),(4,1),(0,0,0,
1,0,0,0,0,0,1,0),(8,3),(0,0,0,0,0,0,0,1,1,0,0),(16,3),(0,0,0,0,0,1,
0,0,1,0,0),(8,3),(0,0,0,0,0,1,1,0,0,0,0),(4,3),(0,0,0,0,1,0,1,0,0,0,
0),(4,3))
SUBRED exit (REMPOL): 0

ANS: ((1 0 2 2) (1 1) (2 1 0 2) (1 1) (0 2 2 1) (1 1) (2 2 1 0) (1 1))
```

8.13 Real root counting for multivariate polynomials

This package contains an implementation for counting real roots of zero dimensional multivariate polynomial ideals with side conditions due to [Pedersen, Roy, Szpirglas 1993] and [Becker, Wörmann 1991]. The implementation was done by [Lippold 1993]. It uses Gröebner Bases for computing in finitely generated residue class algebras. The method of [Ben-Or, Kozen, Reif 1986] is adapted for computing the number of real roots in the case of several side conditions.

We cite from the introduction by the author [Lippold 1993] of the package:

"Das Zählen reeller Nullstellen von Polynomen gehört zu den bekanntesten Problemen der Algebra und bildet die Grundlage für eine Vielzahl von Anwendungen. Auf den Arbeiten von Budan–Fourier und Sturm (...) aus dem 19. Jahrhundert basieren z.B. Algorithmen zur Isolation reeller Nullstellen (siehe [Buchberger *et al.* 1982]) und zur Quantorenelimination in reell abgeschlossenen Körpern (vgl. [Davenport, Siret, Tournier 1988]). Aufgabenstellungen in mehreren Variablen mußten dabei bislang immer auf den univariaten Fall zurückgeführt werden.

Ein von [Pedersen, Roy, Szpirglas 1993] und [Becker, Wörmann 1991] vor kurzem unabhängig voneinander entwickeltes Verfahren gestattet es nun, die gemeinsamen reellen Nullstellen multivariater Polynommengen im nulldimensionalen Fall zu zählen. Die Autoren verallgemeinern dabei Ausführungen von Hermite und Sylvester über den Zusammenhang zwischen der Anzahl der reellen Nullstellen von Polynomen und der Signatur von speziellen Matrizen (...) auf den multivariaten Fall. Bei der Berechnung dieser Matrizen spielt die Theorie der Gröbner Basen eine zentrale Rolle. Eine Stärke dieses neuen Ansatzes liegt in der Möglichkeit zusätzlich noch eine Vorzeichenbedingung an ein Polynom berücksichtigen zu können. Mit dem von [Ben-Or, Kozen, Reif 1986] entwickelten kombinatorischen Argument lassen sich schließlich reelle Nullstellen unter Berücksichtigung

endlich vieler Nebenbedingungen zählen. Übertragen auf die Anwendungsgebiete der Entscheidungsverfahren bzw. der Quantorenelimination in reell abgeschlossenen Körpern ergeben sich damit interessante neue Perspektiven (siehe [Weispfenning 1993]).

Konkret gehen wir von folgender Situation aus: Gegeben ist eine Menge $\emptyset \neq F = \{f_1, \ldots, f_m\}$ von Polynomen in den Variablen $X_1, \ldots, X_n$ mit Koeffizienten aus $\mathbf{Q}$. Die f_i besitzen nur endlich viele gemeinsame Nullstellen in $\mathbf{C}$, was äquivalent ist zu der Aussage, daß das von F erzeugte Ideal nulldimensional ist. Eventuell sind mit den Polynomen $h_1, \ldots, h_s$ noch eine Reihe von Nebenbedingungen zu berücksichtigen. Die Frage ist nun wieviele $(x_1, \ldots, x_n) \in \mathbf{R^n}$ existieren mit $f_i(x_1, \ldots, x_n) = 0$ für $1 \leq i \leq m$ und $h_j(x_1, \ldots, x_n) \varrho_j 0$ für $1 \leq j \leq s$ bei vorgegebenen $\varrho_j \in \{<, =, >\}$."

8.13.1 Examples

```
V := LIST("x","y").  (* Variable list *)

t := DIPTODEF(2).   (* Termorder: 2: invers lexicographic
                       4: total degree invers lexicographic *)

(* Generating polynomials *)
F := DIILRD(V).
(
(3 x**2 - 4 x y),
(5 x - 5 y**2 - 2 y - 1)
)

(* Side conditions *)
H := DIILRD(V).
(
(x**2 + y**2 + 1)
)

G:= DIIPGB(F,0).     (* Groebner basis *)

BEGIN
CLOUT("Generating polynomials:"); BLINES(0);
DIILWR(F,V); BLINES(1);
CLOUT("Side conditions:"); BLINES(0);
DIILWR(H,V); BLINES(1);
CLOUT("Groebner basis:"); BLINES(0);
DIILWR(G,V); BLINES(1);
IF RRZDIM(G) = 1 THEN ZNL := RRICOUNT(G,H,NIL,1)
                 ELSE CLOUT("Not zero-dimensional.") END;

Generating polynomials:
          ( -4 x y +3 x**2 )
          ( -5 y**2 -2 y +5 x -1 )

Side conditions:
          (  y**2 + x**2 +1 )

Groebner basis:
          ( 45 x**3 -56 x**2 +16 x )
          ( 4 x y -3 x**2 )
          ( 5 y**2 +2 y -5 x +1 )

Condition No. 1
Sign-Condition: (1)  Real Zeroes: 2
```

Computing time for Gröbner Basis 340 ms and for real root counting 420 ms.

8.14 Real quantifier elimination

In the modules RQEPRRC and TFORM is an algorithm for real quantifier elimination implemented. The theoretical background is presented in Weispfenning *A New Approach to Quantifier Elimination for Real Algebra* [Weispfenning 1993], see also [Dolzmann 1994]. The programs were implemented by [Dolzmann 1994].

In this section the usage of this program is described. We demand that the user is familiar with the MAS system.

The program uses distributive polynomials for representing terms. So the global variables VALIS, EVORD, and DOMAIN must be set before formulas can be manipulated. For more informations cf. Subsection 8.14.1.

The following procedures are available in the interpreter.

psi:=RQEQE(phi) This is the main procedure for the quantifier elimination. The variables phi and psi are formulas. The returned formula psi is equivalent to phi. After the expansion of the extended boolean operators the following must hold: Each variable is either free or is quantified by one quantifier. Name conflicts are not resolved. Note that the formula ((psi <=> (E x (phi)) does not suffices this condition. The elimination procedure uses the cgb package for computations of Groebner systems. The behavior of this computations is controlled by options of the cgb package. The term orders for the polynomials used in the cgb package and used for the representation of terms in the polynomial equation package must be identical. The computation of psi and the printing of verbose output during the computation is controlled by some options. You can set these options with the procedure RQEOPTSET.

OldOpt:=RQEOPTSET(NewOpt) This procedure sets the options for the quantifier elimination. The variables OldOpt and NewOpt are lists with maximal two elements. These lists have the form (Trace Level,Partial Quantifier Elimination). The pseudo value -1 is used in order to leave an option unchanged. The trace level controls the verbose output of the quantifier elimination. Following trace levels are supported:

0 No verbose output.

1 Very short output. (For insiders only)

2 Long output without long intermediate results.

3 Long output with intermediate results.

The second element of the option list controls the quantifier elimination itself. Following values are supported:

0 The complete quantifier elimination is done. The result formula contains no quantifier.

1 Only partial quantifier elimination of the zero dimensional cases is done.

RQEOPTWRITE() This procedure print the current values of the options.

TfUseDb() This procedure activates the type formula data base. Type formulas are stored in files with the name "TF.d.db", where d is the degree of the type formula. If a required type formula is not stored in the data base, the formula is computed and stored.

TfComputeTf() This procedure deactivates the type formula data base. Required type formulas are computed unconditionally.

Examples will be given after the next subsection.

8.14.1 The Syntax of Formulas

The terms of the formulas of the pq-systems are represented as distributive polynomials. So the corresponding polynomial ring must be fixed. The procedure PQPRING defines the polynomial ring for the pq-system. Argument of this procedure is a list with upto three elements.

1. The domain descriptor for the coefficient of the polynomials used as terms for the pq-system. Only the domain INT is admissible for the quantifier elimination.
2. The list of all variables occuring in all terms of the formula.
3. The term order for the polynomials. The term order used in the cgb package and used for the terms of the pq-system should be equal. Important term orders are

   2 inverse lexicographical term descending order

   4 total degree order

   8 total degree Buchberger lexicographical term order descending order

One example for the usage of PQPRING:

```
d:=ADDDREAD(). INT
PQPRING(LIST(d,LIST("x","a","b"),4)).
```

A value -1 on one position of the list means "Do not change the corresponding value."

An alternative way of the declaration of the underlying polynomial ring is its specification at the beginning of the input of one formula.

The procedure PQIREAD reads a formula from the input stream. The syntax in EBNF notation is contained in table 8.16, the syntax of polynomials is given in table 8.17 (see also table 7.4.3).

Some remarks on the syntax of formulas:

```
Input           = [ PRing ] Formula "."
PRing           = `{' PRingParam {`,' PRingParam} `}'
PRingParam      = `VALIS='VarList | `EVORD='TermOrd | `DOMAIN='Ring
VarList         = `(' Ident {`,' Ident} `)'
TermOrd         = Atom
Ring            = SymbolicDomainDescriptor

Formula         = `(' Formula `)' | AtomicFormula | UnaryOp Formula |
                   Formula BinaryOp Formula | QuantifiedFormula | TruthVal |
                   `#'MasVar
QuantifiedFormula = Quantifier BoundVars `(' Formula `)' |
                    Quantifier BoundVars `:' Formula
BoundVars       = Ident {[`,'] Ident }
TruthVal        = TrueSym | FalseSym
UnaryOp         = NotSym
BinaryOp        = AndSym | OrSym | ImplSym | ReplSym | EquivSym | XorSym
Quantifier      = ForallSym | ExistsSym
AtomicFormula   = `[' DipPolynomial Relation DipPolynomial `]'
Relation        = LessSym | LessOrEqualSym | EqualSym | NotEqualSym |
                  GreaterOrEqualSym | GreaterSym

TrueSym         = `t' | `true' | `verum'
FalseSym        = `f' | `false' | `falsum'
NotSym          = `--' | `~' | `not' | `non'
AndSym          = `/\' | `and' | `et'
OrSym           = `\/' | `or' | `vel'
ImplSym         = `=>' | `==>' | `impl'
ReplSym         = `<=' | `<==' | `repl'
EquivSym        = `<=>' | `equiv'
XorSym          = `<#>' | `xor'
ForallSym       = `a' | `all' | `fa' |  `forall'
ExistsSym       = `e' | `ex' | `exists'
LessSym         = `<' | `les'
LessOrEqualSym  = `<=' | `leq' | `lsq'
EqualSym        = `=' | `equ'
NotEqualSym     = `<>' | `#' | `!=' | `neq'
GreaterOrEqualSym = `>=' | `geq' | `grq'
GreaterSym      = `>' | `gre'
```

Table 8.16: Syntax of Formulas

```
DipPolynomial   = 0 | '(' Term {('+'|'-') Term}
Term            = Power { Power }
Power           = Factor [ '**' Atom ]
Factor          = Ident | Atom

Ident = A|...|Z|a...z|{A|...|Z|a...z|0...9}
        (* Names of variables like "hugo")
Atom  = 1|...|9{0...9}
        (* integers like 4711 *)
```

Table 8.17: Syntax of Polynomials

The input of the symbols like AND, OR, etc. is not case sensitive, but the keywords VALIS, EVORD, DOMAIN must be written with upper case letters. The names of variables of the polynomial ring are case sensitive.

The special formula symbol # is a prefix to mark variables of the interpreter. With this feature a recycling of formulas is possible.

SymbolicDomainDescriptos is a symbolic name for a domain of the arbitrary domain system. For quantifier elimination with RQEQE only INT is admissible.

Here an example for the usage of PQIREAD:

```
phi:=PQIREAD().
{VALIS=(a,b,c,d),EVORD=4,DOMAIN=INT}
(  ex c: all b,a : ((
   ([(a) = (d) ] /\ [ (b) = (c)]) \/
   ([(a) = (c) ] /\ [ (b) = (1)]))
   =>
   [ (a**2) = (b)] )
) .
```

In the last example the polynomial ring is described in the formula input. This description of the underlying polynomial ring is superflous if it was already specified.

An other example:

```
phi:=PQIREAD().
( [(a**2 + 3 a b ) > ( c ) ]).
psi :=PQIREAD().
(#phi or TRUE).
PQPPRT(psi).
```

The result is the formula "`([(a**2 + 3 a b - c ) > 0 ] \/ TRUE)`". Notice that all variables must be declared in the variable list of the underlying polynomial ring.

The operators have different priorities against themselves:

1 negation

2 conjunctions

3 disjunctions

4 implication and Replication

5 equivalent and exclusive or

6 quantifiers

The following procedures are available in the interpreter.

PQMKDNF(phi:LIST):LIST Polynomial equation make disjunctive normal form. phi is a formula; PQMKDNF returns a formula in strict disjunctive normal form which is equivalent to phi.

PQMKCNF(phi:LIST):LIST Polynomial equation make disjunctive normal form. phi is a formula; a formula in strict conjunctive normal form which is equivalent to phi is returned.

PQSIMPLIFY(phi:LIST):LIST Polynomial equation simplify. phi is a formula. A simplification of phi is returned.

PQMKPOS(phi: LIST): LIST Polynomial equation make positive. phi is a formula; a equivalent positive formula is returned i.e. the operator does not occur in the formula.

PQPPRT(phi:LIST) Polynomial equation pretty print. phi is a formula; this writes the formula phi formatted in the output stream.

PQTEXW(phi: LIST) Polynomial equation tex write. The formula phi is printed in tex format in the output stream. (Polynomials are written in the normal mas syntax.)

PQIREAD():LIST Polynomial equation infix read. A formula is read from the input stream.

PQELIMXOPS(phi: LIST): LIST Polynomial equation eliminate extended operation symbols. phi is a formula PQELIMXOPS returns a formula phi1 equivalent to phi. This function replaces all subterms of phi with the operators IMP, REP, EQUIV or XOR with terms with the operators VEL, ET and NON.

PQMKPRENEX(phi,pref:LIST): LIST Polynomial equation make prenex. phi is a formula; pref is an element of {FOREX, FORALL}; a formula psi in prenex normal form is returned. phi must be a relative positive formula without additional operation symbols like IMP, REP, etc. All bound variables in phi must have different specifications (i.e. different names or different types). The only transformation which is used to calculate psi is the interchange of a junctor with a quantifier. The formula psi has the minimal number of blocks of quantifiers under all prenex formulas which are built using only the interchange of a junctor with a quantifier. The argument pref is only respected, if there are two equivalent formulas with the same optimal number of blocks of quantifiers. In this case the formula is returned which has a “pref”-quantifier as the outermost operation symbol.

PQMKVD(phi:LIST): LIST Polynomial equation make variable names disjoint.

PQPRING(R: LIST): LIST Polynomial equation polynomial ring. The global variables that describe the polynomial ring are set. The list R is of the following format: The first entry is the domain descriptor of the field, the second entry is the list of the variables, and the third entry is the term order. Using a -1 one can omit entries. Not all entries must specified. The old parameters are returned.

PQPRINGWR() Polynomial equation polynomial ring write. The description of the polynomial ring is written in the output stream.

8.14.2 Examples

We include 2 short examples:

```
PRAGMA(TIME).
ev:=4.
CGBOPT(LIST(0,1,0,2,0,ev,ev)).
d:=ADDDREAD(). INT
V:=LIST("a","b","x").
PQPRING(LIST(d,V,ev)).
RQEOPTSET(LIST(9,0)).
phi:=PQIREAD().
( E x ( [ ( a x + b ) = 0 ] ) ).
ws:=RQEQE(phi).
PQPPRT(ws).
FORCOUNTAF(ws).

ANS: 4

ANS: ()

ANS: (2 0)

ANS: ((10) (12) (56))

ANS: ((8 0 -1) () 2)

ANS: (3 0)

ANS: FOREX(LVAR(FORVAR(3, 2)), EQU(((1 0 1) (2 1) (0 1 0) (2 1)))).

Input in prenex normal form
(EX x:[(  a x + b ) = 0])
[Number of quantifier blocks: 1
Elimination of an FOREX quantifier
bound Variables(x)
Number of arguments of the disjunction:1
Eliminating one conjunction.
([(  a x + b ) = 0])
Computing a reduced Groebner system ...
 finished.
SysInfo: Time: 67 ms.
Number of cases in the Groebner system: 3
Handle one case of the Groebner system.
Condition:
```

```
  a = 0
  b = 0

Dimension of the ideal = 1
Input in prenex normal form
(EX x:([  a  = 0] /\ [  b  = 0]))
[Number of quantifier blocks: 1
Elimination of an FOREX quantifier
bound Variables(x)
Number of arguments of the disjunction:1
Eliminating one conjunction.
([  a  = 0] /\ [  b  = 0])
Result of the elimination:
([  a  = 0] /\ [  b  = 0])
SysInfo: Time: 34 ms.
]
Handle one case of the Groebner system.
Condition:
  a = 0
  b <> 0

Dimension of the ideal = -1
Handle one case of the Groebner system.
Condition:
  a <> 0

Dimension of the ideal = 0
Groebner Basis:
(   a  x +  b  )

Side conditions '>':
Side conditions '<>':
Time for computation of characteristic polynomial:
SysInfo: Time: 0 ms.
Type formula to compute: T1(c), where
c0 =   - 1
c1 =    1
Time for computing type formula:
SysInfo: Time: 0 ms.
Time for real root count:
SysInfo: Time: 50 ms.
Result of the elimination:
([  a  <> 0] \/ ([  a  = 0] /\ [  b  = 0]))
SysInfo: Time: 250 ms.
]
ANS: FOROR(NEQ(((0 0 1) (2 1))), FORAND(EQU(((0 0 1) (2 1))),
     EQU(((0 1 0) (2 1))))).

Time: read = 0, eval = 267, print = 16, gc = 0.

MAS: ([  a  <> 0] \/ ([  a  = 0] /\ [  b  = 0]))
ANS: ()

ANS: 3
```

The second example:

```
PRAGMA(TIME).
ev:=4.
CGBOPT(LIST(0,1,0,2,0,ev,ev)).
```

```
RQEOPTSET(LIST(0,0)).
phi:=PQIREAD().
{DOMAIN=INT,VALIS=(d,c,a,b)}
ex c:all a:all b:
(((((a - d) <> 0] \/ [(b - c) <> 0]) /\
([(a - c) <> 0] \/ [(b - 1) <> 0]))
\/ [(a**2 - b) = 0])).
ws:=RQEQE(phi).
PQPPRT(ws).
FORCOUNTAF(ws).

ANS: 4

ANS: ()

ANS: (3 0)

DOMAIN: INT (* Integer *)
VALIS:  (d,c,a,b)
EVORD:  2

ANS: FOREX(LVAR(FORVAR(3, 2)), FORALL(LVAR(FORVAR(4, 2)),
     FORALL(LVAR(FORVAR(5, 2)), FOROR(FORAND(FOROR(NEQ(((0 1 0 0)
     (2 1) (0 0 0 1) (2 -1))), NEQ(((1 0 0 0) (2 1) (0 0 1 0)
     (2 -1)))), FOROR(NEQ(((0 1 0 0) (2 1) (0 0 1 0) (2 -1))), NEQ
     (((1 0 0 0) (2 1) (0 0 0 0) (2 -1))))), EQU(((1 0 0 0)
     (2 -1) (0 2 0 0) (2 1)))))))).

ANS: FOROR(FORAND(NEQ(((0 0 0 1) (2 1))), NEQ(((0 0 0 1) (2 1)
     (0 0 0 0) (2 1))), NEQ(((0 0 0 1) (2 1) (0 0 0 0) (2 -1))),
     EQU(((0 0 0 2) (2 1) (0 0 0 0) (2 1))), NEQ(((0 0 0 4)
     (2 1) (0 0 0 0) (2 1)))), FORAND(EQU(((0 0 0 1) (2 1)
     (0 0 0 0) (2 -1))), NEQ(((0 0 0 4) (2 1) (0 0 0 0) (2 1)))),
     FORAND(EQU(((0 0 0 1) (2 1) (0 0 0 0) (2 1))),
     NEQ(((0 0 0 4) (2 1) (0 0 0 0) (2 1))))).

Time: read = 17, eval = 1666, print = 50, gc = 0.

MAS: (([  d  <> 0] /\ [(  d +1 ) <> 0] /\ [(  d -1 ) <> 0] /\
     [(  d**2 +1 ) = 0] /\ [(  d**4 +1 ) <> 0]) \/
     ([(  d -1 ) = 0] /\ [(  d**4 +1 ) <> 0]) \/ ([(  d +1 ) = 0]
     /\ [(  d**4 +1 ) <> 0]))

Time: read = 17, eval = 50, print = 0, gc = 0.
ANS: 9
```

This completes the section on parametric real root counting.

8.15 Construction of involutive bases

We cite from the introduction of the author [Große-Gehling 1995] of the package:

"In dieser Arbeit wird ein neuer Algorithmus zur Lösung von Systemen von polynomialen Gleichungen im nulldimensionalen Fall vorgestellt. Diese neue Methode konstruiert eine involutive Basis eines durch eine Polynommenge erzeugten Ideals. Eine involutive Basis ist eine spezielle Form einer, möglicherweise redundanten, Gröbner-Basis. Anstelle der im Buchberger-Algorithmus [Buchberger 1985] zur Berechnung von Gröbner-Basen benötigten S-Polynome arbeitet dieser Algorithmus mit nicht-multiplikativen

Verlängerungen von Polynomen. Weiterhin werden nicht alle möglichen Reduktionen betrachtet, sondern eine Teilmenge hiervon, die sogenannten *Janet-Reduktionen*.

Diese Arbeit basiert im wesentlichen auf zwei Veröffentlichungen von A. Yu. Zharkov und Yu. A. Blinkov ([Zharkov, Blinkov 1993] und [Zharkov, Blinkov 1993a]), die diese Methode entwickelt und vorgestellt haben."

This package includes the following modules:

ADEXTRA some tools, not absolutely necessary for computing involutive bases.

DIPCJ tools, used from the involutive base algorithms.

DIPDCIB contains the procedure for computing decompositional involutive bases.

DIPIB procedures for computing involutive bases for arbitrary domain polynomials.

DIPIIB procedures for computing involutive bases for integral domain polynomials.

DIPRNIB procedures for computing involutive bases for rational number domain polynomials.

MASLOADJ module which makes procedures available for interactive use.

8.15.1 Computing Janet-irreducible-sets

DILISJ(F,G,red) F is the set to be Janet-reduced, G is the result, red is is a flag which shows if a reduction took place. This procedure is for arbitrary domain polynomials.

G:=DIILISJ(F) F is the set to be Janet-reduced, G is the result. This procedure is only for integral domain polynomials.

G:=DIRLISJ(F) F is the set to be Janet-reduced, G is the result. This procedure is for rational number domain polynomials.

8.15.2 Computing a Janet-normalform of f modulo G

ADNORJ(G,f,h,red) h is a Janet-normalform of f modulo G, red is a flag which shows if a reduction took place. This procedure is for arbitrary domain polynomials.

h:=DIIPNFJ(G,f) h is a Janet-normalform of f modulo G. This procedure is for integral domain polynomials only.

h:=DIRPNFJ(G,f) h is a Janet-normalform of f modulo G. This procedure is for rational number domain polynomals only.

8.15.3 Computing involutive Bases

F is always a set of polynomials and G is an involutive base of Ideal(F). The different algorithms for computing involutive bases are only different implementations of the same idea. They differ in internal details and in computing time. The quickest version is always the version without a number in the procedure name.

arbitrary domain

G:=DIPIB(F) Version from Zharkov, Blinkov: Solving zero-dimensional involutive systems. This procedure makes use of Gerdt's criteriums.

G:=DIPIB2(F) Version from Zharkov, Blinkov: Involutive Bases of zero-dimensional ideals.

G:=DIPIB3(F) Version from Zharkov, Blinkov: Involution Approach to solving systems of algebraic equations.

G:=DIPIB4(F) Another version from Zharkov, Blinkov: Solving zero-dimensional involutive systems.

integral domain

G:=DIIPIB(F) Version from Zharkov, Blinkov: Solving zero-dimensional involutive systems.

G:=DIIPIB2(F) Version from Zharkov, Blinkov: involution approach to solving systems of algebraic equations.

G:=DIIPIB3(F) Version from Zharkov, Blinkov: Involution approach to Solving Systems of Algebraic Equations.

rational numbers domain

G:=DIRPIB(F) Version from Zharkov, Blinkov: Involution Approach to solving systems of algebraic equations.

G:=DIRPIB2(F) Version from Zharkov, Blinkov: Involutive Bases of zero-dimensional ideals.

Note: integral domain and rational number domain implementations are only experimental implementations. They are not optimized.

8.15.4 Setting options

SetDIPIBopt(opt) opt is a list with five options: trace-level, procedure to use for Janet-reduction, Select Strategy for polynomials, Cancel Coefficient and use of Gerdt-Criteriums (only for DIPIB). The options can also be set with the following procedures: SetDIPIBTraceLevel, SetDIPIBISJ, SetDIPIBSelect,

SetDIPIBCancel, SetDIPIBcrit. A trace-level can be given as number between 0 (no information) and 3 (max. information). With a higher trace-level you get more information about progress of computation. For procedures DIPIB2 and DIPIB3 there are two possible selections for a Janet-reductions procedure: 1 - DILISJ, 2 - DIPIRLJ. Select-strategy means the strategy for selecting polynomials from a set of polynomials. Except for DIPIB there are two possible strategys: select the polynomial with minimal total degree of the leading term (1) or select the first polynomial from the set (0). DIPIB always select the polynomial with minimal total degree of the leading term. For integral domain polynomials you can choose two different procedures to cancel down coefficients. 1: Cancel down with the gcd of the coefficients or 0: Cancel down with the leading coefficient (if possible). The last option, the Gerdt-criteriums, is only possible for DIPIB. The default (1) is an use of these criteriums. With a 0 you can swith the use off.

SetDCIBopt(opt) opt is a list with four options: trace-level, decomposition, variable ordering optimization, depth of tree. To set an option you can also use: SetDCIBTraceLevel, SetDCIBDecomp, SetDCIBVarOrdOpt, SetDCIBdepth. The trace-level has the same meaning as before. Decomposition means the choice between factorisation (1) and squarefree decomposition (2). Default is factorisation. If you set DCIBVarOrdOpt to 1 then the variable ordering is optimized before factorisation. The old order is restored after factorisation. With the last option there is a possibility to restrict the depth of the computation tree. A negative number (default) means an unrestricted growth of tree. A positive number means a restriction of depth to this number.

8.15.5 Example

The following example is taken from [Böge *et al.* 1985] Bsp 4, page 91.

```
DIPTODEF(4).
dp:=ADDDREAD().
RN 6
V:=LIST("z","y","x","w","v").
F:=DILRD(V,dp).
(
(v**2 - v + 2 w**2 + 2 x**2 + 2 y**2 + 2 z**2),
(2 v w + 2 w x + 2 x y + 2 y z - w),
(2 v x + w**2 + 2 w y + 2 x z - x),
(2 v y + 2 w x + 2 w z - y),
(v + 2 w + 2 x + 2 y + 2 z -1)
)
G:=DIPIB(F).
DILWR(G,V).
```

The involutive base `G` is as follows. The computing time was 15374ms on a PC running NeXTStep.

```
Time: read = 0, eval = 15374, print = 360, gc = 1641.

(  y**2 x +0.363636 y**3 +0.018595 z y**2 -2.016379 z**2 y -1.540684 z**3 -0.04
```

```
5971 x**2 -0.240458 y x -0.161195 z x -0.193745 y**2 +0.079433 z y +0.413740 z*
*2 -0.003973 w -0.026042 x +0.007344 y +0.033274 z )

(  x w + x**2 + y x +0.222222 z x +0.111111 y**2 -0.333333 z y -0.333333 z**2 -
0.055556 w -0.222222 x +0.111111 z )

(  v +2.000000 w +2.000000 x +2.000000 y +2.000000 z -1.000000 )

(  y w +0.500000 x**2 +2.000000 y x +1.222222 z x +1.611111 y**2 +2.666667 z y
+1.166667 z**2 -0.055556 w -0.222222 x -0.500000 y -0.388889 z )

(  w**2 - x**2 -4.000000 y x -3.555556 z x -2.777778 y**2 -6.666667 z y -3.6666
67 z**2 -0.111111 w +0.555556 x + y +1.222222 z )

(  z w + y x +2.222222 z x +1.111111 y**2 +3.666667 z y +2.666667 z**2 -0.05555
6 w -0.222222 x -0.500000 y -0.888889 z )

(  z**2 x +0.500000 z y**2 +2.181818 z**2 y +1.772727 z**3 +0.045455 y x -0.073
232 z x +0.054293 y**2 -0.219697 z y -0.583333 z**2 +0.012626 w +0.016414 x +0.
011364 y -0.002525 z )

(  z x**2 -1.272727 y**3 -4.326446 z y**2 -8.019534 z**2 y -4.479715 z**3 -0.13
7397 x**2 -0.277611 y x -0.482261 z x +0.302822 y**2 +1.224706 z y +1.403237 z*
*2 -0.034300 w -0.073618 x +0.001878 y +0.030000 z )

(  y x**2 -0.181818 y**3 -1.884298 z y**2 -1.037265 z**2 y -0.422840 z**3 -0.00
8264 x**2 +0.108866 y x +0.203072 z x +0.385383 y**2 +0.890208 z y +0.407713 z*
*2 -0.011587 w -0.032749 x -0.100263 y -0.088922 z )

(  x**3 +0.363636 y**3 +3.154959 z y**2 +4.905935 z**2 y +2.924606 z**3 -0.0914
26 x**2 +0.190120 y x +0.312772 z x -0.231968 y**2 -0.953212 z y -0.927169 z**2
 +0.032804 w +0.034495 x +0.051352 y -0.015900 z )

(  z y x +0.500000 y**3 +2.181818 z y**2 +3.029752 z**2 y +1.353719 z**3 +0.022
727 x**2 -0.042562 y x -0.012489 z x -0.273186 y**2 -0.810193 z y -0.537879 z**
2 +0.010147 w +0.041827 x +0.033678 y +0.028880 z )

(  z**3 y +1.076923 z**4 -0.019231 y**3 -0.011924 z y**2 -0.160639 z**2 y -0.41
8033 z**3 +0.005290 x**2 -0.003227 y x +0.030081 z x -0.007102 y**2 +0.024303 z
 y +0.010086 z**2 +0.000245 w -0.000906 x -0.001463 y +0.003200 z )

(  y**4 +1.794872 z**4 -0.107827 y**3 +0.638567 z y**2 +1.201326 z**2 y +0.0772
15 z**3 +0.069958 x**2 +0.077700 y x +0.120242 z x -0.109202 y**2 -0.084208 z y
 -0.226895 z**2 +0.007726 w +0.005048 x +0.000818 y +0.000575 z )

(  z**2 y**2 -0.538462 z**4 +0.065268 y**3 -0.030036 z y**2 -0.182495 z**2 y +0
.178685 z**3 -0.011406 x**2 -0.017091 y x -0.039788 z x -0.017810 y**2 -0.04813
2 z y -0.018073 z**2 +0.000773 w +0.003669 x +0.007610 y +0.006114 z )

(  z y**3 -0.750583 z**4 -0.212775 y**3 -0.627319 z y**2 -0.474680 z**2 y -0.01
4284 z**3 -0.013292 x**2 +0.023914 y x -0.006341 z x +0.092225 y**2 +0.127603 z
 y +0.131262 z**2 -0.004899 w -0.009098 x -0.010085 y -0.014367 z )

(  z**5 -0.430661 z**4 +0.049304 y**3 +0.091976 z y**2 +0.206040 z**2 y +0.0901
98 z**3 +0.001628 x**2 -0.003740 y x -0.001277 z x -0.014330 y**2 -0.053060 z y
 -0.027672 z**2 +0.000690 w +0.003135 x +0.002239 y +0.002807 z )
```

8.16 Other Packages

In this section we list other important packages with only minimal descriptions.

8.16.1 Greatest common divisors and resultants

This package is from the original ALDES/SAC-2 greatest common divisor and resultant package. It contains algorithms as described in [Collins 1973] and the references given there. The programs are contained in the Modula-2 library `SACPGCD`.

8.16.2 Polynomial factorization

This package is from the original ALDES/SAC-2 polynomial factorization package. It contains algorithms as described in [Collins 1973] and the references given there. The programs are contained in the Modula-2 libraries `SACUPFAC`, `SACMUFAC` and `SACPFAC`.

8.16.3 Polynomial real root isolation

This package is from the original ALDES/SAC-2 univariate polynomial real root isolation package. It contains algorithms as described in [Collins, Loos 1982] and the references given there. The programs are contained in the Modula-2 libraries `SACROOT` and in some of the `SACEXT*` libraries.

8.16.4 Symmetric functions

This package contains programs for the reduction of polynomials with respect to the elementary symmetric functions. By this reduction method it is possible to decide if a given polynomial can be expressed in terms of the the elementary symmetric functions and also to determine such a representation. The algorithms have been developed during a seminar of V. Weispfenning in Passau 1990. The programs are contained in the Modula-2 library `SYMMFU`.

8.16.5 Linear algebra

This is a standard linear algebra package, which contains programs for Gaussian LU-decomposition, determination of special solutions of inhomogeneous systems of linear equations, determination of bases for the 'nullspace' (solutions of the homogeneous system of equations). Further functions contained are matrix multiplication, transposition, rang, determinants and input / output. The programs are written for matricies over the rational numbers and over the integers. The programs are contained in the Modula-2 libraries `LINALGI` and `LINALGRN`.

8.16.6 Linear diophantine equations

This package is from the original ALDES/SAC-2 linear diophantine equation package. It contains algorithms as described in [Chou, Collins 1982] and the references given there.

The programs are contained in the Modula-2 libraries `SACLINDIO`.

8.16.7 Non-Noetherian polynomial rings

The solvable polynomial rings introduced earlier are examples of non-commutative rings, which are still Noetherian (every (left, right) ideal is finitely generated). General non-commutative polynomial rings are known to be no more Noetherian. Nevertheless there is a class of polynomial rings for which there exist finite Gröbner bases for finitely generated ideals. The package contains algorithms as described in [Bader 1994] and the references given there. The programs are contained in the Modula-2 libraries `DINNG`.

This is the end of the packages chapter.

Chapter 9

The ALDES Language

This chapter contains the ALDES language description. The ALDES language has been defined by R. Loos in [Loos 1976]. We describe the revised version as distributed in 1988. The parser has been implemented by K. Rieger. Only the the syntax of the language and notes on differences to the FORTRAN implementation of ALDES are given.

The ALDES parser is invoked via the `PRAGMA(ALDES).` switch. Then a collection of ALDES algorithms is read until the ALDES end of file mark '||' is encountered. ALDES algorithms can be executed as normal MAS algorithms from the MAS interpreter.

In the following sections we discuss first the lexical conventions and then the language syntax.

9.1 Lexical Conventions

The 'atomic' constituents of the language are characters and tokens (character sequences with special meaning).

9.1.1 Character Set

The character set of ALDES is the same as the MAS character set. It consists of the

digits `0123456789`

letters `aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ`

others `.,=+-*/$()␣!"#%&':;<>?@[\]^_‘{}|~`

The number of characters is denoted by χ (= 95 here).

9.1.2 Tokens

Lexical tokens of the language are:

```
#  <  >  =  <=  >=
+  -  *  /  ^
~  \/  /\
(  )  [  ]  {  }  ,  .  ;  :
$  :=  ,...,  ||
keyword  number  identifier
string  char  comment
```

Characters not contained in this list of tokens may only appear in strings and comments.
The keywords are:

if, then, else, while, do, for, repeat, until, case, of,
return, print, goto, go to,
safe, global, array, const, pragma, intrinsic.

The meanings of most of the tokens and keywords should be 'as expected' and are discussed later. At this place we will only say some words on numbers, identifiers, strings and comments.

9.1.3 Numbers

Numbers may be only so called β–integers as in the MAS language.

9.1.4 Identifiers

Identifiers are used as names of variables and names of procedures. The character sequence of an identifier must start with a letter and may be followed by digits and letters. Identifiers are case sensitive, i.e. upper case and lower case letters are distinct. The length of identifiers is restricted by the requirement that they must fit on one input line.

Example: NIL, p123, AL9, XSH, AlongName.

Ornamented identifiers are not supported by the ALDES parser. Naming conventions are as discussed in the MAS language section.

9.1.5 Strings

Character sequences enclosed in double or single quotes are called strings. Within the quotes any character from the character set may appear.

Example:

`"this is a string"` denotes the string `this is a string`,
`"'"` denotes the string ',
`'"'` denotes the string ",
`"x'7""'"` denotes the string `x'7"'`

Strings are internally represented as lists of numbers (β–integers). So all list operations are applicable to strings, e.g. concatenating, reversing etc.

9.1.6 Comments

Comments are sequences of characters enclosed in `[` and `]`. Comments may be nested, i.e. the comment character sequence may contain *pairs* of `[`, `]`.

Comments can appear everywhere except in other tokens.

9.1.7 Blanks

Blanks can appear everywhere except in numbers, identifiers, keywords or multiple letter tokens.

Characters in input lines which do not belong to the MAS character set are converted to blanks. ASCII characters like CR (return), LF (line–feed), EOL (end–of–line) are *ignored* during input form data sets.

9.2 Syntax

In this section we discuss the ALDES language syntax. First we give the complete syntax diagram and the list of syntax errors.

9.2.1 Syntax Diagram

The syntax definition is given in extended BNF notation. That means `name` denotes syntactic entities, `{}` denotes (possibly empty) sequences, `()` denotes required entities, `|` denotes case selection and `[]` denotes optional cases. Terminal symbols are enclosed in double quotes and productions are denoted by `=`. The syntax diagram is listed in table 9.2.1.

For ALDES program constructs, which have no or a different meaning in the MAS environment, a syntax warning message is generated. These are `intrinsic`, `pragma`, `const`, `global` and `safe`. `global` declarations in algorithms are completely ignored since the parser checks for lexicalscope of the variables. `safe` declarations are treated as `VAR` declarations in MAS. There is no distinction between 'safe' and 'unsafe' variables. For undeclared variables `VAR` declarations are generated.

The syntax errors and syntax warnings detected by the parser are summarized in tables 9.2.1 and 9.3.

If a syntax error is detected one of the error messages is displayed followed by the actual input line where the last character read is underscored. However this last character is one character and one lexical token to far. That means the syntax error is caused by one token behind.

Error repair is limited to skipping tokens until something meaningful is found.

In case syntax errors are detected, the execution of the program is totally suppressed, that means no executable code is generated. If a syntax warning is given execution proceeds.

The program constructs `goto` and arrays are only simulated, so they will be interpreted slowly. There exists no ALDES main program; any ALDES procedure can be executed as a MAS procedure, can call MAS procedures and can be called from MAS procedures.

```
program      = { declaration | algorithm } "||"
algorithm    = header { declaration } "(" number ")" statementseq
               { "." "(" number ")" statementseq } "||"
header       = ident ( "(" identlist [ ";" identlist ] ")" |
                       ":=" ident "(" identlist ")" )
statementseq = statement { ";" statement }
statement    = ( "print" string |
               ident [ [ "[" termlist "]" ] ":=" expression |
                       "(" termlist [ ";" termlist ] ")" ] |
               "if" expression "then" statement
                             [ "else" statement ] |
               "while" expression "do" statement |
               "repeat" ( statementseq "until" expression |
                "{" statementseq "}" [ "until" expression ] ) |
               "for" ident ":=" expression [ "," expression ]
                     ",..," expression "do" statement |
               "case" expression "of" "{"
                   { termlist "do" statement ";" } "}" |
               "return" [ "(" [ expression ] ")" ] |
               ( "goto" | "go to" ) ( number | "(" number ")" ) |
               "{" statementseq "}" )
expression   = [ prefixop ] part { oper part }
prefixop     = ( "~" | "+" | "-" )
oper         = ( "+" | "-" | "*" | "/" | "^" |  "=" | "#" |
                 "<" | ">" | "<=" | ">=" | "\/" | "/\" )
part         = ( number | string | char | "$" ident |
                 ident [ ( "(" [ termlist ] ")" |
                           "[" termlist "]" ) ] |
               "(" expression  { "," expression } ")" )
termlist     = expression { "," expression }
identlist    = ident { "," ident }
variable     = ident [ "[" termlist "]" ]
declaration  = ( "global"    variable { "," variable } |
                 "safe"      variable { "," variable } |
                 "array"     ident "[" termlist "]"
                             { "," ident "[" termlist "]" } |
                 "const"     variable "=" expression }
                             { "," variable "=" expression } |
                 "pragma"    variable "=" expression
                             { "," variable "=" expression } |
                 "intrinsic" identlist ) "."
string       = ('"' {character} '"' | "'" {character} "'")
char         = "'" character "'"
ident        = letter { letter | digit }
number       = digit { digit }
```

Table 9.1: ALDES Syntax Diagram

1	identifier expected
2	`)` expected
4	factor expected
8	declaration expected
9	`=` expected
10	`,` or `.` expected
11	`[` expected
12	`]` expected
13	string expected
14	number expected
15	`to` expected
16	`then` expected
17	`of` expected
18	`{` expected
19	`;` or `}` expected
20	`do` expected
22	`,...,` expected
23	`,` or `,...,` expected
24	`(` expected
25	`||` expected
26	statement expected
27	`}` expected
29	`;` or `until` expected
30	`and` or `or` un–expected
31	`/` un–expected
32	`|` un–expected
33	`=` or `:=` expected
34	expression expected
35	`,` or identifier expected
36	declaration or algorithm expected
37	`.` expected

Table 9.2: ALDES Syntax Error Messages

2	`intrinsic` declaration unsupported
3	`pragma` declaration unsupported
6	`.` in header unsupported
7	array as function unsupported
8	`global` declaration in algorithm unsupported
9	`const` declaration unsupported

Table 9.3: ALDES Syntax Warning Messages

9.3 Example

We list a sample ALDES input:

```
PRAGMA(ALDES).
                    b:=AFINV(M,a)
[Algebraic number field inverse.  a is a nonzero
element of q(alpha) for some algebraic number alpha.  M is the
rational minimal polynomial for alpha.  b=1/a.]
(1)  a1:=M;  a2:=a;  v1:=0;  r:=RNINT(1);
     v2:=LIST2(0,r);  repeat {  c:=RNINV(PLDCF(a2));
     v2:=RPRNP(1,c,v2);  if PDEG(a2) = 0 then { b:=v2;
     return };  a2:=RPRNP(1,c,a2);  RPQR(1,a1,a2;q,a3);
     v3:=RPDIF(1,v1,RPPROD(1,q,v2));  a1:=a2;  a2:=a3;
     v1:=v2;  v2:=v3  }||

                    c:=AFPROD(M,a,b)
[Algebraic number field element product.  a and b are elements of
q(alpha) for some algebraic number alpha.  M is the minimal
polynomial of alpha.  c=a+b.]
(1)  cP:=RPPROD(1,a,b);  RPQR(1,cP,M;q,c)||

                    s:=AFSIGN(M,I,a)
[Algebraic number field SIGN.  M is the integral minimal polynomial
of a real algebraic number alpha.  I is an acceptable isolating
interval for alpha.  a is an element of q(alpha).  s=SIGN(a).]
     safe sS,sP,n,sH,s.
(1)  [a rational.]  if a = 0 then { s:=0;  return};
     if PDEG(a) = 0 then { s:=RNSIGN(SECOND(a));
     return }.
(2)  [Obtain the greatest squarefree divisor of an integral
     polynomial similiar to a.]  IPSRP(1,a;r,aP);  sS:=RNSIGN(r);
     aS:=IPPGSD(1,aP);  IS:=I;  FIRST2(IS;u,v);  sP:=0.
(3)  [Obtain an isolating interval for alpha containing no roots
     of als.  RETURN SIGN(a(alpha)). ]
     repeat {  n:=IUPVSI(aS,IS);  w:=RIB(u,v);
     if n = 0 then {s:=IUPBES(aP,w);
     s:=sS  *  s;  return };  if sP = 0 then sP:=IUPBES(M,v);
     sH:=IUPBES(M,w);  if sH # sP then u:=w
     else { v:=w;  sP:=sH };  IS:=LIST2(u,v)  }||

                    c:=AFSUM(a,b)
[Algebraic number field element sum.  a and b are elements of
q(alpha) for some algebraic number alpha.  c=a+b.]
(1)  c:=RPSUM(1,a,b);  return||

|| (* *)
```

The `PRAGMA(ALDES)` statement switches to the ALDES parser. Then comes a sequence of ALDES algorithms. The ALDES parser is terminated by the end of file mark `||`. The dummy (MAS) comment might be necessary in some situations to stop the scanner from reading the next token.

The ornamented identifiers are denoted according to the implementation ALDES transliteration scheme. For example `a*` = `aS`, `s^` = `sH` and `s'` = `sP`.

The used library functions (like `RNINT` or `RPRNP`) must be accessible from the MAS interpreter to be executable.

Chapter 10

System Commands

In this chapter we summarize display commands, pragmas, how to access the operating system and command line parameters.

10.1 Information Display

The first important system commands are concerned with displaying information on the system configuration and what is going on during computations. There are three groups of commands: 1) for configuration display, 2) for specification display and 3) for environment display.

Configuration display commands show which external functions or predefined functions are available. Specification display commands show defined units, sorts, variables, signatures and generic items. Environment display commands show storage usage, stream usage, symbol table contents, and top level variable bindings. A summary of this commands follows:

help and

HELP lists all information available on defined and accessible functions. That is all compiled procedures and functions, all interpreter procedures and functions and all signature definitions. The syntax is

```
help(name[,mod])   or   help(start,end[,mod])
```

'`Name`' means the first characters of a range of names, '`start,end`' means a range of names and '`all`' means all names (this implies `Loaded`). '`,mod`' is optional and '`mod`' can be '`ModulName`' to list module names of the procedures, '`Loaded`' to list loaded procedures, '`Comment`' to list procedure comments (this is the default except for `all`).

EXTPROCS lists all accessible external compiled functions and procedures.

SIGS lists all signature definitions of functions.

GENERICS lists all generic function definitions.

VARS lists all defined variables together with the type information.

SORTS lists all defined sorts names.

UNITS lists all defined unit names.

BIOS displays information on stream I/O usage,

GCM displays information on the storage usage,

SYMTB displays the LISP symbol table including properties,

DUMPENV lists all variable bindings, that is all variables in the top level environment together with their current value. The output is in LISP syntax:

```
(SETQ var value)
```

so it should be possible to dump the variables to a disk file, and later read them in again. Declarations are lost.

LISTENV same as DUMPENV except output is in Modula–2 like syntax. Note that probably some constructs can not be read in again.

PRAGMA(SHOW) displays the actual pragma settings.

Note that all display information can be printed to a file using output stream setting.

10.2 Pragmas

Besides displaying information the MAS system provides commands to modify the state of the running MAS program. These settings are organized by the `PRAGMA` command. The pragmas come in two groups, one for the selection of the actual parser and one for debugging purposes.

The general syntax of the pragma command is as follows:

```
PRAGMA(subcommand).
```

Where `subcommand` is one of the commands discussed in the sequel. If an unknown subcommand is entered, then a list of all available pragmas is displayed. A list of the actual pragma settings is available with the `SHOW` subcommand.

The subcommands which modify and select the parser are as follows:

MODULA input / output is in Modula–2 like syntax, that means the MAS parser for the MAS language is used.

LISP input / output is in LISP syntax.

ALDES input is in ALDES–2 syntax, that means the ALDES parser is used to read the next input until the ALDES end of file mark '||' is encountered.

operator	LISP name	generic name
+	ADD	SUM
-	SUB	DIF
-	SUB	NEG
*	MUL	PROD
/	QUOT	Q
%	REM	REMAIN
		REZIP
^	POW	EXP

Table 10.1: Generic Operators and Functions

GENPARSE for the artihmetical operators the generic names or LISP names are generated. The correspondence is as shown in table 10.1.

The following subcommands turn **flags** ON and OFF:

TIME if ON the times for parsing, evaluation and printing are shown (initial OFF),

TRACE if ON the arguments of the evaluator are displayed, may produce messy output (initial OFF),

DEBUG if ON prints the result produced by the actual parser (initial OFF)

FUSSY switches to strong(er) type checking as usual.

SLOPPY switches to weak(er) type checking.

The actual setting of these flags can be displayed with the `SHOW` subcommand.

10.3 Operating System

With `EXIT` it is possible to leave the MAS main program and return to the operating system. On some computers the MAS system further provides access to the operating system during a running MAS session. Most important is the possibility to call an editor from within MAS. The commands are summarized as follows:

EXIT Leaves the MAS system.

DOS(`"prog parms"`**)** calls the program '`prog`' with the parameters '`parms`'. The meaning of the string depends on the operating system.

EDIT(`"data set name"`**)** edits the specified data set. The editor is expected to be 'EDITOR.PRG' on the current directory. The editor on disk is 'microEMACS 3.9'. The data set name string is prefixed by the string "␣`@MAS.RC`␣" which specifies the startup file for EMACS to be 'MAS.RC'.

10.4 Input / Output

The MAS input / output is organized in so called streams. A discussion of streams and their usage is contained in the chapter on the MAS language 3. In this section we give only a summary of the functions.

The stream switching functions are:

IN("stream") the current input stream is switched to the stream 'stream',

OUT("stream") the current output stream is switched to the stream 'stream',

SHUT("stream") the specified stream is closed.

The 'stream' name may be prefixed by a 'device name' to specify non–disk data sets:

CON: is the terminal

WIN: is a window (not yet implemented)

RAM: is an internal memory stream, 'RAM–disk'

GRA: is a graphic window (not yet implemented)

NUL: is a dummy stream to suppress output, always empty on input, never full on output,

Other 'device names' are passed to the operating system and are usually interpreted as disk data sets.

10.5 Command Line Parameters

The executable MAS program accepts several command line parameters to configure the running system.

The MAS command line syntax is as follows:

```
mas [Options] [File]
```

where `[File]` is a file with MAS input (the default extension is `.mas` and `[Options]` are some of the following options.

1. `--help`
   Display a list of available command line parameters and exit.
2. `--version`
   Output version information and exit.
3. `-C` or `--copyright`
   Displays copyright and copying conditions.
4. `-c COMMAND` or `--command=COMMAND`
   Execute MAS command `COMMAND` before reading File..

5. `-e` or `--exit`
   Exit after all input files have been read and processed.

6. `-E` or `--exit-on-error`
   Exit when an error occurs.

7. `-f FILE` or `--file=FILE`
   Process input initialization file `FILE` instead of default. The file name option '`-f`' can be used to overwrite the default file name ' /.masrc' during startup.

8. `-m SIZE` or `--memorysize=SIZE`
   Set size of MAS memory to SIZE kbytes. The memory option '`-m`' gives the number of Kilobyte storage, requested from the operating system.

9. `-o[FILE]` or `--output[=FILE]`
   Write output to file (named `FILE` or some default name).

10. `-R` or `--no-readline`
    Do not use the GNU readline library for interactive input.

The parameters may appear in any order. On multiple occurrences of parameters, the last occurrence is used.

10.6 The LISP Interpreter

This section describes some issues of the LISP interpreter which are not covered elsewhere. It is usually not required for the use of the MAS system. LISP whizzards may find some background information regarding the MAS LISP implementation. In the chapter on the internal structure of MAS there is also a section on implementation dependent issues.

10.6.1 Functions and Variables

Since in LISP everything is a function and there is no special syntax beyond the S–expression syntax we describe the MAS LISP by the functions. The semantics of the functions is as expected from other LISP systems. The list of all available LISP functions is as follows:

List Processing:
CAR, CDR, CONS, REVERSE, JOIN, NIL (LISP like versions)
FIRST, RED, COMP, INV, CONC, ADV (SAC–2 like versions).

For convenience both the LISP names and the ALDES /SAC–2 names are available. The names are in corresponding order. NIL is the empty list. The ALDES constant '()' is not parsed, so NIL must be used.

The next functions correspond to the operators `+, -, *, /, %` in the MAS language.

Arithmetic:
ADD, SUB, MUL, QUOT, REM.

Relations:
EQ, NE, LE, LT, GE, GT, NOT, AND, OR.

These functions are generated from the MAS relations `=, #, <=, <, >=, >`.

The following is a complete list of the intrinsic LISP functions. The functions with Standard LISP names have the same meaning as in SLISP. Others are described elsewhere.

Functions:
DE, DF, DM, DG, LAMBDA, FLAMBDA, MLAMBDA, GLAMBDA,
VAR, SORT, SIG, MAP, RULE,
UNIT, SPEC, IMPL, MODEL, AXIOMS,
SETQ, ASSIGN, QUOTE, COND,
PROGN, LIST, IF, WHILE, REPEAT,
PROGA, GOTO, ARRAY, LABEL,

The VAR, SIG, MAP, RULE, UNIT, SPEC, IMPL, MODEL and AXIOMS functions are used by the specification component. The PROGA, GOTO, LABEL and ARRAY functions are used for the interpretation of ALDES algorithms.

10.6.2 What is Not Contained

Several functions which are normally available from LISP interpreters are not accessible in MAS LISP, but they are normally available (probably with different name) at the Modula–2 programming level:

CATCH, THROW, ASSOC, PUT, GET, EXPLODE, IMPLODE, GENSYM, SUBLIS.

Definitely not available are:

DO–loops or FOR–loops, a LISP compiler.

Of course a FOR–loop construct can be defined as macro, if one accepts an ugly syntax. GOTO is only available in PROGA for ALDES sequential statements.

Probably available in future releases will be:

LOOP–EXIT–END, CASE, graphics, ...

A CASE construct is available in ALDES, it is simulated using the COND function.

Not contained as primitive but definable are

APPLY, MAP, EVAL.

Here MAP means the LISP MAP function family and not the specification component MAP function. The MAP function family can be defined as procedures, as explained in the section 'Talking LISP'.

Chapter 11

Internal Structure of MAS

In this chapter we will provide some background on the internal structure of MAS. First we discuss the components of MAS, then the program layout, the configuration and available libraries.

11.1 System Components

The MAS system components are identified in table 11.1. Active components (programs) are enclosed in square boxes and passive components (data) are enclosed in oval boxes. Arrows indicate flow of data and lines between boxes show that the components are related in some way.

As already mentioned MAS itself is a Modula–2 program. Thus the MAS program can be recompiled and linked together with other symbolic and numerical libraries by a suitable Modula–2 compiler. This is shown as an arrow from the compiler box on the right to the enclosing MAS box on the left.

On the top line the editor box both acts on the Modula–2 source code (on the right) and the MAS input data (on the left). The input is processed by the following internal components:

1. The parser for the MAS language (Parse box): character strings in concrete syntax are transformed into abstract syntax trees. Static syntax check together with variable scope analysis is performed.

2. The specification processor (Specification box) with an attached data base of declarations (Declarations box): declarations are extracted from the parse tree and stored in the declaration base, information is retrieved during interpretation. The declarations reflect the Modula–2 source code and the library structure.

3. The LISP interpreter (LISP box): according to the type or the function name of an S–expression inner most (that is eager) evaluation is performed.

4. The interface to the compiled library procedures (Call box): if external functions are encountered then compiled procedures are restored and called with the appropriate parameters.

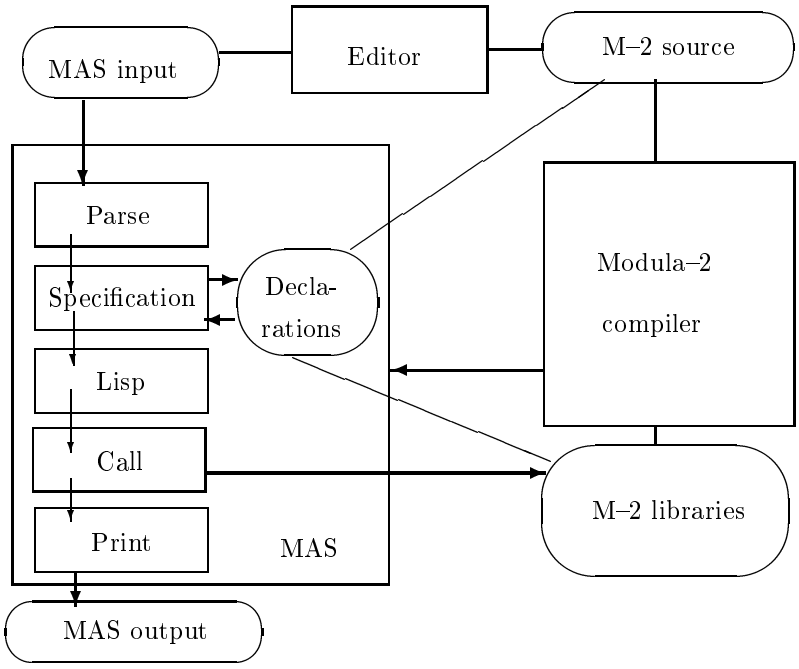


Table 11.1: System Components

5. Finally the results are displayed by the (pretty) printing part (Print box).

11.2 Module Layout of MAS

In this section we will discuss the internal structure of MAS as a Modula-2 program. The principal module structure is shown in table 11.2.

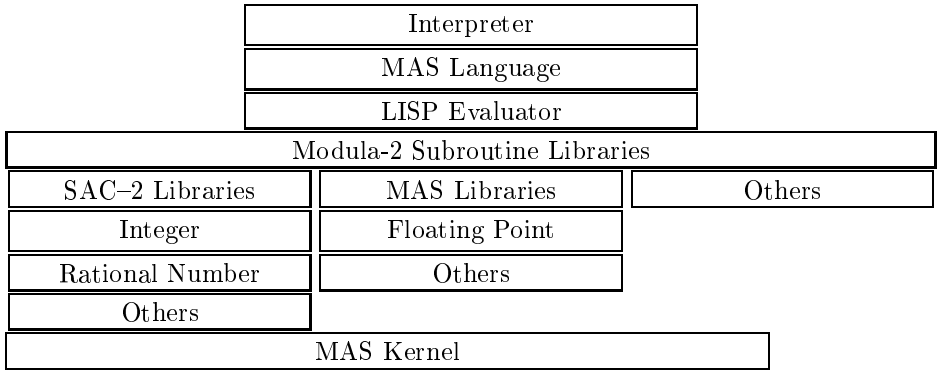


Table 11.2: Module Structure

The MAS interpreter main program uses the parser module 'MAS Language' and the LISP evaluator module. Various library modules can then be connected to the LISP interpreter. All list processing modules use the MAS kernel with its storage management system and basic input/output system.

MAS is an open system. Beside the ALDES /SAC–2 and the MAS libraries other libraries can be connected to the interpreter. In a first test we used a numerical library developed at the Technische Hochschule Aachen by [Engeln, Reutter 1988]. It was not only possible to call numerical programs from MAS, but moreover it was also possible to call MAS routines from the numerical programs.

In a differential equation solver we called the LISP evaluator from within the function which computes the right hand side of the differential equation. This might not be the fastest way to do it but it was possible to define different right hand sides interactively as MAS procedures and then integrate them numerically.

The design of the garbage collector allows arbitrary mixtures of list processing programs with numerical or other programs. It also allows mixtures with other Modula–2 data types such as real numbers, vectors, arrays or records. However it only collects garbage produced by the MAS list processing system, not any other dynamically allocated data structures. The collector uses a mark and sweep technique and is not compacting despite of the open system design.

Restriction: *MAS list pointers may not be part of other* **dynamically allocated** *data structures, since then the garbage collector can not get information about them.*

This restriction can be overcome in the following way: Dynamically allocated data structures must be linked together in some way, so use the MAS list processing for the links and

put pointers to the dynamically allocated data in the FIRST fields of the MAS lists. If no list pointers are stored in other dynamically allocated data structures, then no restrictions exist.

11.2.1 Program Dependencies

A simplified list of the dependencies of the program modules is contained in the following list. Modules with smaller numbers depend on those with higher numbers.

1. MAS.MOD
   main program.
2. SACSYM, MASSPEC, MASLISP, MASPARSE, MASLOAD,
   (.DEF and .MOD),
   symbol system, interpreter and parser.
3. SACD, SACI, SACM, SACRN, SACPRIM, MASAPF,
   MASF (.DEF and .MOD),
   arithmetic system, or various other programs which are connected to the interpreter.
4. SACBIOS, SACLIST (.DEF and .MOD),
   rest of ALDES /SAC-2 basic system.
5. MASELEM, MASSTOR, MASBIOS (.DEF and .MOD),
   basic system adapted for Modula–2 compiler and Atari computer, containing a garbage collector and file handling.

Modules under 3 and 4 belong to the MAS kernel, modules under 1 and 2 constitute the MAS interpreter and parser. Under 3 only an example configuration is given. The MAS configuration set up is discussed in a later section.

11.3 Implementation Issues of the LISP Interpreter

The idea of combining an ALGOL like programming language with LISP is borrowed from A.C. Hearn's RLISP, which is part of the Reduce computer algebra system [Hearn 1987]. Our LISP interpreter is also influenced by SLISP (Standard LISP) [Marti *et al.* 1978]. The actual implementation of the interpreter followed the book 'LISP' of [Stoyan, Goerz 1984]. A similar approach, implementing LISP in BCPL, was persuaded by [Fitch, Norman 1977].

The current implementation uses the MAS storage management and input /output management which have their origin in the ALDES / SAC-2 Basic and List Processing System by [Collins, Loos 1980], [Loos 1976].

The symbol handling was first done by the ALDES / SAC-2 Symbol System, which uses an unbalanced tree for the symbol table. In the present version (≥ 0.6) we use a hash table with balanced symbol tree entries. Therefore the generation of alphabetical lists (as in the HELP command) needs some more time as in the previous system. Without the specification component there are typically less than $100 - 200$ symbols defined, since sub–procedures in compiled code which are not accessible from the interpreter do not count.

With the specifications of algebraic structures loaded about 700 – 1000 symbols are defined. In contrast in a running Reduce system about 4000 symbols and 2000 functions are defined.

The variable binding mechanism is the so called 'deep access' binding using a linear list for the symbols and values (ALIST). At the moment there are typically less than 50 values bound to symbols, since variables in compiled code are not stored in the ALIST. The MAS parser performs static scope anlysis and declares textualy local variables, but at LISP level dynamic scope is implemented.

The minimal MAS executable module is about 90 KB code. This includes storage management, input / output, symbol handling, LISP interpreter, specification component and the parser. Not included are the arithmetic routines. In contrast XLISP is about 76 KB, microEMACS 3.8 is about 80KB, muSIMP-86 is about 57 KB, the muLISP-87 kernel is about 50 KB and SLISP/370 is about 50 KB of code.

A full fledged MAS including arbitrary precision arithmetic, a polynomial system, two Gröbner bases packages, a numerical differential equation solver is about 176 KB. In contrast Derive is about 209 KB, the Reduce savefile is about 610 KB (+ 213 KB for PSL) and the bare Reduce (without factorizer and integrator) is reported to be about 459 KB.

11.3.1 LISP to Modula–2 Interface

The interface between LISP and the compiled code is realized with the Modula–2 procedure types. The procedure code pointers are stored in the property list of a symbol together with some signature information. Upon application of such a function symbol, the procedure is restored, the actual parameters are supplied and the procedure is called.

A simplified example of this mechanism is explained by the following Modula–2 code piece:

```
TYPE procf1 = PROCEDURE(LIST): LIST;
PROCEDURE name(a: LIST): LIST;
          BEGIN  ... RETURN(x) END;
VAR   f: procf1;
      a: ADDRESS;
      l: LIST;
BEGIN a:=ADDRESS(name); ...
      f:=procf1(a);
      l:=f(2); ...
```

With `a:=ADDRESS(name)` the code pointer of the function 'name' is saved in a, and can further be stored in the property list of a symbol. Later an executable function can be restored with `f:=procf1(a)` and can then be executed like any other function: `l:=f(2)`.

In this example the user himself is responsible for using the correct type conversion `procf1`. For execution it is absolutely necessary that the number and types of the actual and formal arguments are identical. For this MAS exports type dependent functions for the declaration of compiled code for interactive usage.

Such a declaration looks like

```
FROM MASLISPU IMPORT Compiledf1;
FROM MASSTOR IMPORT FIRST;
```

```
Compiledf1(FIRST,"CAR");
```

here the compiled function FIRST (with 1 input parameter) is made available as LISP function CAR. And provisions are made to prevent CAR from being called with the wrong number of arguments. The Modula–2 compiler itself checks that the type of the function FIRST matches the type of the first parameter of 'Compiledf1'. So the procedure types are an incredibly helpful feature of Modula–2.

11.3.2 Configuration Management

Despite of the large amount of available procedures it seems necessary to provide some mechanisms to select only the ones which one is actually interested in. The Modula–2 procedure types together with the MAS interpreter design allow for a transparent configuration management.

The association of a compiled procedure with a function symbol of the MAS language is accomplished in a way described in the section on LISP implementation.

In this section we will first discuss some key aspects of the MAS configuration management and then we will give a sample listing of a 'load module'.

In the MAS main program a procedure named `InitExternals` is called, which in turn activates several other `InitExternals`$_{\sqcup}$ procedures. The name externals indicates code that is not required to run the LISP kernel.

With the InitExternals procedures any desired configuration of the MAS system can be achieved. For example if no arithmetic routines are required their declaration can be left out from the InitExternals. Or if only Gröbner bases are to be studied, one can include declarations of the respective procedures.

Four points should be observed:

first, the complete **type checking mechanism** of Modula–2 is in effect. The compiler will detect discrepancies between the declaration of the procedures and their usage. Consider the procedure `IWRITE`. In the import list it is specified from which module IWRITE is to be taken, here `SACI`. Then it is used as input to the `Compiledp1` procedure. If accidentally some other procedure is used (p.e. `Compiledp2`) then the compiler will complain about a type mismatch error.

second, the linker will take care that all subroutines used in the IWRITE procedure are collected together for the executable program. So if only a top level procedure is made accessible to LISP, the linker looks for all **lower level routines** required to execute the top level routine.

third, the configuration of the MAS system can be done using a **familiar program development system**. It is not necessary to learn about a special configuration manager. Simply specify all required routines in the InitExternals and re–link the MAS main program and you have a fitting MAS system.

fourth, only the **minimal number** of procedures is packed together. On other LISP systems or LISP based computer algebra systems the whole packages (or modules) are put together, since it is not distinguished between used (accessible) and unused (not accessible) procedures.

A listing of part of the arithmetic InitExternals is shown:

```
IMPLEMENTATION MODULE MASLOADA;

FROM MASLISPU IMPORT Compiledp1, ...

FROM SACI IMPORT IWRITE, IREAD, INEG, IPROD, ISUM, IDIF,
                 IQ, IREM, IQR, ISIGNF, IABSF, IEXP,
                 ICOMP, IGCD, ILCM, IRAND, ILWRIT;

FROM SACRN IMPORT RNWRIT, RNDWR, RNREAD, RNSIGN, RNCOMP,
                  RNNEG, RNABS, RNINT, RNRED,
                  RNSUM, RNDIF, RNPROD, RNQ;

PROCEDURE InitExternalsA;
(*Tell Modula about external compiled procedures. *)
BEGIN
(*1*) (*from SACI. *)
      Compiledp1(IWRITE,"IWRITE");
      Compiledp1(ILWRIT,"ILWRIT");
      Compiledf0(IREAD,"IREAD");
      Compiledf1(ISIGNF,"ISIGN");
      Compiledf2(ICOMP,"ICOMP");
      Compiledf1(INEG,"INEG");
      ...
      Compiledf2(IGCD,"IGCD");
      Compiledf2(ILCM,"ILCM");
      Compiledf1(IRAND,"IRAND");
(*2*) (*from SACRN. *)
      Compiledp2(RNDWR,"RNDWR");
      Compiledp1(RNWRIT,"RNWRIT");
      Compiledf0(RNREAD,"RNREAD");
      Compiledf2(RNRED,"RNRED");
      ...
      Compiledf1(RNSIGN,"RNSIGN");
      Compiledf2(RNDIF,"RNDIF");
      Compiledf2(RNPROD,"RNPROD");
      Compiledf2(RNQ,"RNQ");
(*3*) (*from MASAPF. *) ...
(*4*) (*from SACPRIM. *) ...
(*5*) (*from MASF. *) ...
(*9*) END InitExternalsA;

END MASLOADA.
```

11.4 Libraries

The libraries are organized in groups, which are placed in separate directories. The currently available libraries are summarized in the sequel. Besides the SAC, DIP and MAS libraries also any other Modula–2 program libraries can be interfaced to the MAS system.

11.4.1 Kernel

The kernel is contained in the directory `MASKERN`.

List Tools
MAS Basic I/O System
MAS BIOS Utility
MAS Configuration
MAS Elementary Functions
MAS Error
MAS Signal Handling
MAS Storage
MAS mtc [Modula-2 to C]
Portability
SAC Basic I/O System
SAC List Processing
System Informations
clock
kpathsearch
readline
MAS Signal Handling
Setjmp

11.4.2 Interpreter, LISP, Main Program

The LISP interpreter, the parser and the main program are contained in the directories `MASLISP` and `MASMAIN`.

Aldes Parser
MAS Lisp
MAS Lisp Utility
MAS Parser
MAS Representation
MAS Specification
MAS Symbol
MAS/SAC Symbol System 2
Modula Global Variable Implementation Module
SAC Symbol System
SAC Symbol 2

MAS Load
MAS Load A
MAS Load B
MAS Load C
MAS Load D
MAS Load E
MAS Load Symmetric Functions
MAS Load J
MAS Load L
MAS Load M
MAS Load Q
MAS Load Syzygy
MAS Utility
MAS Symbol to DIP
MAS Logic Configuration Implementation Module
MAS Logic Demonstration Implementation Module
Masload Polynomial Equation Simplify

11.4.3 Basic arithmetic

The basic artihmetic is contained in the directory `MASARITH`.

MAS Arbitrary Precision Floating Point
MAS Complex Number
MAS Combinatorial System
MAS Floating Point
MAS Integer
MAS Octonion Number
MAS Quaternion Number
MAS Rational Number
MAS Set
SAC Combinatorical System
SAC Digit
SAC Integer
SAC Modular Digit and Integer
SAC Factorization and Prime Number
SAC Rational Number
SAC Set

11.4.4 Polynomial arithmetic

The polynomial arithmetic is contained in the directory `MASPOLY`.

DIP Common Polynomial System
DIP Integral
DIP Integer Polynomial
DIP Rational
DIP Rational Number Polynomial
DIP Termorder Optimization
SAC Dense Polynomial
SAC Integer Polynomial System
SAC Modular Polynomial
SAC Polynomial System
SAC Rational Polynomial

11.4.5 Ring theory, algebraic geometry

The modules for ring theory and algebraic geometry are contained in the directories `SACRING` and `MASRING`.

SAC Algebraic Number Field
SAC Extensions 1
SAC Extensions 2
SAC Extensions 3
SAC Extensions 4
SAC Extensions 5
SAC Extensions 6
SAC Extensions 7
SAC Extensions 8
SAC Modular Univariate Polynomial Factorization
SAC Polynomial Factorization
SAC Polynomial GCD and RES System
SAC Polynomial Real Root
SAC Univariate Polynomial Factorization
DIP Ideal Decomposition 0 System
DIP Dimension
DIP GCD
DIP Ideal System

DIP Integral D-Groebner Bases
DIP Integral Groebner Bases
DIP Rational Function
DIP Rational Groebner Bases
DIP Ideal Real Root System
DIP Zero Dimensional Ideal
MAS Finite Field
MAS Polynomial GCD and RES System
Universal Groebner Bases

11.4.6 Non-commutative Polynomials

The non-commutative polynomial arithmetic is contained in the directory `MASNC`.

DIP Groebner bases for non noetherian polynomial rings
DIP Exterior Algebra
MAS Non-commutative Product
MAS Non-commutative Center
MAS Non-commutative Groebner Bases

11.4.7 Arbitrary domain Polynomials

The arbitrary domain polynomial arithmetic and the comprehensive Gröbner base package are contained in the directory `MASDOM`.

Arbitrary Domain Tools
Comprehensive-Groebner-Bases Applications
Comprehensive-Groebner-Bases Data-Structures
Comprehensive-Groebner-Bases Utility Functions
Comprehensive-Groebner-Bases Main Programms
Comprehensive-Groebner-Bases Miscellaneous Programs
Comprehensive-Groebner-Bases System
DIP Arbitrary Domain
DIP Arbitrary Domain Groebner Basis
DIP Decompositional Groebner Bases
DIP Domain D-Groebner Bases
DIP Groebner Bases
Distributive Polynomials Tools
MAS Domain Algebraic Number

MAS Domain Arbitrary Precision Floating Point
MAS Domain Complex Number
MAS Domain Finite Field
MAS Domain Integer
MAS Domain Integral Polynomial
MAS Domain Modular Digit
MAS Domain Modular Integer
MAS Domain Octonion Number
MAS Domain Quaternion Number
MAS Domain Rational Function
MAS Domain Rational Number
MAS Domain Rational Polynomial
MAS Arbitrary Domain

11.4.8 Module Arithmetic

The linear algebra packages, the syzygy package and the polynomial invariants package are contained in the directory `MASMODUL`.

G-Symmetric Integral Polynomial System
G-Symmetric Rational Polynomial System
MAS Linear Algebra Integer
MAS Linear Algebra Rational Number
Noether Polynomial System
SAC Linear Diophantine Equation System
Substitution Group Polynomial System
Symmetric Functions
Syzygy Functions
Syzygy Groebner Base
Syzygy Utility Programs
Syzygy Main Programs
DIP Rational Extended Groebner Bases

11.4.9 Involutive Bases

The involutive bases package is contained in the directory `MASIB`.

Arbitrary domain extra definition module

DIP Common Polynomial System in the sense of Janet

DIP Decompositional Involutive Bases

DIP Common Polynomial System in the sense of Janet

DIP Integral Polynomial System in the sense of Janet

DIP Rational Numbers Polynomial in the sense of Janet

11.4.10 Real root counting

The Real root counting package is contained in the directory `MASROOT`.

Linear algebra definition module

Real Root Arbitrary Domain

Real Root Integral

Real Root Univariate Arbitrary Domain

Real Root Univariate Integral

11.4.11 Logic formulas and quantifier elimination

The first order logic formulas and quantifier elimination package is contained in the directory `MASLOG`.

Maslog

Maslog Demonstration

Maslog Base

Maslog Input Output System

Polynomial Equation Base

Polynomial Equation Simplification

Real Quantifier Elimination with Parametric Real Root Count

Type Formula

Appendix A

Distribution

A.1 Distribution files

The distribution is by means of gnuzip'ed tar files. Form the distribution files you need at least an executable, the tutorial and the examples.

If you pick the source code we recommend to pick also the definition module and indexes document along with it. To compile the source code you will also need the Modula-2 to C translator "mtc", the "reuse" library, gnumake, and a C-compiler (preferably gcc). Further we recommend the GNU readline library and the kpathsea library.

The following files are available

```
- executables

mas-hppa1.1-hp-hpux9.03-1.00.tar.gz HP Version
mas-i386-unknown-os2-1.00.tar.gz OS/2 Version
mas-i486-unknown-linux-1.00.tar.gz Linux Version
mas-mab-next-nextstep3-1.00.tar.gz NeXT Version
mas-rs6000-ibm-aix3.2.5-1.00.tar.gz IBM Version
mas-sparc-sun-sunos4.1.3C-1.00.tar.gz SUN Version


- documentation:

mastut.tar.gz MAS tutorial and interactive users guide in LaTeX.
mastut.ps.gz as PostScript

masdef.tar.gz MAS Modula-2 definition modules, 3 indexes
and specifications in LaTeX.
masdef.ps.gz as PostScript


- examples and test files:
```

```
masexam.tar.gz examples, help files, test files.


- Modula-2 source code:

massrc.tar.gz
```

A.2 Installation

A.2.1 Unpack

To install MAS unpack the respective files. E.g. with

```
     gnutar xfvz masexam.tar.gz
or
     gzip -d -c masexam.tar.gz | tar -cvf -
```

in some directory. The files will unpack into the following subdirectories:

```
mas/doc
       /mastut Tutorial and Users Guide.
       /masdef Definitions and indexes document.
       /massys   System document.
   /<machine> Executables in respective machine directory
       mas Unix executable.
       mas.exe          OS2 executable, requires /dll.
       emx.exe mas.out DOS extender and executable.
   /exam Examples *.in
                Copying information
        mas.ini initialization file.
        spec.ini  specification initialization file.
        helpup.in Help initialization file.
        testall.mas Driver file for /test directory.
        /help Comments and module information.
        /spec Example specifications.
        /test Test files.
   /dll emx dyn. link libs for OS2.
   /src Modula-2 source code
       /maskern System dependent files, memory, IO, ...
       /maslisp LISP interpreter, parsers, ...
       /masmain  Main program, interfaces, ...
       /masarith Basic Arithmetic, integer, rational, ...
       /maspoly Polynomial systems, recursive, distributive, ...
       /masring Ideals, Groebner bases, algeb. geometry, ...
       /masmodul  Linear algebra, diophantine equations, syzygies, ...
       /masnc Non-commutative solvable polynomial rings, ...
       /sacring Polynomial factorization, real roots, gcd, res, ...
       /masroot Real root counting, ...
       /maslog Logic formuals, Real Quantifier Elimination, ...
       /masdom Domain coefficients, comprehensive G bases, ...
       /masib Involutive bases, ...
```

File naming conventions:

```
*.md Modula-2 definition modules.
*.mi Modula-2 implementation modules.
*.h C header files.
*.c C code files.
*.o object code files.
*.a library archives.

*.ini  MAS initialization files.
*.hlp MAS help information.
*.in MAS input files.
*.mas MAS input files.
*.out MAS output files.
```

A.2.2 Test

Test the installation with the following command

```
[path]mas -m 4000 -f test-all.mas -e -omytest
```

from the /exam directory. This produces a file 'mytest' which you can compare to the supplied 'test-all.orig' file. The warnings are intentional and only lines with timings should "diff"er.

A.3 Start – Stop

Add the mas/bin directory to the PATH or use the complete pathname in the following examples.

```
- start         'mas' or 'mas.exe' or 'emx mas.out'
- banner        'This is MAS the Modula-2 Algebra System, Version 1.xx.'
- system prompt 'MAS: '
- system answer 'ANS: '
- input (e.g.)  'a:=2*3.' A statement is terminated by period '.'
- help with     'help.' or 'help(name).' or 'help(name,Loaded).'
- interupt      ^C     CNTRL-C
- leave with    'EXIT.' or 'exit.' or 'quit.'
```

A.3.1 Path and Compile

On all systems add the mas/¡machine¿ directory to the PATH or use the complete pathname to call the MAS executables.

On OS2 systems also add the mas/dll directory to the LIBPATH and reboot or use your existing emx dlls.

To compile MAS unpack the source code and create a directory "¡machine¿" for your machine type. From the directory mas/¡machine¿ execute ../configure to generate the Makefile, then execute gnumake to compile a mas executable.

Further details can be found in the readme file accompanied with the source code.

A.3.2 Notes

```
- Some help facilities need an 'awk' program.
- The Makefiles for the source code may need an
  'awk' or 'sed' program or other unix utilities.
```

A.4 Release and Change Notes

Major mathematical library changes of the current version 1.0 (June 1996) are:

```
- added a package for counting real roots based on Hermites method
  by F. Lippold,
- added a package on permutation invariant polynomials by M. Goebel,
- added an optimized Groebner base package
  (including the "sugar"-method) by C. Rose,
- added a package to compute factorized Groebner bases by J. Pfeil,
- added a logic formula representation with simplification package
  and real quantifier elimination package by A. Dolzmann,
- a package for involutive bases by R. Grosse-Gehling,
- Improved comprehensive Groebner Base algorithms using factorization
  condition evaluation and case elimination by M.~Pesch,
- arbitrary domain polynomial system extended to a
  generic Groebner base package.
```

Major system changes of the current version 1.0 (June 1996) are:

```
- MAS language accepts small letter key words
  and braces {} to denote list expressions.
- Improved error handling and user signal processing to
  examine long running computations.
- GNU readline for easier command line editing.
- Improved batch processing capabilities.
- Distribution now uses GNU autoconfig for easy compilation.
- Using Kpathsearch Library from K.~Berry.
- Generic garbage collection support for most architectures.
```

The major system changes between release 0.6 and 0.7 (April 1993) are:

```
- Distribution based on Modula-2 to C translator and
  a C distribution which will work on 'most' workstations.
- New support for PC 386 and higher (OS2 2.0 and higher) with
  emx dll runtime libraries.
```

```
- New support for PC 386 and higher (DOS 5.0) with
  emx DOS extender.
- Dropped support for the Atari, Amiga and PC XT up to 286.
  That means, that we do no more distribute executables for
  these systems, but if you have the maskern(el) you can get
  the new source code (except maskern) and compile it on
  your system.
- The HELP and help command has been changed to provide
  name ranges and more information from the procedure comments.
```

Major mathematical library changes between release 0.6 and 0.7 are:

```
- added comprehensive Groebner base package by E. Schoenfeld,
- arbitrary domain polynomial system implemented,
- added several new basic arithmetic packages:
  complex numbers, quaternion numbers, octonion numbers,
  finite fields,
- added package for computation in non-commutative polynomial
  rings of solvable type: *-product, left Groebner base,
  two-sided Groebner base, elements in the center,
- added a package for the computation of generators for the
  module of syzygies of systems of homogeneous polynomial
  equations and Groebner bases for modules over polynomial rings
  (also available for solvable polynomial rings) by J. Phillip,
- added universal Groebner base package by T. Belkahia,
- added d-Groebner base and e-Groebner base packages
  for Groebner bases over the integers and univariate rational
  polynomial rings by W. Mark.
```

The major changes between release 0.3 and 0.6 (March 1991) are:

```
- added language extensions for specification capabilities,
- added a parser for the ALDES language and
  possibility for interpretation of ALDES programs,
- added a linear algebra package,
- added an interface between the MAS language
  and the distributive polynomial system,
- improved symbol handling by hash tables combined with
  balanced trees,
- EMS support for IBM PC implementations.
```

The minor changes between release 0.3 and 0.6 are:

```
- PRAGMA construct for the state definition of the
  MAS executable.
- Overloading of MAS arithmetical operators by
  generic function names.
- Typed string constants in MAS expressions.
```

```
- VAR parameters in MAS procedure declarations in ALDES style.
- Static scope analysis by the parser.
- Explicit stack overflow check since not all compilers
  handled stack overflow correctly.
```

Release notes for Version 0.3 (November 1989):

```
- The MAS parser has been changed for better Modula-2
  compatibility.
- MAS LISP has been made more robust against incorrect
  user input.
- The MAS main program has been enhanced to recognize
  the following command line parameters:
          -m number-of-KB
          -f data-set-name
   - the memory option '-m' gives the number of Kilo-Byte
     storage, requested from the operating system.
   - the file name option '-f' can be used to overwrite the
     default file name 'MAS.INI' during startup. With this
     option MAS can be run in batch mode if the EXIT
     statement is contained in the data set.
```

A.5 Copyrights

```
MAS:         (c) 1989-1996, by H. Kredel, M. Pesch.
ALDES/SAC-2: (c) 1982, by G.E.Collins, R.Loos.
```

All Rights Reserved. Permission is granted for unrestricted noncommercial use and noncommercial redistribution if and only if the copyright notice is retained when a copy is made. There are no known bugs, however I disclaim any usefulness and make no warranty on the correctness of the Modula-2 Algebra System. For certain machines and/or operating systems further copying restrictions apply, e.g. see the files

```
copying.mas, copying.reuse, copying.mtc,
copying.emx, copying.gnu, copying.lib and copying.bsd
```

in the `exam` directory. The C code has been generated from the Modula-2 sources of MAS with the 'mtc' 'Modula-2 to C' translator by GMD Karlsruhe. Although it is not required, you should get a copy of it from some ftp site to have the sources of the used libraries. The executables for PC have been compiled using the GNU gcc compiler with the emx runtime system by Ernst Mattes. The latest versions and documentation of emx can also be found on ftp servers.

Bibliography

[Apel, Lassner 1988] J. Apel, W. Lassner, *An extension of Buchberger's algorithm and calculations in enveloping fields of Lie algebras*, J. Symb. Comp., **6**, pp 361–370, 1988.

[Appel *et al.* 1988] A.W. Appel, R. Milner, R.W. Harper, D.B. MacQueen, *Standard ML Reference Manual (preliminary draft)*, University of Edinburgh, LFCS Report, 1988.

[Armbruster, Kredel 1986] D. Armbruster, H. Kredel, *Constructing universal unfoldings using Gröbner bases*, J. Symb. Comp., **2**, pp 383–388, 1986.

[Bader 1994] I. Bader, *Gröbner bases for non noetherian polynomial rings*, Diplomarbeit, Universität Passau, 1994.

[Becker, Weispfenning 1993] T. Becker, V. Weispfenning, in cooperation with H. Kredel, *Gröbner Bases – A Computational Approach to Commutative Algebra*, Springer Verlag, Graduate Texts in Mathematics 141, New York, 1993.

[Becker, Wörmann 1991] E. Becker, T. Wörmann, *On the Trace Formula for Quadratic Forms and some Applications*, Proc. RAGSQUAD, 1991.

[Belkahia 1992] T. Belkahia, *Implementierung eines Algorithmus zur Konstruktion universeller Gröbner Basen in SAC-2/ALDES*, Diplomarbeit, Universität Passau, 1992.

[Ben-Or, Kozen, Reif 1986] M. Ben-Or, D. Kozen, J. Reif, *The Complexity of Elementary Algebra and Geometry*, Journal of Computer and System Sciences 32, pp 251–264, 1986.

[Buchberger 1965] B. Buchberger, *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*, Dissertation, University of Insbruck 1965.

[Buchberger 1970] B. Buchberger, *Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems*, Aequ. Math. **4**, pp 374–383, 1970.

[Buchberger 1979] B. Buchberger, *A criterion for detecting unneccessary reductions in the reduction of Gröbner bases*, Proc. EUROSAM'79, Springer, Lect. Notes Comp. Sci. **72**, pp 3–21, 1979.

[Buchberger *et al.* 1982] B. Buchberger, G.E. Collins, R.G. Loos, *Computer algebra - symbolic and algebraic computation* Computing Supplement, Vienna: Springer, 1982.

[Buchberger 1985] B. Buchberger, *Gröbner bases: An algorithmic method in polynomial ideal theory*, In: (N.K.Bose ed.) Progress, directions and open problems in multidimensional systems theory. pp 184–232. Dordrecht: Reidel Publ. Comp., 1985.

[Böge *et al.* 1985] W. Böge, R. Gebauer, H. Kredel, *Gröbner bases using SAC-2*, Proc. EUROCAL '85, European Conference on Computer Algebra, Linz 1985, Springer Lect. Notes Comp. Sci. **204**, pp 272–274, 1986.

[Böge *et al.* 1986] W. Böge, R. Gebauer, H. Kredel, *Some Examples for Solving Systems of Algebraic Equations by Calculating Gröbner Bases*, J. Symbolic Computation, **1**, pp 83–98, 1986.

[Butcher 1984] C.J. Butcher, *An application of the Runge Kutta space*, BIT Computer Science Numerical Mathematics **24**, pp 425–440, 1984.

[Calmet, Lugiez 1987] J. Calmet, D. Lugiez, *A knowledge–based system for Computer Algebra*, SIGSAM Bulletin 1987, **21** / 1, pp 7–13, February 1987.

[Carra-Ferro 1986] G. Carra-Ferro, *Some Upper Bounds for the Multiplicity of an Autoreduced Subset of* $\mathbf{N}^m$ *and their Application*, AAECC-3, Grenoble 1985, Springer LNCS, Vol. 229, 1986.

[Carra-Ferro 1987] G. Carra-Ferro, *Some Properties of lattice points and their application to differential algebra*, Communications in Algebra, 15(12), pp 2625–2632, 1987.

[Chou, Collins 1982] T.-W.J. Chou, G.E. Collins, *Algorithms for the solution of systems of linear diophantine equations*, SIAM J. Computing, Vol. 11, No. 4, pp 687–708, 1982.

[Collins 1971] G.E. Collins, *SAC-1*, Technical Reports 129, 115, 156, 135, 145. Computer Sci. Dep., U. of Wisconsin, Madison, Wis., 1971.

[Collins 1973] G.E. Collins, *Computer algebra of polynomials and rational functions*, American Math. Monthly, 80, pp 725–755, 1973.

[Collins 1974] G.E. Collins, *Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition*, Preliminary Report, Proc. EUROSAM 1974, Stockholm, August 1974, pp 80–90.

[Collins, Loos 1980] G.E. Collins, R.G. Loos, *ALDES/SAC-2 now available*, SIGSAM Bulletin 1982, and several reports distributed with the ALDES/SAC-2 system.

[Collins, Loos 1982] G.E. Collins, R.G. Loos, *Real Zeros of Polynomials*, in B. Buchberger, G.E. Collins, R.G. Loos, *Computer algebra – symbolic and algebraic computation*, Vienna: Springer, pp 83–94, 1982.

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

[Davenport 1990] J.H. Davenport, B.M. Trager, *Scratchpad's View of Algebra I: Basic Commutative Algebra*, Proc. DISCO '90 Capri, LNCS 429, pp 40–54, Springer, 1990.

[Dolzmann 1994] A. Dolzmann, *Reelle Quantorenelimination durch parametrisches Zählen von Nullstellen*, Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1994.

[Engeln, Reutter 1988] G. Engeln-Müllges, F. Reutter, *Formelsammlung zur Numerischen Mathematik mit Modula-2 Programmen*, BI-Wissenschaftsverlag, Mannheim 1988.

[Fitch, Norman 1977] J.P. Fitch, A.C. Norman, *Implementing LISP in a High-level Language*, SOFTWARE-PRACTICE AND EXPERIENCE Vol.7, pp 713–725, 1977.

[Garey, Johnson 1978] M.R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, H. Freeman, San Francisco, 1978, printing 1984.

[Gebauer, Kredel 1983] R. Gebauer, H. Kredel, *Distributive Polynomial System*, Several Technical Reports, Institut für Angewandte Mathematik, Universität Heidelberg, 1983.

[Gebauer, Kredel 1983a] R. Gebauer, H. Kredel, *Buchberger algorithm system*, Technical Report, Institut für Angewandte Mathematik, Universität Heidelberg, 1983. (see also SIGSAM Bulletin Vol. 18, No. 1, p 19.)

[Gebauer, Möller 1988] R. Gebauer, H.-M. Möller, *On an installation of Buchberger's algorithm*, J. Symb. Comp., **6**/2/3, pp 275-286, 1988.

[Geddes *et al.* 1986] B.W. Char, G.J. Fee, K.O. Geddes, G.H. Gonnet, M.B. Monagan, *A Tutorial Introduction to Maple*, J. Symbolic Computation 2, pp 179–200, 1986.

[Geddes *et al.* 1992] K.O. Geddes, S.R. Czapor, G. Labahn, *Algorithms for Computer Algebra*, Kluwer Academic Publishers, Boston, 1992.

[Gianni 1986] P. Gianni, *Using Gröbner to reduce solving equations to root finding*, Meeting on Computer and Commutative Algebra, Genova, May 1986.

[Gianni 1987] P. Gianni, *Properties of Gröbner bases under specialization*, Proc. EUROCAL'87, European Conference on Computer Algebra, Leipzig, GDR, 1987, Springer LNCS 378, pp 293–297, 1989.

[Gianni *et al.* 1986] P. Gianni, B. Trager, G. Zacharias, *Gröbner Bases and Primary Decomposition of Polynomial Ideals*, J. Symbolic Computation, **6**, No 2/3, pp 149–167, 1988.

[Giovini *et al.* 1991] A. Giovini, T. Mora, G. Niesi, L. Robbiano, C. Traverso, *"One sugar cube, please" or Selection strategies in the Buchberger algorithm*, J. of the ACM, pp 49–54, 1991.

[Giusti 1984] M. Giusti, *Some Effectivity Problems in Polynomial Ideal Theory*, EUROSAM'84, Springer LNCS, Vol. 174, pp 159–171, 1984.

[Goldberg 1981] A. Goldberg, *Introducing the Smalltalk–80 System*, Byte 6, 8 pp 14–35, August 1981.

[Göbel 1992] M. Göbel, *Reduktion G-symmetrischer Polynome für beliebige Permutationsgruppe G*, Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1992.

[Göbel 1993] M. Göbel, *Using Buchberger's Algorithm in Invariant Theory*, ACM SIGSAM Bulletin 27/4, pp 3–9, 1993.

[Göbel 1995] M. Göbel, *Computing Bases for Permutation-Invariant Polynomials*, Journal of Symbolic Computation 19, pp 285-291, 1995.

[Gräbe, Lassner] H.-G. Gräbe, W. Lassner, *A Parallel Gröbner Factorizer*, Universität Leipzig, Preprint, 1994.

[Gröbner 1968/70] W. Gröbner, *Algebraische Geometrie I, II*, Bibliographisches Institut, Mannheim, 1968, 1970.

[Große-Gehling 1995] R. Große-Gehling, *Konstruktion involutiver Basen im Computeralgebra-System MAS*, Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1995.

[Hearn 1987] A.C. Hearn, *REDUCE 3.3*, The Rand Corporation, 1987.

[Hermann 1926] G. Hermann, *Die Frage der endlich vielen Schritte in der Theorie der Polynomideale*, Math. Ann. Bd. 95, pp 736–788, 1926.

[Jenks *et al.* 1984] R.D. Jenks *et al.*, *SCRATCHPAD II, An Experimental Computer Algebra System, Abbreviated Primer and Examples*, Mathematical Sciences Department, IBM, Yorktown Heights, 1984.

[Jenks *et al.* 1985] R.D. Jenks *et al.*, *Scratchpad II Programming Language Manual*, Computer Algebra Group, IBM, Yorktown Heights, NY, 1985.

[Kalkbrenner 1987] M. Kalkbrenner, *Solving systems of algebraic equations by using Gröbner bases*, Proc. EUROCAL'87, European Conference on Computer Algebra, Leipzig, GDR, 1987, Springer LNCS 378, pp 282–292, 1989.

[Kandri-Rody 1984] A. Kandri-Rody, *Effective Methods in the Theory of Polynomialideals*, Ph.D.Thesis, RPI, Troy, NY, May 1984.

[Kandri-Rody 1985] A. Kandri-Rody, *Dimension of ideals in polynomial rings*, Proc. Combinatorial Algorithms in algebraic structures, Otzenhausen 1985, J.Avenhaus, K.Madlener Eds., Fachbereich Informatik, University of Kaiserslautern, 1985.

[Kandri-Rody, Weispfenning 1988] A. Kandri-Rody, V. Weispfenning, *Non-commutative Gröbner bases in algebras of solvable type*, J. Symb. Comp., **9**, pp 1–26, 1990. also available as: Technical Report University of Passau, MIP-8807, March 1988.

[Knuth 1981] D.E. Knuth, *The Art of Computer Programming*, Vol. 2, *Seminumerical Algorithms (second edition)*, Addison-Wesley, 1981.

[Kraft 1984] H.-P. Kraft, *Geometrische Methoden in der Invariantentheorie*, Aspects of Mathematics, Vieweg, Braunschweig/Wiesbaden, 1984.

[Kredel 1985] H. Kredel, *Über die Bestimmung der Dimension von Polynomidealen*, Diplomarbeit, Fakultät für Mathematik Universität Heidelberg, 1985.

[Kredel 1987] H. Kredel, *Primary Ideal Decomposition*, Proc. EUROCAL'87, European Conference on Computer Algebra, Leipzig, GDR, 1987, Lecture Notes Computer Science **378**, pp 270–281, 1989.

[Kredel 1988] H. Kredel, *From SAC-2 to Modula-2*, Proc. ISSAC'88 Rome, Lecture Notes Computer Science **358**, pp 447–455, 1989.

[Kredel 1989] H. Kredel, *Real Roots of Zero-dimensional Ideals*, Manuscript, Universität Passau, 1989.

[Kredel 1990] H. Kredel, *MAS Modula-2 Algebra System*, Proc. DISCO 90 Capri, Springer LNCS **429**, pp 270–271, 1990.

[Kredel 1990a] H. Kredel, *Computing in polynomial rings of solvable type*, Proc. IV. Int. Conv. Computer Algebra in Physical Research 1990, JINR, Dubna, Moscow UdSSR, May 1990, World Scientific, Singapore, pp 211–221, 1991.

[Kredel 1991] H. Kredel, *The MAS Specification Component*, Proc. PLILP '91, Passau, 1991, LNCS **528**, pp 39–50, 1991.

[Kredel 1992] H. Kredel, *Solvable Polynomial Rings*, Dissertation Universität Passau, 1992, Verlag Shaker, Aachen 1993.

[Kredel, Weispfenning 1988] H. Kredel, V. Weispfenning, *Computing Dimension and Independent Sets for Polynomial Ideals*, J. Symbolic Computation (1988), **6**, pp 231–247, 1988. See also MIP-8809, Universität Passau, April 1988.

[Kronecker 1882] L. Kronecker, *Grundzüge einer arithmetischen Theorie der algebraischen Größen*, J. reine angew. Math. Bd. **92**, pp 1–122, 1882.

[Kutzler, Stifter 1986] B. Kutzler, S. Stifter, *Automated geometry theorem proving using Buchberger's algorithm*, CAMP–Publ.–Nr. 85–29.0, University of Linz 1986; Proc. SYMSAC '86, Waterloo, Canada, 1986.

[Lasker 1905] , E. Lasker, *Zur Theorie der Moduln und Ideale*, Math. Ann. Bd. 60, pp 20-116, 1905.

[Lawrence 1987] D.M. Lawrence, *micro EMACS 3.8 Editor*, 1987.

[Lazard 1982] D. Lazard, *Commutative algebra and computer algebra*, Springer Lect. Notes Comp. Sci. **144**, pp 40–48, 1982.

[Lazard 1985] D. Lazard, *Ideal Bases and Primary Decomposition: Case of Two Variables*, J. Symbolic Computation **1**, pp 261–270, 1985.

[Lippold 1993] F. Lippold, *Implementierung eines Verfahrens zum Zählen reeller Nullstellen multivariater Polynome*, Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1993.

[Loos 1976] R.G. Loos, *The Algorithm Description Language ALDES (Report)*, SIGSAM Bulletin **14**/1, pp 15–39, 1976.

[Loos 1982] R.G. Loos, *Computing in Algebraic Extensions*, in B. Buchberger, G.E. Collins, R.G. Loos, *Computer algebra – symbolic and algebraic computation*, Vienna: Springer, pp 173–187, 1982.

[Macaulay 1916] F.S. Macaulay, *Algebraic theory of modular systems*, Cambridge Tract. 1916.

[Mark 1992] W. Mark, *Gröbner Basen über Hauptidealringen und euklidischen Ringen*, Diplomarbeit, Universität Passau, 1992.

[Marti *et al.* 1978] J.B. Marti, A.C. Hearn, M.L. Griss, C. Griss, *Standard Lisp Report*, University of Utah, Salt Lake City, 1978.

[Mateti 1988] P. Mateti, *Guläm Shell*, 1987.

[Melenk *et. al.* 1989] H. Melenk, H.M. M"oller, W. Neun, *Symbolic Solution of Large Stationary Chemical Kinetics Problems*, Impact of Computing in Science and Engineering 1, pp 138-167, 1989.

[Morgan 1983] A.P. Morgan, *A Method for Computing all Solutions to Systems of Polynomial Equations*, ACM/TOMS **9**, pp 1–17, 1985.

[Möller, Mora 1983] H.M. Möller, F. Mora, *The computation of the Hilbert function*, in Proc. EUROCAL '83, Computer algebra, Springer LNCS **162**, 1983.

[Möller, Mora 1986] H.M. Möller, T. Mora, *New constructive methods in classical ideal theory*, J. of Algebra, **100**, pp 138–178, 1986.

[Mora 1985] T. Mora, *Gröbner bases for non-commutative polynomial rings*, Proc. AAECE-3, Grenoble 1985, LNCS 229, pp 353–362, 1985.

[Mora 1986] T. Mora, *Standard bases and non-Noetherianity: Non-commutative polynomial rings*, Proc. AAECE-4, Karlsruhe 1986, LNCS 307, pp 98–109, 1986.

[Mora, Robbiano 1988] T. Mora, L. Robbinano, *The Gröbner fan of an ideal*, J. Symbolic Computation 6/2, pp 183–208, 1988.

[Noether 1916] E. Noether, *Der Endlichkeitssatz der Invarianten endlicher Gruppen*, Mathe. Ann. 77, pp 89–92, 1916.

[Noether 1921] E. Noether, *Ideal Theorie in Ringbereichen*, Math. Ann. Bd. 8, pp 24–66, 1921.

[Patera *et al.* 1976] J. Patera, R.T. Sharp, P. Winternitz, H. Zassenhaus, *Invariants of real low dimnension Lie algebras*, J. Math. Phys., vol. 17, no. 6, pp 986–993, 1976.

[Pavelle, Wang 1985] R. Pavelle, P.R. Wang, *MACSYMA From F to G*, J. Symbolic Computation 1, pp 69–100, 1985.

[Pedersen, Roy, Szpirglas 1993] P. Pedersen, M.-F. Roy, A. Szpirglas, *Counting Real Zeros in the Multivariate Case*, Proc. MEGA'92, F. Eyssette, A. Galligo, Eds., Birkhäuser Boston, pp 203-224, 1993.

[Pfeil 1994] J. Pfeil, *Implementierung von Varianten des Buchberger-Algorithmus mit Polynomfaktorisierung*, Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1994.

[Philipp 1991] J. Philipp, *Syzygien Berechnung im Computer Algebra System MAS*, Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1991.

[Pohst, Yun 1981] M.E. Pohst, D.Y.Y. Yun, *On solving systems of algebraic equations via ideal bases and elimination theory*, Proc. 1981 ACM Symposium on Symbolic and Algebraic Computations, pp 206–211, 1981.

[Rich *et al.* 1988] A. Rich, J. Rich, D.R. Stoutemyer, *DERIVE A Mathematical Assistant*, The Soft Warehouse. Honolulu, Hawaii, 1988.

[Rose 1995] C. Rose, *The MAS module DIPAGB: An Implementation of the Normal with Sugar Selection Strategy in the Buchberger Algorithm*, Fakultät für Mathematik und Informatik, Universität Passau, 1995.

[Robbiano 1985] L. Robbiano, *Term orderings on the polynomial ring*, Proc. EUROCAL '85, European Conference on Computer Algebra, Linz 1985, Springer Lect. Notes Comp. Sci. **204**, pp 513–517, 1986.

[Schemmel 1987] K.-P. Schemmel, *An extension of Buchberger's algorithm to compute all reduced Gröbner bases of a polynomial ideal*, Proc. EUROCAL '87, Leipzig, Springer LNCS Vol. 378, pp 300–310, 1987.

[Schönfeld 1991] E. Schönfeld, *Parametrische Gröbnerbasen im Computer Algebra System ALDES/SAC-2*, Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1991.

[Schrader 1976] R. Schrader, *Zur konstruktiven Idealtheorie*, Diplomarbeit, Mathematisches Institut II, Universität Karlsruhe, 1976.

[Schwartz 1988] N. Schwartz, *Stability of Gröbner bases*, J. pure and appl. Algebra, **53**, pp 171–186, 1988.

[Seidenberg 1974] A. Seidenberg, *Constructions in Algebra*, Trans. Amer. Math. Soc. **197**, pp 273–313, 1974.

[Stoutemyer 1986] D.R. Stoutemyer, *μ-MATH-86*, The Soft Warehouse. Honolulu, Hawaii, 1986.

[Stoyan, Goerz 1984] H. Stoyan, G. Goerz, *LISP*, Springer Verlag, Heidelberg, 1984.

[Sturmfels 1993] B. Sturmfels, *Algorithms in Invariant Theory*, Vienna: Springer, 1993.

[TDI 86] TDI, *Modula-2/ST Compiler*, Clifton, Bristol, UK, 1986.

[Trinks 1978] W. Trinks, *Über Buchbergers Verfahren Systeme algebraischer Gleichungen zu lösen*, J. Number Theory, Vol. 10, pp 475–488, 1978.

[Weinberger *et al.* 1976] J.P. Weinberger, L.P. Rothschild, *Factoring Polynomials Over Algebraic Number Fields*, ACM/TOMS **2**, No. 4, pp 335–350, 1976.

[Weispfenning 1975] V. Weispfenning, *Model-completeness and elimination of quantifiers for subdirect products of structures*, J. of Algebra, 36, pp 252–277, 1975.

[Weispfenning 1987] V. Weispfenning, *Admisible orders and linear forms*, ACM SIGSAM Bulletin, Vol. 21, No. 2, pp 16–18, May 1987.

[Weispfenning 1986] V. Weispfenning, *Some bounds for the construction of Gröbner Bases*, Proc. AAECC–4, Karlsruhe 1986, Springer LNCS Vol. 307, pp 195–201, 1988.

[Weispfenning 1987a] V. Weispfenning, *Constructing universal Gröbner Bases*, Proc. AAECC–5, Springer LNCS Vol. 356, pp 408–417, 1987.

[Weispfenning 1987b] V. Weispfenning, *Gröbner bases in polynomial rings over commutative regular rings*, Proc. EUROCAL '87, Leipzig, Springer LNCS Vol. 378, pp 336–347, 1987.

[Weispfenning 1990] V. Weispfenning, *Comprehensive Gröbner bases*, preprint in: Technical Report University of Passau, MIP-9003, 1990.

[Weispfenning 1993] V. Weispfenning, *A New Approach to Quantifier Elimination for Real Algebra*, Proceedings of the Collins Symposium "QEPCAD"; Linz, Austria Oct. 1993 (to be appear). preprint in: Technical Report University of Passau, MIP-9305, 1993.

[Winkler *et al.* 1985] F. Winkler, B. Buchberger, F. Lichtenberg, H. Rolletschek, *An algorithm for constructing canonical bases (Gröbner bases) of polynomial ideals*, ACM/TOMS **11**, pp 66–78, 1985.

[Wirsing 1986] M. Wirsing, *Structured Algebraic Specifications: A Kernel Language*, Theoretical Computer Science **42**, pp 123–249, Elsevier Science Publishers B.V. (North–Holland) (1986).

[Wirth 1985a] N. Wirth, *Programming in Modula-2*, Springer, Berlin, Heidelberg, New York, 1985.

[Wirth 1985b] N. Wirth, *Compilerbau*, Teubner Verlag, Stuttgart, 1985.

[Wirth 1988] N. Wirth, *From Modula to Oberon*, pp 661–670, *The Programming Language Oberon*, pp 670–690, Software-Practice and Experience Vol. 18(7), July 1988.

[Wirth 1989] N. Wirth, J. Gutknecht, *The Oberon System*, pp 857–893, Software-Practice and Experience Vol. 19(9), September 1989.

[Wolfram 1988] Wolfram Research Inc., *Mathematica*, Addison-Wesley, Reading, 1988.

[Zacharias 1978] G. Zacharias, *Generalized Gröbner bases in commutative polynomial rings*, Bachelor thesis, MIT, Dep. of Comp. Sci., 1978.

[Zharkov, Blinkov 1993] A. Yu. Zharkov, Yu. A. Blinkov, *Involution Approach to Solving Systems of Algebraic Equations*, Proceedings of the IMACS'93, pp 11–16, 1993.

[Zharkov, Blinkov 1993a] A. Yu. Zharkov, Yu. A. Blinkov, *Involutive Bases of zero-dimensional Ideals*, submitted to: Journal Symbolic Computation.

[Zharkov, Blinkov 1993b] A. Yu. Zharkov, Yu. A. Blinkov, *INVBASE: A Package for Computing Involutive Bases*, contained in the Reduce INVBASE-Package, available from Konrad-Zuse-Zentrum Berlin.

[Zharkov 1994] A. Yu. Zharkov, *Solving Zero-Dimensional Involutive Systems*, to appear in Proc. MEGA '94.

Index