Faculty of Applied Mathematics

University of Twente
University for Technical and Social Sciences

P O. Box 217
7500 AE Enschede
The Netherlands
Phone +31-53-4893400
Fax +31-53-4892255
Email c.h g.hassing-assink@math.utwente.nl

MEMORANDUM No. 1331

A manual for the package TOOLS 2.1

G.F. POST

AUGUST 1996

ISSN 0169-2690

A manual for the package TOOLS 2.1.

Gerhard Post

August 21, 1996

Abstract

The package TOOLS for REDUCE contains procedures for handling algebraic operators and some procedures which are handy when doing algebra. In particular we provide a mechanism to handle (multi)linear operators.

AMS subject classification : 68C20.

Keywords: Computer algebra.

1 Introduction

If one is dealing with large systems of algebraic expressions, there are a number of operations that one would like to be able to perform. Let us give some examples.

Suppose one has the expression

```
1: equ := 15*a(5)*beta + 10*a(4) + 3*a(2) + a(1) + 7;
```

Maybe one would like to know what kernels of the type `a(i)` are present:

```
2: get_kernels(equ,a);

{a(5),a(4),a(2),a(1)}
```

Or just one of them (if one wants to solve respect to some `a(i)`):

```
3: get_kernel(equ,a);

a(5)

4: linear_solve(equ,get_kernel(equ,a));

  - 10*a(4) - 3*a(2) - a(1) - 7
-----------------------------------
             15*beta
```

Or one would like to know all coefficients of the a(i):

```
4: operator_coeff(equ,a);

{7,{a(1),1},{a(2),3},{a(4),10},{a(5),15*beta}}
```

This type of operations are not very difficult to produce in REDUCE, however they are not present. The package TOOLS tries to fill in this gap. Apart from the operations above, we provide some procedures to handle expressions which are non-linear. In particular we provide a procedure multi_coeff which generalizes coeff and operator_coeff:

```
5: equ;

15*a(5)*beta + 10*a(4) + 3*a(2) + a(1) + 7

6: multi_coeff(equ,{a,beta});

{7,{a(1),1},{a(2),3},{a(4),10},{a(5)*beta,15}}
```

Finally we mention here the multilinear declaration:

```
6: multilinear d(a); % makes d linear w.r.t the operator a.

7: d(equ);

15*d(a(5))*beta + 10*d(a(4)) + 3*d(a(2)) + d(a(1)) + 7*d(1)
```

We refer to the next sections for a detailed description.

The package TOOLS was constructed by Marcel Roelofs, who made the versions 1.1 up to 1.17. Quite extensive additions made by the present author are included in version 2.1.

All procedures described here, are expected to be called from algebraic mode; the switch exp is expected to be on.

A final remark. The names used for the procedures are sometimes long. One can shorten them by using the define mechanism.

```
1:
define lsa=linear_solve_and_assign;

2:
lsa(a - 3,a);

3
```

2 Manipulating algebraic operators

A way to think of algebraic operators in REDUCE is as indexed variables; one has a collections of variables which more or less play the same role. Then it is natural to reflect this in the names of the variables. A way to do this is using an algebraic operator. If one has a large number of variables and a large number of expressions, it would be pleasant to have certain tools to handle all the information. Here we provide some procedures. We split the procedures in two types: the first part treats procedures which investigate expressions with respect to one or more operators, the second part treats procedures which give information on the use of one operator. The next section treats a way for aliasing variables as operator elements. Finally we describe a more efficient way to handle "large" operators, i.e. operators with many elements set or used.

2.1 Examining expressions

2.1.1 get_kernels

Call: `get_kernels(exprss,op_list)`

Goal: The polynomial expression `exprss` is scanned for the presence of operator kernels of the operators in `op_list`. Here `op_list` is either an algebraic list of `atoms` or an `atom`. All occurrences of the operator(s) are returned in an algebraic list.

Remarks: The expression `exprss` is expected to be a polynomial in the operator kernels of `op_list`. Therefore only the numerator is checked; the denominator is ignored. Moreover only top level presence is detected, for deeper levels you should use `get_deep_kernels`.

Examples:

```
1: get_kernels(a 2*b 2-3,b);

{b(2)}

2: get_kernels(a 2*b 2-3*a(1,2*a 3),{a,b});

{a(2),b(2),a(1,2*a(3))}

3: get_kernels(a 2*sqrt b 2-3*a(1,2*b 3),b);

{}
```

2.1.2 get_deep_kernels

Call: `get_deep_kernels(exprss,op_list)`
Goal: The expression `exprss` is scanned for the presence of operator kernels of the operators in `op_list`. Here `op_list` is either an algebraic list of `atoms` or an `atom`. All different occurrences of operator kernels are returned in an algebraic list.
Remark: The denominator of `exprss` is also considered.
Examples:

```
1:  get_deep_kernels(a 2*sqrt b 2-3*a(1,2*b 3),b);

{b(3),b(2)}

2: get_deep_kernels(a 2*b 2-3*a(1,2*b 3)/b(x),{a,b});

{a(1,2*b(3)),b(3),b(2),b(x),a(2)}

3:  get_deep_kernels(a(a(e^(a 2)/a 3))/a 4,a);

       a(2)        a(2)
      e           e
{a(a(-------)),a(-------),a(2),a(3),a(4)}
      a(3)        a(3)
```

2.1.3 get_kernel

Call: `get_kernel(exprss,op_list)`
Goal: The scalar expression `exprss` is scanned for the presence of operator kernels of the operators in `op_list`. Here `op_list` is either an algebraic list of `atoms` or an `atom`. The first encountered appearance is returned.
Remarks: The expression `exprss` is expected to be a polynomial in the elements of `op_list`. Therefore only the numerator is checked; the denominator is ignored; moreover only top level presence is detected.
Examples:

```
1: get_kernel(a 2*b 2-3,b);

b(2)

2: get_kernel(a 2*b 2-3,{a,b});
```

```
a(2)

3: get_kernel(sqrt a 2/a 3,a); % empty result
```

2.1.4 operator_coeff

Call: `operator_coeff(exprss,op_list)`

Goal: The linear expression `exprss` is decomposed with respect to the operator kernels from `op_list`. The result is an algebraic list of which the first element is the term not containing operator kernels from `op_list`, and the remaining terms are pairs (algebraic lists of two elements) of which the first element is the operator kernel, and the second element is its coefficient.

Remarks: The expression should be linear; for a non-linear variant see below, `multi_coeff`. An error occurs when `exprss` is not linear. Only top level appearance is found; deeper occurrences are ignored (without error message).

Examples:

```
1: operator_coeff(2*a(2)*b(3) + a(1,3)*c + a(1) + beta,a);

{beta,{a(1),1},{a(1,3),c},{a(2),2*b(3)}}

2: operator_coeff(b(3) + a(1,3)*c + a(1) + beta,{a,b});

{beta,{b(3),1},{a(1),1},{a(1,3),c}}

3: operator_coeff(sqrt a 2+a 3,a);

{sqrt(a(2)),{a(3),1}}
```

2.1.5 independent_part

Call: `independent_part(exprss,op_list)`

Goal: The terms of `exprss`, independent of `op_list`, are found.

Remarks: The denominator should be independent of `op_list`; only top level dependencies are checked. Hence the result is the same as `first operator_coeff(exprss,op_list)`.

Examples:

```
1: independent_part(a 2^3+a 2*b 3+b 4+23,{a,b});
```

```
23

2: independent_part(1/a 2,a);

***** INDEPENDENT_PART: denominator not independent
```

2.1.6 top_level_operators

Call: `top_level_operators(exprss)`
Goal: The operators appearing at top level in `exprss` are found.
Remark: Both, numerator and denominator are checked. The results can be used as second argument in `operator_coeff`.
Example:

```
1: top_level_operators((sqrt a 1+b 3*c 4+a1)/d 4);

{b,c,expt,d}  % sqrt is internally expt !!

2: equ:=13*a 3+15*b 5;

equ := 13*a(3) + 15*b(5)

3: operator_coeff(equ,top_level_operators equ);

{0,{b(5),15},{a(3),13}}
```

2.2 Diagnosis of an algebraic operator

Next we describe some procedures to retrieve information stored for an algebraic operator. These procedures are rather simple, but nevertheless quite useful.

2.2.1 known

Call: `known(opr)`
Goal: Returns the number of elements of `opr` that are known.
Remark: The number of known elements is taken to be the length of the `kvalue-list`.
Example:

```
1: a 1:=a 2+b 3;

a(1) := a(2) + b(3)

2: a 4:=a 7+a 8+3*a1;

a(4) := a(8) + a(7) + 3*a1

3: known a;

2
```

2.2.2 used

Call: `used(opr)`
Goal: Returns the number of elements of `opr` that are used.
Remark: The number of used elements is taken to be the length of the `klist`.
Example:

```
1: a 1:=a 2+b 3;

a(1) := a(2) + b(3)

2: a 4:=a 7+a 8+3*a1;

a(4) := a(8) + a(7) + 3*a1

3: used a;
```

2.2.3 clear_op

Call: clear_op a1,a2,...,an
Goal: Clears the operators a1,a2,...,an.
Remark: We allow clear_op also be used to clear elements of na_operators, see below.
Example:

```
1: a 1:=b 3+1;

a(1) := b(3) + 1

2: prop 'a;

((kvalue ((a 1) (!*sq (((((b 3) . 1) . 1) . 1) . 1) t)))
 (klist ((a 1) nil)) (simpfn . simpiden)
 (breakfunction . reset))

3: prop 'b;

((klist ((b 3) nil used!*)) (simpfn . simpiden))

4: clear_op a,b;

5:  prop 'a;  % empty result.
```

2.2.4 write_defs

Call: write_defs opr
Goal: Writes all known elements of the operator opr from the kvalue.
Remark: If the switch nat is off, then the written results, saved in a file, can be used as input for a new session.
Example:

```
1: a 1:=a 2+6;


a(1) := a(2) + 6
```

```
2: a 4:=a 7+a 8+3*a1;

a(4) := a(8) + a(7) + 3*a1

10: write_defs a;

a(1) := a(2) + 6

a(4) := a(8) + a(7) + 3*a1
```

2.2.5 reassign_defs

Call: `reassign_defs opr`
Goal: Assigns to all known elements of `opr` the current value, i.e. nested assignments are worked out.
Remark: After this command simplifications can be much faster.
Example:

```
1:  a 1:=a 2+6;

a(1) := a(2) + 6

2:  a 2:=a 3+10;

a(2) := a(3) + 10

3: get('a,'kvalue);

(((a 1) (!*sq (((((a 2) . 1) . 1) . 6) . 1) nil))
 ((a 2) (!*sq (((((a 3) . 1) . 1) . 10) . 1) nil)))

4: reassign_defs a;

5:  get('a,'kvalue);

(((a 1) (!*sq (((((a 3) . 1) . 1) . 16) . 1) nil))
 ((a 2) (!*sq (((((a 3) . 1) . 1) . 10) . 1) nil)))
```

2.3 Aliasing scalars as algebraic operator elements

Here we describe a mechanism to make operator elements look like `atoms`. The procedures described are purely devised for readability and aesthetical reasons. What you should be aware of is that the identifiers you type are internally operator elements (the parser makes the convertion). This way one can apply the procedures mentioned above for operators and one can work with non-commuting "optical" identifiers.

2.3.1 `operator_representation`

Call: `operator_representation(opr,alg_list_1,alg_list_2)`

Goal: To represent the `atoms` {`a_1, ... ,a_n`} , resp. {`b_1, ... ,b_m`} in the algebraic list `alg_list_1`, resp `alg_list_2` as elements of the operator `opr`. Here `a_1, ... ,a_n` are represented as `opr(1), ... ,opr(n)` and `b_1, ... ,b_m` are represented as `opr(-1), ... ,opr(-m)`.

The third argument `alg_list_2` is optional.

Remark: The reason to split the representation in positive and negative is that it can be profitable to distinguish between two types of arguments. Here we especially think of linear spaces with a $\mathbb{Z}_2$-grading ("odd" and "even" elements.)

Example:

```
1: operator_representation(x,{f1,f2},{g1,g2,g3});

2: multi_coeff(3+4*f1*g2+7*g3*g4,x);

{3,{g3,7*g4},{f1*g2,4}}

3: g3:=5;

g3 := 5

5: f1:=6*al;

f1 := 6*al

6: get('x,'kvalue);

(((x -3) 5) ((x 1) (!*sq ((((al . 1) . 6)) . 1) t)))

7: noncom x;

8: f2*g1-g1*f2;
```

```
f2*g1 - g1*f2

9: on defn;
10: f1;
(aeval (list 'x 1))
```

2.3.2 add_to_operator_representation

Call: `add_to_operator_representation(opr,alg_list_1,alg_list_2)`

Goal: To add the atoms {a_1, ... ,a_n} and {opt_a_1) ... ,opt_a_m} from `alg_list_1` and `alg_list_2` to the operator_representation of `opr`. Hence we assume that already some elements have been represented, say `opr(1)`, ... ,`opr(n)` and `opr(-1)`, ... ,`opr(-m)`.
Now a_1, ... ,a_r are represented as `opr(n+1)`, ... ,`opr(n+r)` and opt_a_1, ... ,opt_a_s as `opr(-m-1)`, ... ,`opr(-m-s)`.

The third argument `alg_list_2` is again optional.

Remark: Only adding is allowed. For other changes one first has to apply `clear_operator_representation`, after which one can restart by calling `operator_representation`.

Example:

```
1: operator_representation(x,{f1,f2});

2: add_to_operator_representation(x,{},{g1,g2,g3});

3: f1:=1;

f1 := 1

4: g1:=-1;

g1 := -1

5: get('x,'kvalue);

(((x 1) 1) ((x -1) -1))

6: clear g1;

7: g2:=-2;
```

```
g2 := -2

8:  get('x,'kvalue);

(((x 1) 1) ((x -2) -2))
```

2.3.3 clear_operator_representation

Call: `clear_operator_representation(opr)`
Goal: To remove the connection between the atoms and operator elements as set by `operator_representation`.
Remark: If values are assigned, these remain, i.e. the `klist` and `kvalue` of `opr` are not touched. (You can removed them by `clear_op`).
Example:

```
1: operator_representation(x,{f1,f2},{g1,g2,g3});

2: f1:=5;

f1 := 5

3: g3:=-10;

g3 := -10

4: prop 'x;

((alias_vector 3 2 . [g3 g2 g1 nil f1 f2])
 (kvalue ((x 1) 5) ((x -3) -10))
 (klist ((x 1) nil) ((x -3) nil))
 (prifn . print_alias) (simpfn . simpiden) used!*)

5: clear_operator_representation x;

6: prop 'x;

((kvalue ((x 1) 5) ((x -3) -10))
 (klist ((x 1) nil) ((x -3) nil))
 (simpfn . simpiden) used!*)
```

2.4 Nested association operators

The `klist` and `kvalue` mechanism is not very efficient for operators with many operator elements, as these lists are both search linearly. If there are several hundreds of elements it is better to look for other ways of handling such operators. Here we provide an example. Using it is very simple: instead of `operator a1,a2,...` one should do `na_operator a1,a2,...`. The procedure `clear_op` can be used to clear either operator elements or the whole operator at once.

2.4.1 `na_operator`

Call: `na_operator a_1, ... ,a_n`

Goal: To turn the `atoms a_1, ... ,a_n` into nested association operators. In particular the handling of the `klist` and `kvalue` is combined into the list `na_values`. Externally there are no differences between `na_operators` and `operators`. Only one should use `clear_op` (see below) instead of `clear` for removing values.

Remark: The alternative is especially meant for (extremely?) big operators. Real differences are found in case of 100 elements or more; for just a few elements (up to 50) the differences are small. We give a time comparison of a session with the Fibonacci sequence, and an example chosen especially to make `na_operators` look superb.

Care must be taken in turning operators, with special simplification functions, into `na_operators`. For example, the simplification function of `df` is `simpdf`, which uses explicitly the kvalue-list.

Example:

```
1: na_operator a;

2: operator b;

3: on time;

Time: 30 ms

4: for i:=2:10 do a i:=a(i-1)+a(i-2);

Time: 10 ms

5: for i:=2:10 do b i:=b(i-1)+b(i-2);

Time: 10 ms
```

```
6: for i:=2:50 do a i:=a(i-1)+a(i-2);

Time: 30 ms

7: for i:=2:50 do b i:=b(i-1)+b(i-2);

Time: 50 ms

8: for i:=2:100 do a i:=a(i-1)+a(i-2);

Time: 50 ms

9: for i:=2:100 do b i:=b(i-1)+b(i-2);

Time: 120 ms

10: for i:=2:1000 do a i:=a(i-1)+a(i-2);

Time: 1570 ms

11: for i:=2:1000 do b i:=b(i-1)+b(i-2);

Time: 7100 ms
```

A new session:

```
1: na_operator a;

2: operator b;

3: on time;

Time: 30 ms

4: for i:=1:20
   do for j:=1:20 do for k:=1:20 do a(1,j,k):=i*j*k;

Time: 1480 ms

5: for i:=1:20
   do for j:=1:20 do for k:=1:20 do b(1,j,k):=i*j*k;

Time: 260170 ms
```

2.4.2 clear_op

Call: `clear_op a_1,...,a_n`
Goal: To clear the values of the operator elements `a_1,...,a_n`
Remarks: If no value is assigned, a warning is printed. If one of the arguments `a_1,...,a_n` is an `atom`, it is assumed that it is an operator name; correspondingly this operator is cleared (see above).
Example:

```
1: na_operator a,b;

2: a 1:=4;

a(1) := 4

3: b 3:=7;

b(3) := 7

4: clear_op a 1,b 1,b 3;

*** CLEAR_OP: b(1) not found

5: b 3;

b(3)
```

3 Performing algebraic operations

Here we will describe some procedures for performing algebraic operations. The first procedure we describe is `multi_coeff`. It is an extension of `coeff` and `operator_coeff` at the same time. Hence it decomposes multi-variate expressions in (monic) monomials and their coefficient. The reason for keeping `operator_coeff` is that it is faster.

Next we describe two procedures, `multilinear` and `multimorph`, which makes an operator multilinear w.r.t some operators or variables. Their difference is that `multilinear` only works for operators and linear expressions in this operator, and `multimorph` works for non-linear expressions and variables as well.

3.1 multi_coeff

Call: `multi_coeff(exprss,vars)`

Goal: The expression **exprss** is decomposed with respect to **vars**. Here **vars** is an **atom**, representing a variable or an operator name or **vars** is an algebraic list of such. The result is an algebraic list of which the first element is the part independent of **vars**, and the remaining list are pairs, of which the first element is the monomial in **vars** and the second element the corresponding coefficient. Non-commuting variables in **vars** are allowed.

Remarks: The expression **exprss** is expected to be a polynomial in the elements of **vars**. Therefore only the numerator is checked; the denominator should be independent of **vars**. IMPORTANT: the *coefficients* must be commuting objects. If not, all kind of strange results may occur.

Examples:

```
1: multi_coeff(3+4*x 5+6*x 7*x 8+9*y+10*a*x 11,{x,y});

{3, {y,9}, {x(5),4}, {x(8)*x(7),6}, {x(11),10*a}}

2: noncom x;

3: multi_coeff(x 1*x 2+3*x 4*x 5+x 2*x 1+7,x);

{7,{x(1)*x(2),1},{x(2)*x(1),1},{x(4)*x(5),3}}

4: multi_coeff(x 1+3*x^2,x); % even this works.

     2
{0,{x ,3},{x(1),1}}


5: noncom a;   % so non-commuting coefficients!

6: multi_coeff(a 1*x 1-x 1*a 1,x);

{0,{x(1),0}}  % this is not what you want, probably, while:

7: multi_coeff(a 1*x 1-x 1*a 1,{a,x});

{0,{x(1)*a(1),-1},{a(1)*x(1),1}} % this is OK.
```

3.2 multilinear

Call: `multilinear map(op_list [,resimp_function])`

Goal: The call turns the mapping `map` into a multilinear operator. Hence `map` is used in the form `map(arg_1,arg_2, ... , arg_n)`. Here the arguments are elements of a linear space: products of scalars with vectors, where the vectors are represented by operator elements from the operators of `op_list`, an algebraic list of `atoms`. The denominator is ignored. The role of the `resimp_function` is explained later. Let us first give some examples.

Examples:

```
1: multilinear d({x,y});

2: operator x,y;

3:  d(x 1+3*x 2);

3*d(x(2)) + d(x(1))

4:  d(al*x(2,1)+y 4,al*x(2,1)+y 4);

                    2
d(x(2,1),x(2,1))*al  + d(x(2,1),y(4))*al + d(y(4),x(2,1))*al

 + d(y(4),y(4))

5: for all ii,jj such that ii neq jj let d(x ii,x jj)=0;

6: for all ii let d(x ii,x ii)=1;

7: d(x 1+2*x 2+x 3,3*x 1 + 4*x 2+ 5*x 3);

16

8: d(x 1*x 2,x 3);  % if you really want this, see multimorph.

***** SPLIT_F: expression not linear w.r.t. {x,y}

9: d(1/x 1);  % take care of this.

 d(1)
------
 x(1)
```

Goal (continued): The third argument in the call of multilinear is optional. If it is present, it is called the resimp_function of map. To explain the use, let us explain the method of multilinear. To map is given a new simplification function (instead of the default simpiden), namely simp_multilinear. This takes care of the multilinear splitting. If this is finished we are left with a sum of expressions, which are now offered to simpiden, unless map has a resimp_function. Hence the resimp_function takes care of the elementary pieces. In the last example above (where $\{ x_i \}$ is made orthonormal), the resimp_function would take care of $d(x_i, x_j)$. As an example let us give a resimp_function to map which has the same effect: d(x(1),x(j)) is made orthonormal, all other expressions are left untouched.

```
1:
lisp procedure resimp_d u;   % NB: the car of u is just d.
begin scalar arg1,arg2;
 if length cdr u neq 2       % The cdr of u are the args of d.
 then return simpiden u;     % We only act if d has 2 args.
 arg1:=cadr u;
 arg2:=caddr u;              % There are exactly 2 arguments.
 if atom arg1 or atom arg2
 then return simpiden u;     % The arguments can be 1.
 if car arg1 neq 'x
   or car arg2 neq 'x        % check if both args are x(..):
 then return simpiden u;     % if not, we do nothing.
 return                      % the result is either 0 or 1:
 if cadr arg1=cadr arg2
 then (1 . 1)                % 1 as sqform (like simpiden)!!
 else (nil . 1)              % 0 as sqform.
end$

2: operator x,y;

3: multilinear d({x,y},resimp_d);

4: d(x 1+2*x 2+x 3,3*x 1 + 4*x 2+ 5*x 3);

16

5: d(x 1, y 3);

d(x(1),y(3))

6: d(x 1, y3);
```

```
d(x(1),1)*y3
```

To summarize: the resimp_function gets as argument `map(a_1, ... ,a_n)`, and it should return an sqform.

3.3 multimorph

Call: `multimorph map(op_list [,resimp_function])`

Goal: The call turns the mapping `map` into a multilinear operator in an *algebra*. Hence `map` is used in the form `map(arg_1,arg_2, ... ,arg_n)`. Here the arguments are elements of an algebra : products of scalars with algebra elements, which are represented by (products of) operator elements from the operators of `op_list`, an algebraic list of `atoms`. The denominator is ignored. The role of the `resimp_function` is similar like in `multilinear` (see below).

Examples:

```
1: operator x,y,

2: multimorph d({x,y});

3: d(x 1*x 2+3*x 3,y 3+a1*y 4);

d(x(2)*x(1),y(4))*a1 + d(x(2)*x(1),y(3)) + 3*d(x(3),y(4))*a1

 + 3*d(x(3),y(3))

4: d(x 1,x 2*x 3):=23;

d(x(1),x(3)*x(2)) := 23

5: d(x 1,1+3*x 2*x 3);

d(x(1),1) + 69
```

Goal (continued): The resimp_function works in the same way as in multilinear. As a useful example, we turn d into a morphism of algebras.

```
1:
lisp procedure algebra_morphism(u);
resimp_algebra_morphism1(cadr u,car u)$
```

```
2:
lisp procedure resimp_algebra_morphism1(exprss,morphism);
if atom exprss
then simpiden list(morphism,exprss)
else if car exprss='times
   then algebra_morphism_times(cdr exprss,morphism)
   else if car exprss='expt
       then (algebra_morphism_expt(arg,caddr exprss,morphism))
      where arg=resimp_algebra_morphism1(cadr exprss,morphism)
       else simpiden list(morphism,exprss)$

3:
lisp procedure algebra_morphism_times(u,morphism);
if null u
then 1 . 1
else multsq(resimp_algebra_morphism1(car u,morphism),
            algebra_morphism_times(cdr u,morphism))$

4:
lisp procedure algebra_morphism_expt(u,n,morphism);
if n=1 then u
else multsq(u,algebra_morphism_expt(u,n-1,morphism))$

5: multimorph d(x,algebra_morphism);

6: operator x;

7: d(x 1*x 2*x 3 + 4*x 5^6 + 7);

         6
4*d(x(5))  + d(x(3))*d(x(2))*d(x(1)) + 7*d(1)

% note that the order is changed; all objects are commuting:

8: noncom d;

9: d(x 1*x 2);

d(x(2))*d(x(1))   % so noncom d is not enough:
                  % x 1*x 2 evaluates to x 2*x 1.

10: noncom d,x;

11: d(x 1*x 2);
```

```
d(x(1))*d(x(2))
```

4 Solving linear equations

In our programs we want to do a lot of automated computations on algebraic expressions containing algebraic operators. In particular we think it is convenient to have, together with the procedures `linear_solve` and `linear_solve_and_assign`, a procedure that searches an algebraic expression for kernels of some specified operator with respect to which the algebraic expression is linear, but with the coefficients of these kernels not depending on some other operators.

Let us give an example in which such a procedure can be used fruitfully. Suppose we have an expression `a(3)*a(2)-a(1)` from which we want to solve one the `a(i)`'s automatically. Taking the first operator element at sight, we would get `a(3):=a(1)/a(2)`. This, however, is undesirable, because `a(2)` may be equated to 0 during the process, in which case we are in trouble. Therefore the solution should be `a(1):=a(3)*a(2)`.

But how can we discover that we must solve for `a(1)`? The answer to this question is to use the procedure `solvable_kernels`, which we will specify in a moment: the call `solvable_kernels(a(3)*a(2)-a(1),a,a)` searches the expression `a(3)*a(2)-a(1)` for kernels of operator `a` (second argument), but only those which don't have coefficients containing kernels of operator `a` (third argument). Hence this call returns the list `a(1)` which is exactly the list of all kernels for which we may solve without risc.

4.1 `linear_solve`

Call: `linear_solve(exprn,var)`

Goal: Solve the equation `exprn = 0` with respect to the kernel `var`. Here `var` should occur linearly. The solution is *not* assigned to `var` (like in `solve`).

Remarks: The procedure first factorizes the numerator of `exprn`; the denominator is ignored. The factors independent of `var` and also *double* factors are removed. Only after this linear solving is tried.

Examples:

```
1:  linear_solve(a - 3,a);

3
```

```
2: linear_solve(x*(a-3),a);

3

3: linear_solve((a-3)^100,a);

3
```

4.2 linear_solve_and_assign

Call: `linear_solve_and_assign(exprn,var)`
Goal: Solve the equation `exprn = 0` with respect to the kernel `var`. Here `var` should occur linearly. The solution is assigned to `var`.
Remark: The procedure first applies `linear_solve`, see above. Thereafter this value is assigned to `var`.
Examples:

```
1: linear_solve_and_assign(a - 3,a);

3

2: a;

3
```

4.3 solvable_kernels

Call: `solvable_kernels(exprn,oplist,bad_oplist)`
Goal: Here `oplist` and `bad_oplist` and algebraic lists of operator names. The expression `exprn` is checked for linear occurences of operator elements of `oplist`. These operator elements are selected. If the coefficient of such an element contains operator elements from `bad_oplist`, then such element is removed.
Examples:

```
1: solvable_kernels(a 1*a 2 - a 3*c 4 + a 4,{a},{c});

{a(1),a(2),a(4)}

2: solvable_kernels(a 1*a 2 - a 3*c 4 + a 4,{a}, a );
```

```
{a(3),a(4)}

3: solvable_kernels(a 1*a 2 - a 3*c 4 + a 4, a ,{a,c});

{a(4)}
```

Remark: This allows to make an automatic solver. We give an example of it. We assume the equations are stored in an operator equ in particular in the operator elements equ(i_1), ... , equ(i_n). Hence we use equ globally. The variable equ_list is the algebraic list { i_1, ... , i_n } . We solve for a, the second argument, an operator name, and forbid the operators from ac, the third argument. The result is the list of unsolved equations.

```
1:
procedure lin_solve_equ(equ_list,a,ac);
begin scalar vars,curr_equ,unsolved_list;
      unsolved_list:={};
      for each j in equ_list
      do << curr_equ:=equ j;
            vars:=solvable_kernels(curr_equ,a,ac);
            if length vars > 0
           then linear_solve_and_assign(curr_equ,first vars)
           else unsolved_list:= j . unsolved_list
        >>;
    if unsolved_list={}
    then return;
    write "The following equations from equ are not solved:";
    return reverse unsolved_list
end$

2: equ(1):=c 1*a 2-3;

equ(1) := a(2)*c(1) - 3

3: equ(3):=a 4-7;

equ(3) := a(4) - 7

4: equ(10):= 3*a 2-5;

5: lin_solve_equ({1,3,10}, a ,{a,c});

The following equations from equ are not solved:
```

```
{1}             % this, {1}, is the returned value.

6: equ 1;       % can be solved now, at least with respect to c.

 5*c(1) - 9
------------
     3

7: lin_solve_equ(ws 5,c,c);

8: write_defs a;

a(4) := 7

         5
a(2) := ---
         3

9: write_defs c;

         9
c(1) := ---
         5

END OF SESSION, end of manual.
```

Index