16.5 ASSIST: Useful utilities for various applications

ASSIST contains a large number of additional general purpose functions that allow a user to better adapt REDUCE to various calculational strategies and to make the programming task more straightforward and more efficient.

Author: Hubert Caprasse.

16.5.1 Introduction

The package ASSIST contains an appreciable number of additional general purpose functions which allow one to better adapt REDUCE to various calculational strategies, to make the programming task more straightforward and, sometimes, more efficient.

In contrast with all other packages, ASSIST does not aim to provide either a new facility to compute a definite class of mathematical objects or to extend the base of mathematical knowledge of REDUCE . The functions it contains should be useful independently of the nature of the application which is considered. They were initially written while applying REDUCE to specific problems in theoretical physics. Most of them were designed in such a way that their applicability range is broad. Though it was not the primary goal, efficiency has been sought whenever possible.

The source code in ASSIST contains many comments concerning the meaning and use of the supplementary functions available in the algebraic mode. These comments, hopefully, make the code transparent and allow a thorough exploitation of the package. The present documentation contains a non–technical description of it and describes the various new facilities it provides.

16.5.2 Survey of the Available New Facilities

An elementary help facility is available both within the MS-DOS and Windows environments. It is independent of the help facility of REDUCE itself. It includes two functions:

`ASSIST` is a function which takes no argument. If entered, it returns the informations required for a proper use of `ASSISTHELP`.
`ASSISTHELP` takes one argument.

i. If the argument is the identifier `assist`, the function returns the information necessary to retrieve the names of all the available functions.

ii. If the argument is an integer equal to one of the section numbers of the present documentation. The names of the functions described in that section are obtained.

There is, presently, no possibility to retrieve the number and the type of the arguments of a given function.

The package contains several modules. Their content reflects closely the various categories of facilities listed below. Some functions do already exist inside the KERNEL of REDUCE . However, their range of applicability is *extended*.

- Control of Switches:

```
SWITCHES SWITCHORG
```

- Operations on Lists and Bags:

```
MKLIST KERNLIST ALGNLIST LENGTH
POSITION FREQUENCY SEQUENCES SPLIT
INSERT INSERT_KEEP_ORDER MERGE_LIST
FIRST SECOND THIRD REST REVERSE LAST
BELAST CONS ( . ) APPEND APPENDN
REMOVE DELETE DELETE_ALL DELPAIR
MEMBER ELMULT PAIR DEPTH MKDEPTH_ONE
REPFIRST REPREST ASFIRST ASLAST ASREST
ASFLIST ASSLIST RESTASLIST SUBSTITUTE
BAGPROP PUTBAG CLEARBAG BAGP BAGLISTP
ALISTP ABAGLISTP LISTBAG
```

- Operations on Sets:

```
MKSET SETP UNION INTERSECT DIFFSET SYMDIFF
```

- General Purpose Utility Functions:

```
LIST_TO_IDS MKIDN MKIDNEW DELLASTDIGIT DETIDNUM
ODDP FOLLOWLINE == RANDOMLIST MKRANDTABL
PERMUTATIONS CYCLICPERMLIST PERM_TO_NUM NUM_TO_PERM
COMBNUM COMBINATIONS SYMMETRIZE REMSYM
SORTNUMLIST SORTLIST ALGSORT EXTREMUM GCDNL
DEPATOM FUNCVAR IMPLICIT EXPLICIT REMNONCOM
KORDERLIST SIMPLIFY CHECKPROPLIST EXTRACTLIST
```

- Properties and Flags:

```
PUTFLAG PUTPROP DISPLAYPROP DISPLAYFLAG
CLEARFLAG CLEARPROP
```

- Control Statements, Control of Environment:

```
NORDP DEPVARP ALATOMP ALKERNP PRECP
SHOW SUPPRESS CLEAROP CLEARFUNCTIONS
```

- Handling of Polynomials:

```
ALG_TO_SYMB SYMB_TO_ALG
DISTRIBUTE LEADTERM REDEXPR MONOM
LOWESTDEG DIVPOL SPLITTERMS SPLITPLUSMINUS
```

- Handling of Transcendental Functions:

```
TRIGEXPAND HYPEXPAND TRIGREDUCE HYPREDUCE
```

- Coercion from Lists to Arrays and converse:

```
LIST_TO_ARRAY ARRAY_TO_LIST
```

- Handling of n-dimensional Vectors:

```
SUMVECT MINVECT SCALVECT CROSSVECT MPVECT
```

- Handling of Grassmann Operators:

```
PUTGRASS REMGRASS GRASSP GRASSPARITY GHOSTFACTOR
```

- Handling of Matrices:

```
UNITMAT MKIDM BAGLMAT COERCEMAT
SUBMAT MATSUBR MATSUBC RMATEXTR RMATEXTC
HCONCMAT VCONCMAT TPMAT HERMAT
SETELTMAT GETELTMAT
```

- Control of the HEPHYS package:

```
REMVECTOR REMINDEX MKGAM
```

In the following all these functions are described.

16.5.3 Control of Switches

The two available functions i.e. `SWITCHES, SWITCHORG` have no argument and are called as if they were mere identifiers.

`SWITCHES` displays the actual status of the most frequently used switches when manipulating rational functions. The chosen switches are

```
EXP, ALLFAC, EZGCD, GCD, MCD, LCM, DIV, RAT,
INTSTR, RATIONAL, PRECISE, REDUCED, RATIONALIZE,
COMBINEEXPT, COMPLEX, REVPRI, DISTRIBUTE.
```

The selection is somewhat arbitrary but it may be changed in a trivial fashion by the user.

The new switch DISTRIBUTE allows one to put polynomials in a distributed form (see the description below of the new functions for manipulating them.).

Most of the symbolic variables !*EXP, !*DIV, ... which have either the value T or the value NIL are made available in the algebraic mode so that it becomes possible to write conditional statements of the kind

```
IF !*EXP THEN DO ......

IF !*GCD THEN OFF GCD;
```

SWITCHORG resets the switches enumerated above to the status they had when **starting** REDUCE .

16.5.4 Manipulation of the List Structure

Additional functions for list manipulations are provided and some already defined functions in the kernel of REDUCE are modified to properly generalize them to the available new structure BAG.

i. Generation of a list of length n with all its elements initialized to 0 and possibility to append to a list l a certain number of zero's to make it of length n:

```
MKLIST n ;     n is an INTEGER

MKLIST(l,n);     l is List-like, n is an INTEGER
```

ii. Generation of a list of sublists of length n containing p elements equal to 0 and q elements equal to 1 such that

$$p + q = n.$$

The function SEQUENCES works both in algebraic and symbolic modes. Here is an example in the algebraic mode:

```
SEQUENCES 2 ; ==> {{0,0},{0,1},{1,0},{1,1}}
```

An arbitrary splitting of a list can be done. The function SPLIT generates a list which contains the splitted parts of the original list.

```
SPLIT({a,b,c,d},{1,1,2}) ==> {{a},{b},{c,d}}
```

The function ALGNLIST constructs a list which contains n copies of a list bound to its first argument.

```
ALGNLIST({a,b,c,d},2); ==> {{a,b,c,d},{a,b,c,d}}
```

The function KERNLIST transforms any prefix of a kernel into the list prefix. The output list is a copy:

```
KERNLIST (<kernel>); ==> {<kernel arguments>}
```

Four functions to delete elements are DELETE, REMOVE, DELETE_ALL and DELPAIR. The first two act as in symbolic mode, and the third eliminates from a given list *all* elements equal to its first argument. The fourth acts on a list of pairs and eliminates from it the *first* pair whose first element is equal to its first argument :

```
DELETE(x,{a,b,x,f,x}); ==> {a,b,f,x}

REMOVE({a,b,x,f,x},3); ==> {a,b,f,x}

DELETE_ALL(x,{a,b,x,f,x}); ==> {a,b,f}

DELPAIR(a,{{a,1},{b,2},{c,3}}; ==> {{b,2},{c,3}}
```

iv. The function ELMULT returns an *integer* which is the *multiplicity* of its first argument inside the list which is its second argument. The function FREQUENCY gives a list of pairs whose second element indicates the number of times the first element appears inside the original list:

```
ELMULT(x,{a,b,x,f,x}) ==> 2

FREQUENCY({a,b,c,a}); ==> {{a,2},{b,1},{c,1}}
```

v. The function INSERT allows one to insert a given object into a list at the desired position.

The functions INSERT_KEEP_ORDER and MERGE_LIST allow one to keep a given ordering when inserting one element inside a list or when merging two lists. Both have 3 arguments. The last one is the name of a binary boolean ordering function:

```
ll:={1,2,3}$

INSERT(x,ll,3); ==> {1,2,x,3}

INSERT_KEEP_ORDER(5,ll,lessp); ==> {1,2,3,5}

MERGE_LIST(ll,ll,lessp); ==> {1,1,2,2,3,3}
```

Notice that MERGE_LIST will act correctly only if the two lists are well ordered themselves.

vi. Algebraic lists can be read from right to left or left to right. They *look* symmetrical. One would like to dispose of manipulation functions which reflect this. So, to the already defined functions FIRST and REST are added the functions LAST and BELAST. LAST gives the last element of the list while BELAST gives the list *without* its last element.
Various additional functions are provided. They are:

```
  .  ("dot"), POSITION, DEPTH, MKDEPTH_ONE,
PAIR, APPENDN, REPFIRST, REPREST
```

The token "dot" needs a special comment. It corresponds to several different operations.

1. If one applies it on the left of a list, it acts as the CONS function. Note however that blank spaces are required around the dot:

```
4 . {a,b}; ==> {4,a,b}
```

2. If one applies it on the right of a list, it has the same effect as the PART operator:

```
{a,b,c}.2; ==> b
```

3. If one applies it to a 4–dimensional vectors, it acts as in the HEPHYS package.

POSITION returns the POSITION of the first occurrence of x in a list or a message if x is not present in it.

DEPTH returns an *integer* equal to the number of levels where a list is found if and only if this number is the *same* for each element of the list otherwise it returns a message telling the user that the list is of *unequal depth*. The function MKDEPTH_ONE allows to transform any list into a list of depth equal to 1.

PAIR has two arguments which must be lists. It returns a list whose elements are *lists of two elements.* The n^{th} sublist contains the n^{th} element of the first list and the n^{th} element of the second list. These types of lists are called *association lists* or ALISTS in the following. To test for these type of lists a boolean function ABAGLISTP is provided. It will be discussed below.
APPENDN has *any* fixed number of lists as arguments. It generalizes the already existing function APPEND which accepts only two lists as arguments. It may also be used for arbitrary kernels but, in that case, it is important to notice that *the concatenated object is always a list*.
REPFIRST has two arguments. The first one is any object, the second one is a list. It replaces the first element of the list by the object. It works like the symbolic function REPLACA except that the original list is not destroyed.
REPREST has also two arguments. It replaces the rest of the list by its first argument and returns the new list *without destroying* the original list. It is analogous to the symbolic function REPLACD. Here are examples:

```
ll:={{a,b}}$
ll1:=ll.1; ==> {a,b}
ll.0; ==> list
0 . ll; ==> {0,{a,b}}

DEPTH ll; ==> 2

PAIR(ll1,ll1); ==> {{a,a},{b,b}}

REPFIRST{new,ll); ==> {new}

ll3:=APPENDN(ll1,ll1,ll1); ==> {a,b,a,b,a,b}

POSITION(b,ll3); ==> 2

REPREST(new,ll3); ==> {a,new}
```

vii. The functions ASFIRST, ASLAST, ASREST, ASFLIST, ASSLIST, RESTASLIST act on ALISTS or on lists of lists of well defined depths and have two arguments. The first is the key object which one seeks to associate in some way with an element of the association list which is the second argument.
ASFIRST returns the pair whose first element is equal to the first argument.
ASLAST returns the pair whose last element is equal to the first argument.
ASREST needs a *list* as its first argument. The function seeks the first sublist of a list of lists (which is its second argument) equal to its first argument and returns it.
RESTASLIST has a *list of keys* as its first argument. It returns the collection of pairs which meet the criterium of ASREST.
ASFLIST returns a list containing *all pairs* which satisfy the criteria of the function ASFIRST. So the output is also an association list.
ASSLIST returns a list which contains *all pairs* which have their second element equal to the first argument.
Here are a few examples:

```
lp:={{a,1},{b,2},{c,3}}$

ASFIRST(a,lp); ==> {a,1}

ASLAST(1,lp); ==> {a,1}

ASREST({1},lp); ==> {a,1}

RESTASLIST({a,b},lp); ==> {{1},{2}}

lpp:=APPEND(lp,lp)$

ASFLIST(a,lpp); ==> {{a,1},{a,1}}

ASSLIST(1,lpp); ==> {{a,1},{a,1}}
```

vii. The function SUBSTITUTE has three arguments. The first is the object to be substituted, the second is the object which must be replaced by the first, and the third is the list in which the substitution must be made. Substitution is made to all levels. It is a more elementary function than SUB but its capabilities are less. When dealing with algebraic quantities, it is important to make sure that *all* objects involved in the function have either the prefix lisp or the standard quotient representation otherwise it will not properly work.

16.5.5 The Bag Structure and its Associated Functions

The LIST structure of REDUCE is very convenient for manipulating groups of objects which are, a priori, unknown. This structure is endowed with other properties such as "mapping" i.e. the fact that if `OP` is an operator one gets, by default,

```
OP({x,y}); ==> {OP(x),OP(y)}
```

It is not permitted to submit lists to the operations valid on rings so that, for example, lists cannot be indeterminates of polynomials.
Very frequently too, procedure arguments cannot be lists. At the other extreme, so to say, one has the `KERNEL` structure associated with the algebraic declaration `operator` . This structure behaves as an "unbreakable" one and, for that reason, behaves like an ordinary identifier. It may generally be bound to all non-numeric procedure parameters and it may appear as an ordinary indeterminate inside polynomials.
The `BAG` structure is intermediate between a list and an operator. From the operator it borrows the property of being a `KERNEL` and, therefore, may be an indeterminate of a polynomial. From the list structure it borrows the property of being a *composite* object.

Definition:

A bag is an object endowed with the following properties:

1. It is a `KERNEL` i.e. it is composed of an atomic prefix (its envelope) and its content (miscellaneous objects).
2. Its content may be handled in an analogous way as the content of a list. The important difference is that during these manipulations the name of the bag is *kept*.
3. Properties may be given to the envelope. For instance, one may declare it `NONCOM` or `SYMMETRIC` etc. . . .

Available Functions:

i. A default bag envelope `BAG` is defined. It is a reserved identifier. An identifier other than `LIST` or one which is already associated with a boolean function may be defined as a bag envelope through the command `PUTBAG`. In particular, any operator may also be declared to be a bag. **When and only when** the identifier is not an already defined function does `PUTBAG` put on it the property of an OPERATOR PREFIX. The command:

```
PUTBAG id1,id2,....idn;
```

declares `id1,.....,idn` as bag envelopes. Analogously, the command

```
CLEARBAG id1,...idn;
```

eliminates the bag property on `id1,...,idn`.

ii. The boolean function `BAGP` detects the bag property. Here is an example:

```
aa:=bag(x,y,z)$

if BAGP aa then "ok"; ==> ok
```

iii. The functions listed below may act both on lists or bags. Moreover, functions subsequently defined for SETS also work for a bag when its content is a set. Here is a list of the main ones:

```
FIRST, SECOND, LAST, REST, BELAST, DEPTH, LENGTH,
REVERSE,
MEMBER, APPEND, .  ("dot"), REPFIRST, REPREST
...
```

However, since they keep track of the envelope, they act somewhat differently. Remember that

the NAME of the ENVELOPE is KEPT by the functions
`FIRST, SECOND and LAST.`

Here are a few examples (more examples are given inside the test file):

```
PUTBAG op; ==> T

aa:=op(x,y,z)$

FIRST op(x,y,z); ==> op(x)
```

```
REST op(x,y,z); ==> op(y,z)

BELAST op(x,y,z); ==> op(x,y)

APPEND(aa,aa); ==> op(x,y,z,x,y,z)

APPENDN(aa,aa,aa); ==> {x,y,z,x,y,z,x,y,z}

LENGTH aa; ==> 3

DEPTH aa; ==> 1

MEMBER(y,aa); ==> op(y,z)
```

When "appending" two bags with *different* envelopes, the resulting bag gets the name of the one bound to the first parameter of APPEND. When APPENDN is used, the output is always a list.
The function LENGTH gives the number of objects contained in the bag.

iv. The connection between the list and the bag structures is made easy thanks to KERNLIST which transforms a bag into a list and thanks to the coercion function LISTBAG which transforms a list into a bag. This function has 2 arguments and is used as follows:

```
LISTBAG(<list>,<id>); ==> <id>(<arg_list>)
```

The identifier <id>, if allowed, is automatically declared as a bag envelope or an error message is generated.

Finally, two boolean functions which work both for bags and lists are provided. They are BAGLISTP and ABAGLISTP. They return t or nil (in a conditional statement) if their argument is a bag or a list for the first one, or if their argument is a list of sublists or a bag containing bags for the second one.

16.5.6 Sets and their Manipulation Functions

Functions for sets exist at the level of symbolic mode. The package makes them available in algebraic mode but also *generalizes* them so that they can be applied to bag-like objects as well.

i. The constructor MKSET transforms a list or bag into a set by eliminating duplicates.

```
MKSET({1,a,a}); ==> {1,a}
MKSET bag(1,a,1,a); ==> bag(1,a)
```

SETP is a boolean function which recognizes set–like objects.

```
if SETP {1,2,3} then ... ;
```

ii. The available functions are

```
UNION, INTERSECT, DIFFSET, SYMDIFF.
```

They have two arguments which must be sets otherwise an error message is issued. Their meaning is transparent from their name. They respectively give the union, the intersection, the difference and the symmetric difference of two sets.

16.5.7 General Purpose Utility Functions

Functions in this sections have various purposes. They have all been used many times in applications in some form or another. The form given to them in this package is adjusted to maximize their range of applications.

i. The functions MKIDNEW DELLASTDIGIT DETIDNUM LIST_TO_IDS handle identifiers.

MKIDNEW has either 0 or 1 argument. It generates an identifier which has not yet been used before.

```
MKIDNEW(); ==> g0001

MKIDNEW(a); ==> ag0002
```

DELLASTDIGIT takes an integer as argument and strips it from its last digit.

```
DELLASTDIGIT 45; ==> 4
```

DETIDNUM deletes the last digit from an identifier. It is a very convenient function when one wants to make a do loop starting from a set of indices $a_1, \ldots, a_n$.

```
DETIDNUM a23; ==> 23
```

LIST_to_IDS generalizes the function MKID to a list of atoms. It creates and intern an identifier from the concatenation of the atoms. The first atom cannot be an integer.

```
LIST_TO_IDS {a,1,id,10}; ==> a1id10
```

The function ODDP detects odd integers.

The function FOLLOWLINE is convenient when using the function PRIN2. It allows one to format output text in a much more flexible way than with the function WRITE.
Try the following examples :

```
<<prin2 2; prin2 5>>$ ==> ?

<<prin2 2; followline(5); prin2 5;>>; ==> ?
```

The function == is a short and convenient notation for the SET function. In fact it is a *generalization* of it to allow one to deal also with KERNELS:

```
operator op;

op(x):=abs(x)$

op(x) == x; ==> x

op(x); ==> x

abs(x); ==> x
```

The function RANDOMLIST generates a list of random numbers. It takes two arguments which are both integers. The first one indicates the range inside which the random numbers are chosen. The second one indicates how many numbers are to be generated. Its output is the list of generated numbers.

```
RANDOMLIST(10,5); ==> {2,1,3,9,6}
```

MKRANDTABL generates a table of random numbers. This table is either a one or two dimensional array. The base of random numbers may be either an integer or a decimal number. In this last case, to work properly, the switch rounded must be ON. It has three arguments. The first is either a one integer or a two integer list. The second is the base chosen to generate the random numbers. The third is the chosen name for the generated array. In the example below a two-dimensional table of random integers is generated as array elements of the identifier ar.

```
MKRANDTABL({3,4},10,ar); ==>

        *** array ar redefined

                {3,4}
```

The output is the dimension of the constructed array.

PERMUTATIONS gives the list of permutations of n objects. Each permutation is itself a list. CYCLICPERMLIST gives the list of *cyclic* permutations. For both functions, the argument may also be a bag.

```
PERMUTATIONS {1,2} ==> {{1,2},{2,1}}

CYCLICPERMLIST {1,2,3} ==>

          {{1,2,3},{2,3,1},{3,1,2}}
```

PERM_TO_NUM and NUM_TO_PERM allow to associate to a given permutation of n numbers or identifiers a number between 0 and $n! - 1$. The first function has the two permutated lists as its arguments and it returns an integer. The second one has an integer as its first argument and a list as its second argument. It returns the list of permutated objects.

```
PERM_TO_NUM({4,3,2,1},{1,2,3,4}) ==> 23

NUM_TO_PERM(23,{1,2,3,4}); ==> {4,3,2,1}
```

COMBNUM gives the number of combinations of n objects taken p at a time. It has the two integer arguments n and p.

COMBINATIONS gives a list of combinations on n objects taken p at a time. It has two arguments. The first one is a list (or a bag) and the second one is the integer p.

```
COMBINATIONS({1,2,3},2) ==> {{2,3},{1,3},{1,2}}
```

REMSYM is a command that suppresses the effect of the REDUCE commands symmetric or antisymmetric .

SYMMETRIZE is a powerful function which generates a symmetric expression. It has 3 arguments. The first is a list (or a list of lists) containing the expressions which will appear as variables for a kernel. The second argument is the kernel-name and the third is a permutation function which exists either in algebraic or symbolic mode. This function may be constructed by the user. Within this package the two functions PERMUTATIONS and CYCLICPERMLIST may be used. Examples:

```
ll:={a,b,c}$

SYMMETRIZE(ll,op,cyclicpermlist); ==>

        OP(A,B,C) + OP(B,C,A) + OP(C,A,B)

SYMMETRIZE(list ll,op,cyclicpermlist); ==>

        OP({A,B,C}) + OP({B,C,A}) + OP({C,A,B})
```

Notice that, taking for the first argument a list of lists gives rise to an expression where each kernel has a *list as argument*. Another peculiarity of this function is the fact that, unless a pattern matching is made on the operator OP, it needs to be reevaluated. This peculiarity is convenient when OP is an abstract operator if one wants to control the subsequent simplification process. Here is an illustration:

```
op(a,b,c):=a*b*c$

SYMMETRIZE(ll,op,cyclicpermlist); ==>

          OP(A,B,C) + OP(B,C,A) + OP(C,A,B)

REVAL ws; ==>

          OP(B,C,A) + OP(C,A,B) + A*B*C

for all x let op(x,a,b)=sin(x*a*b);

SYMMETRIZE(ll,op,cyclicpermlist); ==>

           OP(B,C,A) + SIN(A*B*C) + OP(A,B,C)
```

The functions SORTNUMLIST and SORTLIST are functions which sort lists. They use the *bubblesort* and the *quicksort* algorithms.

SORTNUMLIST takes as argument a list of numbers. It sorts it in increasing order.

SORTLIST is a generalization of the above function. It sorts the list according to any well defined ordering. Its first argument is the list and its second argument is the ordering function. The content of the list need not necessarily be numbers but must be such that the ordering function has a meaning. ALGSORT exploits the PSL SORT function. It is intended to replace the two functions above.

```
l:={1,3,4,0}$  SORTNUMLIST l; ==> {0,1,3,4}

ll:={1,a,tt,z}$ SORTLIST(ll,ordp); ==> {a,z,tt,1}

l:={-1,3,4,0}$  ALGSORT(l,>); ==> {4,3,0,-1}
```

It is important to realise that using these functions for kernels or bags may be dangerous since they are destructive. If it is necessary, it is recommended to first apply KERNLIST to them to act on a copy.

The function EXTREMUM is a generalization of the already defined functions MIN, MAX to include general orderings. It is a 2 argument function. The first is the list and the second is the ordering function. With the list ll defined in the last example, one gets

```
EXTREMUM(ll,ordp); ==> 1
```

GCDNL takes a list of integers as argument and returns their gcd.

iii. There are four functions to identify dependencies. FUNCVAR takes any expression as argument and returns the set of variables on which it depends. Constants are eliminated.

```
FUNCVAR(e+pi+sin(log(y)); ==> {y}
```

DEPATOM has an **atom** as argument. It returns it if it is a number or if no dependency has previously been declared. Otherwise, it returns the list of variables which the prevoius DEPEND declarations imply.

```
depend a,x,y;

DEPATOM a; ==> {x,y}
```

The functions EXPLICIT and IMPLICIT make explicit or implicit the dependencies. This example shows how they work:

```
depend a,x; depend x,y,z;

EXPLICIT a; ==> a(x(y,z))

IMPLICIT ws; ==> a
```

These are useful when one wants to trace the names of the independent variables and (or) the nature of the dependencies.

KORDERLIST is a zero argument function which displays the actual ordering.

```
korder x,y,z;

KORDERLIST; ==> (x,y,z)
```

iv. A command REMNONCOM to remove the non-commutativity of operators previously declared non-commutative is available. Its use is like the one of the command NONCOM.

v. Filtering functions for lists.

CHECKPROPLIST is a boolean function which checks if the elements of a list have a definite property. Its first argument is the list, its second argument is a boolean function (FIXP NUMBERP ...) or an ordering function (as ORDP).

EXTRACTLIST extracts from the list given as its first argument the elements which satisfy the boolean function given as its second argument. For example:

```
if CHECKPROPLIST({1,2},fixp) then "ok"; ==> ok

l:={1,a,b,"st")$

EXTRACTLIST(l,fixp); ==> {1}

EXTRACTLIST(l,stringp); ==> {st}
```

vi. Coercion.

Since lists and arrays have quite distinct behaviour and storage properties, it is interesting to coerce lists into arrays and vice-versa in order to fully exploit the advantages of both datatypes. The functions ARRAY_TO_LIST and LIST_TO_ARRAY are provided to do that easily. The first function has the array identifier as its unique argument. The second function has three arguments. The first is the list, the second is the dimension of the array and the third is the identifier which defines it. If the chosen dimension is not compatible with the the list depth, an error message is issued. As an illustration suppose that ar is an array whose components are 1,2,3,4. then

```
ARRAY_TO_LIST ar; ==> {1,2,3,4}

LIST_TO_ARRAY({1,2,3,4},1,arr}; ==>
```

generates the array arr with the components 1,2,3,4.

vii. Control of the HEPHYS package.

The commands REMVECTOR and REMINDEX remove the property of being a 4-vector or a 4-index respectively.

The function MKGAM allows to assign to any identifier the property of a Dirac gamma matrix and, eventually, to suppress it. Its interest lies in the fact that, during a calculation, it is often useful to transform a gamma matrix into an abstract operator and vice-versa. Moreover, in many applications in basic physics, it is interesting to use the identifier g for other purposes. It takes two arguments. The first is the identifier. The second must be chosen equal to **t** if one wants to transform it into a gamma matrix. Any other binding for this second argument suppresses the property of being a gamma matrix the identifier is supposed to have.

16.5.8 Properties and Flags

In spite of the fact that many facets of the handling of property lists is easily accessible in algebraic mode, it is useful to provide analogous functions *genuine* to the algebraic mode. The reason is that, altering property lists of objects, may easily destroy the integrity of the system. The functions, which are here described, **do ignore** the property list and flags already defined by the system itself. They generate and track the *addtional properties and flags* that the user issues using them. They offer him the possibility to work on property lists so that he can design a programming style of the "conceptual" type.

i. We first consider "flags".
To a given identifier, one may associate another one linked to it "in the background". The three functions PUTFLAG, DISPLAYFLAG and CLEARFLAG handle them.

PUTFLAG has 3 arguments. The first one is the identifier or a list of identifiers, the second one is the name of the flag, and the third one is T (true) or 0 (zero). When the third argument is T, it creates the flag, when it is 0 it destroys it. In this last case, the function does return nil (not seen inside the algebraic mode).

```
PUTFLAG(z1,flag_name,t); ==> flag_name

PUTFLAG({z1,z2},flag1_name,t); ==> t

PUTFLAG(z2,flag1_name,0) ==>
```

DISPLAYFLAG allows one to extract flags. The previous actions give:

```
DISPLAYFLAG z1; ==>{flag_name,flag1_name}

DISPLAYFLAG z2 ; ==> {}
```

CLEARFLAG is a command which clears *all* flags associated with the identifiers $id_1, \ldots, id_n$.

ii. Properties are handled by similar functions. PUTPROP has four arguments. The second argument is, here, the *indicator* of the property. The third argument may be *any valid expression*. The fourth one is also T or 0.

```
PUTPROP(z1,property,x^2,t); ==> z1
```

In general, one enters

```
PUTPROP(LIST(idp1,idp2,..),<propname>,<value>,T);
```

To display a specific property, one uses DISPLAYPROP which takes two arguments. The first is the name of the identifier, the second is the indicator of the property.

```
                                                  2
DISPLAYPROP(z1,property); ==> {property,x   }
```

Finally, CLEARPROP is a nary commmand which clears *all* properties of the identifiers which appear as arguments.

16.5.9 Control Functions

Here we describe additional functions which improve user control on the environment.

i. The first set of functions is composed of unary and binary boolean functions. They are:

```
ALATOMP x;     x is anything.
ALKERNP x;     x is anything.
DEPVARP(x,v); x is anything.

   (v is an atom or a kernel)
```

ALATOMP has the value T iff x is an integer or an identifier *after* it has been evaluated down to the bottom.

ALKERNP has the value T iff x is a kernel *after* it has been evaluated down to the bottom.

DEPVARP returns T iff the expression x depends on v at **any level**.

The above functions together with PRECP have been declared operator functions to ease the verification of their value.

NORDP is equal to NOT ORDP.

ii. The next functions allow one to *analyze* and to *clean* the environment of REDUCE created by the user while working **interactively**. Two functions are provided:
SHOW allows the user to get the various identifiers already assigned and to see their type. SUPPRESS selectively clears the used identifiers or clears them all. It is to be stressed that identifiers assigned from the input of files are **ignored**. Both functions have one argument and the same options for this argument:

```
SHOW (SUPPRESS) all
SHOW (SUPPRESS) scalars
SHOW (SUPPRESS) lists
SHOW (SUPPRESS) saveids    (for saved expressions)
SHOW (SUPPRESS) matrices
SHOW (SUPPRESS) arrays
SHOW (SUPPRESS) vectors
             (contains vector, index and tvector)
SHOW (SUPPRESS) forms
```

The option all is the most convenient for SHOW but, with it, it may takes some time to get the answer after one has worked several hours. When entering REDUCE the option all for SHOW gives:

```
SHOW all; ==>

        scalars are: NIL
        arrays are: NIL
        lists are: NIL
        matrices are: NIL
        vectors are: NIL
        forms are: NIL
```

It is a convenient way to remind the various options. Here is an example which is valid when one starts from a fresh environment:

```
a:=b:=1$

SHOW scalars; ==>  scalars are: (A B)

SUPPRESS scalars; ==> t

SHOW scalars; ==>  scalars are: NIL
```

iii. The CLEAR function of the system does not do a complete cleaning of OPERATORS and FUNCTIONS . The following two functions do a more

complete cleaning and, also, automatically takes into account the *user* flag and properties that the functions PUTFLAG and PUTPROP may have introduced.

Their names are CLEAROP and CLEARFUNCTIONS. CLEAROP takes one operator as its argument.
CLEARFUNCTIONS is a nary command. If one issues

```
CLEARFUNCTIONS a1,a2, ... , an $
```

The functions with names a1,a2, ... ,an are cleared. One should be careful when using this facility since the only functions which cannot be erased are those which are protected with the lose flag.

16.5.10 Handling of Polynomials

The module contains some utility functions to handle standard quotients and several new facilities to manipulate polynomials.

i. Two functions `ALG_TO_SYMB` and `SYMB_TO_ALG` allow one to change an expression which is in the algebraic standard quotient form into a prefix lisp form and vice-versa. This is done in such a way that the symbol `list` which appears in the algebraic mode disappears in the symbolic form (there it becomes a parenthesis "()") and it is reintroduced in the translation from a symbolic prefix lisp expression to an algebraic one. Here, is an example, showing how the wellknown lisp function `FLATTENS` can be trivially transposed inside the algebraic mode:

```
algebraic procedure ecrase x;
lisp symb_to_alg flattens1 alg_to_symb algebraic x;

 symbolic procedure flattens1 x;
 % ll; ==> ((A B) ((C D) E))
 % flattens1 ll; (A B C D E)
   if atom x then list x else
   if cdr x then
       append(flattens1 car x, flattens1 cdr x)
     else flattens1 car x;
```

gives, for instance,

```
ll:={a,{b,{c},d,e},{{{z}}}}$


ECRASE ll; ==> {A, B, C, D, E, Z}
```

The function `MKDEPTH_ONE` described above implements that functionality.

ii. `LEADTERM` and `REDEXPR` are the algebraic equivalent of the symbolic functions `LT` and `RED`. They give, respectively, the *leading term* and the *reductum* of a polynomial. They also work for rational functions. Their interest lies in the fact that they do not require one to extract the main variable. They work according to the current ordering of the system:

```
pol:=x++y+z$

LEADTERM pol; ==> x

korder y,x,z;

LEADTERM pol; ==> y

REDEXPR pol; ==> x + z
```

By default, the representation of multivariate polynomials is recursive. It is justified since it is the one which takes the least memory. With such a representation, the function LEADTERM does not necessarily extract a true monom. It extracts a monom in the leading indeterminate multiplied by a polynomial in the other indeterminates. However, very often, one needs to handle true monoms separately. In that case, one needs a polynomial in *distributive* form. Such a form is provided by the package GROEBNER (H. Melenk et al.). The facility there is, however, much too involved in many applications and the necessity to load the package makes it interesting to construct an elementary facility to handle the distributive representation of polynomials. A new switch has been created for that purpose. It is called DISTRIBUTE and a new function DISTRIBUTE puts a polynomial in distributive form. With that switch set to **on**, LEADTERM gives **true** monoms.

MONOM transforms a polynomial into a list of monoms. It works *whatever the position of the switch* DISTRIBUTE.

SPLITTERMS is analoguous to MONOM except that it gives a list of two lists. The first sublist contains the positive terms while the second sublist contains the negative terms.

SPLITPLUSMINUS gives a list whose first element is the positive part of the polynomial and its second element is its negative part.

iii. Two complementary functions LOWESTDEG and DIVPOL are provided. The first takes a polynomial as its first argument and the name of an indeterminate as its second argument. It returns the *lowest degree* in that indeterminate. The second function takes two polynomials and returns both the quotient and its remainder.

16.5.11 Handling of Transcendental Functions

The functions TRIGREDUCE and TRIGEXPAND and the equivalent ones for hyperbolic functions HYPREDUCE and HYPEXPAND make the transformations to

multiple arguments and from multiple arguments to elementary arguments. Here is a simple example:

```
aa:=sin(x+y)$

TRIGEXPAND aa; ==> SIN(X)*COS(Y) + SIN(Y)*COS(X)

TRIGREDUCE ws; ==> SIN(Y + X)
```

When a trigonometric or hyperbolic expression is symmetric with respect to the interchange of `SIN (SINH)` and `COS (COSH)`, the application of `TRIG(HYP)-REDUCE` may often lead to great simplifications. However, if it is highly assymetric, the repeated application of `TRIG(HYP)-REDUCE` followed by the use of `TRIG(HYP)-EXPAND` will lead to *more* complicated but more symmetric expressions:

```
aa:=(sin(x)^2+cos(x)^2)^3$

TRIGREDUCE aa; ==> 1



bb:=1+sin(x)^3$



TRIGREDUCE bb; ==>

        - SIN(3*X) + 3*SIN(X) + 4
       ---------------------------
                    4

 TRIGEXPAND ws; ==>


       3                    2
  SIN(X)  - 3*SIN(X)*COS(X)  + 3*SIN(X) + 4
  ----------------------------------------
                     4
```

16.5.12 Handling of n–dimensional Vectors

Explicit vectors in EUCLIDEAN space may be represented by list-like or bag-like objects of depth 1. The components may be bags but may **not** be lists. Functions are provided to do the sum, the difference and the scalar product. When the space-dimension is three there are also functions for the cross and mixed products. SUMVECT, MINVECT, SCALVECT, CROSSVECT have two arguments. MPVECT has three arguments. The following example is sufficient to explain how they work:

```
l:={1,2,3}$

ll:=list(a,b,c)$

SUMVECT(l,ll); ==> {A + 1,B + 2,C + 3}

MINVECT(l,ll); ==> { - A + 1, - B + 2, - C + 3}

SCALVECT(l,ll); ==> A + 2*B + 3*C

CROSSVECT(l,ll); ==> { - 3*B + 2*C,3*A - C, - 2*A + B}

MPVECT(l,ll,l); ==> 0
```

16.5.13 Handling of Grassmann Operators

Grassman variables are often used in physics. For them the multiplication operation is associative, distributive but anticommutative. The KERNEL of REDUCE does not provide it. However, implementing it in full generality would almost certainly decrease the overall efficiency of the system. This small module together with the declaration of antisymmetry for operators is enough to deal with most calculations. The reason is, that a product of similar anticommuting kernels can easily be transformed into an antisymmetric operator with as many indices as the number of these kernels. Moreover, one may also issue pattern matching rules to implement the anticommutativity of the product. The functions in this module represent the minimum functionality required to identify them and to handle their specific features.

PUTGRASS is a (nary) command which give identifiers the property of being the names of Grassmann kernels. REMGRASS removes this property.

GRASSP is a boolean function which detects grassmann kernels.

GRASSPARITY takes a **monom** as argument and gives its parity. If the monom is a simple grassmann kernel it returns 1.

GHOSTFACTOR has two arguments. Each one is a monom. It is equal to

```
(-1)**(GRASSPARITY u * GRASSPARITY v)
```

Here is an illustration to show how the above functions work:

```
PUTGRASS eta; ==> t

if GRASSP eta(1) then "grassmann kernel"; ==>

                grassmann kernel

aa:=eta(1)*eta(2)-eta(2)*eta(1); ==>

        AA :=  - ETA(2)*ETA(1) + ETA(1)*ETA(2)

GRASSPARITY eta(1); ==> 1

GRASSPARITY (eta(1)*eta(2)); ==> 0

GHOSTFACTOR(eta(1),eta(2)); ==> -1

grasskernel:=
  {eta(~x)*eta(~y) => -eta y * eta x when nordp(x,y),
  (~x)*(~x) => 0 when grassp x};

exp:=eta(1)^2$

exp where grasskernel; ==> 0

aa where grasskernel; ==>  - 2*ETA(2)*ETA(1)
```

16.5.14 Handling of Matrices

This module provides functions for handling matrices more comfortably.

i. Often, one needs to construct some UNIT matrix of a given dimension. This

construction is done by the system thanks to the function UNITMAT. It is a nary function. The command is

```
UNITMAT M1(n1), M2(n2), .....Mi(ni) ;
```

where M1,...Mi are names of matrices and n1, n2, ..., ni are integers .

MKIDM is a generalization of MKID. It allows one to connect two or several matrices. If u and u1 are two matrices, one can go from one to the other:

```
matrix u(2,2);$  unitmat u1(2)$

u1; ==>

         [1  0]
         [    ]
         [0  1]


mkidm(u,1); ==>

         [1  0]
         [    ]
         [0  1]
```

This function allows one to make loops on matrices like in the following illustration. If U, U1, U2,.., U5 are matrices:

```
FOR I:=1:5 DO U:=U-MKIDM(U,I);
```

can be issued.

ii. The next functions map matrices on bag-like or list-like objects and conversely they generate matrices from bags or lists.

COERCEMAT transforms the matrix U into a list of lists. The entry is

```
COERCEMAT(U,id)
```

where id is equal to list othewise it transforms it into a bag of bags whose envelope is equal to id.

BAGLMAT does the opposite job. The **first** argument is the bag-like or list-like object while the second argument is the matrix identifier. The entry is

```
BAGLMAT(bgl,U)
```

bgl becomes the matrix U . The transformation is **not** done if U is *already* the name of a previously defined matrix. This is to avoid ACCIDENTAL redefinition of that matrix.

ii. The functions SUBMAT, MATEXTR, MATEXTC take parts of a given matrix.

SUBMAT has three arguments. The entry is

```
SUBMAT(U,nr,nc)
```

The first is the matrix name, and the other two are the row and column numbers. It gives the submatrix obtained from U by deleting the row nr and the column nc. When one of them is equal to zero only column nc or row nr is deleted.

MATEXTR and MATEXTC extract a row or a column and place it into a list-like or bag-like object. The entries are

```
MATEXTR(U,VN,nr)

MATEXTC(U,VN,nc)
```

where U is the matrix, VN is the "vector name", nr and nc are integers. If VN is equal to list the vector is given as a list otherwise it is given as a bag.

iii. Functions which manipulate matrices. They are MATSUBR, MATSUBC, HCONCMAT, VCONCMAT, TPMAT, HERMAT

MATSUBR MATSUBC substitute rows and columns. They have three arguments. Entries are:

```
MATSUBR(U,bgl,nr)

MATSUBC(U,bgl,nc)
```

The meaning of the variables `U, nr, nc` is the same as above while `bgl` is a list-like or bag-like vector. Its length should be compatible with the dimensions of the matrix.

`HCONCMAT VCONCMAT` concatenate two matrices. The entries are

```
HCONCMAT(U,V)

VCONCMAT(U,V)
```

The first function concatenates horizontally, the second one concatenates vertically. The dimensions must match.

`TPMAT` makes the tensor product of two matrices. It is also an *infix* function. The entry is

```
TPMAT(U,V) or U TPMAT V
```

`HERMAT` takes the hermitian conjuguate of a matrix The entry is

```
HERMAT(U,HU)
```

where `HU` is the identifier for the hermitian matrix of `U`. It should be **unassigned** for this function to work successfully. This is done on purpose to prevent accidental redefinition of an already used identifier .

iv. `SETELMAT GETELMAT` are functions of two integers. The first one resets the element `(i,j)` while the second one extracts an element identified by `(i,j)`. They may be useful when dealing with matrices *inside procedures*.