# Implementation of multilinear operators in REDUCE and applications in mathematics

Marcel Roelofs Peter K.H. Gragert

University of Twente, Department of Applied Mathematics
P.O. Box 217, 7500 AE Enschede, The Netherlands
E-mail: roelofs@math.utwente.nl

## ABSTRACT

In this paper we introduce and implement a concept for dealing with mathematical bases of linear spaces and mappings (multi)linear with respect to such bases, in REDUCE (cf. [1]). Using this concept we give some examples how to implement some well known (multi)linear mappings in mathematics with very little effort. Moreover we implement a procedure *operatorcoeff* similar to the standard REDUCE procedure *coeff*, but now for linear spaces instead of polynomial rings.

## 1. INTRODUCTION

A concept of utmost importance in mathematics is the notion of a linear space, i.e., a space which is supplied with a basis, such that any of its elements can be uniquely written as a linear combination of basis elements. Moreover, linear spaces admit (multi)linear mappings which are completely determined by their action on basis elements. More specifically, if $\{e_1, \ldots, e_n\}$ is a basis of a linear space $E$, $x_1, \ldots, x_m \in E$ are given by

$$x_j = \sum_{i=0}^{n} x_j^i e_i \qquad (j = 1, \ldots, m)$$

and $P : E^m \to F$ is a (multi)linear mapping from $E$ to some linear space $F$ then

$$P(x_1, \ldots, x_m) = \sum_{i_1=1}^{n} \cdots \sum_{i_m=1}^{n} x_1^{i_1} \cdots x_m^{i_m} P(e_{i_1}, \ldots, e_{i_m}).$$

In spite of its importance REDUCE has no extensive facilities for linear spaces or (multi)linear mappings. In fact, the only possibility is to declare an operator to be *linear*. *Linear* operators, however, can only be linear in one argument, and, moreover, must act on very specific linear spaces, namely polynomial rings in one variable. Also other computer algebra system such as Maple or Mathematica only have facilities similar to the *linear* statement in REDUCE and lack facilities for multilinear operators.

In this paper we will introduce a concept for the representation of more general linear spaces in REDUCE, set up an environment for general *multilinear* mappings on such linear spaces, and, finally, implement a procedure *operatorcoeff* which acts similar to the standard REDUCE procedure *coeff* but now for linear spaces in our context instead of just polynomial rings.



The paper is organised as follows. In section 2 we will explain the main ideas behind the concept, after which we will implement the main procedure *split_f* which finds all basis elements in a standard form together with their coefficients. In section 3 we will implement *multilinear* operators and the procedure *operatorcoeff* using *split_f*. In section 4 we will give some examples how *multilinear* operators can be used to implement some well known (multi)linear mappings in mathematics.

Finally appendix A contains the source of some essential procedures, whereas appendix B contains the source of one of the examples. Both appendices were taken directly from the RWEB sources describing the programs, i.e., files containing a mixture of documentation and pieces of program. Such a file can either be turned into a TeX file with all pieces of program formatted nicely, or into a source file by removing the documentation and putting all pieces of program into the correct order. WEB was originally developed by Knuth to document his TeX program (cf. [2]), but later adapted by Ramsey to suit any Algol like language (cf. [3], anonymous ftp: `princeton.edu: ftp/pub/spiderweb.tar.Z`). Using Ramsey's system we produced a version of WEB suitable for REDUCE and rlisp sources, called RWEB, and which is available by anonymous ftp at `utmfu0.math.utwente.nl: ftp/pub/RWEB`.

## 2. LINEAR SPACES AND ANALYSIS OF STANDARD FORMS

We will represent linear spaces in REDUCE by giving its basis as a set of elements of some algebraic operator. For instance, the basis $\{e_1, \ldots, e_n\}$ from the previous section could be represented by a set of operator elements $E(1), \ldots, E(n)$. It may even be convenient to use more operators to give a basis. We will see an example of this in section 4.

Using this representation we can represent all kinds of linear spaces in a more or less natural way, whereas still all basis elements can be recognized at once, namely as operator elements of a certain operator. A basis element $x_1^{i_1} \cdots x_n^{i_n}$ of a polynamial ring in $n$ variables $x_1, \ldots, x_n$, for instance, could be represented by an operator element $P(i_1, \ldots, i_n)$ in this way.

In fact, mostly we are not only interested in pure linear spaces, but also in the part of an expression independent of any basis element, which we will from now on denote by the "independent part" of the expression. We can look at it as an extension of the original basis with an additional basis element, namely 1.

Using this convention for basis elements the main problem for the implementation of *multilinear* operators and the procedure *op-*

| | | |
|---|---|---|
| <standard quotient> | ::= | (<numerator> . <denominator>) |
| <numerator> | ::= | <standard form> |
| <denominator> | ::= | <standard form> |
| <standard form> | ::= | **nil** \| <domain element> \| (<leading term> . <reductum>) |
| <leading term> | ::= | (<leading power> . <leading coefficient>) |
| <reductum> | ::= | <standard form> |
| <leading power> | ::= | (<main variable> . <leading degree>) |
| <leading coefficient> | ::= | <standard form> |
| <main variable> | ::= | <identifier> \| <operator element> |
| <leading degree> | ::= | <integer> |
| <operator element> | ::= | (<operator name>) \| (<operator name> <arguments>) |
| <operator name> | ::= | <identifier> |

**Table 1:** Syntax of standard quotients.

*eratorcoeff* is how to find all basis elements and their coefficients in an algebraic expression, as well as the independent part. For the solution to this problem we recall that algebraic expressions in REDUCE are stored as standard quotients which essentially satisfy the syntax of table 1, which is valid if the switch *exp* is **on** (which is the initial setting) and where we have left out the parts that are of no particular interest to us. If the switch *exp* is **off**, a main variable may also be some other algebraic expression in prefix notation. It can be easily checked that this setting will not be useful in our case and even may lead to invalid results using the algorithm described below. Therefore we require *exp* to be **on**.

Since we can divide all coefficients of basis elements and the independent part by the denominator of the algebraic expression at the end of the evaluation, it is clear that for our purposes we only need to analyse numerators of algebraic expressions, which are standard forms. Due to the recursive definition of a standard form, we can readily deduce an algorithm to find all of its basis elements and their coefficients as well as the independent part.

For this suppose that we have to analyse a standard form $F$ which is a sum $\sum_i T_i$ of standard terms $T_i$, whereas each $T_i$ is the product of a leading power $P_i$ and a leading coefficient $C_i$ and has $V_i$ as its main variable. Further suppose that we want to deliver a list $L$ containing the independent part of $F$ together with all basis elements and their coefficients, where *oplist* is the list of operators, elements of which are allowed as basis elements. It may be clear that this can be done by checking all terms $T_i$ separately. Our algorithm for analysing a term $T_i$ essentially consists of distinguishing the following 4 cases:

1. $T_i$ is **nil**. This term will not contribute, so we have to take no action.
2. $T_i$ is a domain element, in particular is not an operator element of one of the operators on *oplist*, hence it must be added to the independent part of $F$. We can check if $T_i$ is a domain element by using the REDUCE procedure *domainp*.
3. $V_i$ is an operator element of one of the operators on *oplist*, hence if $V_i$ occurs linearly in $T_i$, i.e., the leading degree is 1 and $C_i$ does not contain other operator elements of operators on *oplist*, we have to update $L$, i.e., if $V_i$ already occurs in $L$ as a basis element we have to add $C_i$ to its coefficient, otherwise we have to add $V_i$ as a basis element to $L$ with coefficient $C_i$. If $V_i$ does not occur linearly we can stop with an error message.
4. $V_i$ is not an operator element of one of the operators on *oplist* (i.e., it is an identifier or an operator element of some other operator). In this case we can recursively examine $C_i$ for the occurence of basis elements, if we keep in mind that the coefficients of basis elements found there have to be multiplied with the additional factor $P_i$.

The actions described above are implemented in the (recursive) procedure *split_f(form,oplist,fact,kc_list)*, where *form* is the standard form to be analysed. The third argument *fact* is a factor (as in case 4) with which the coefficient of some basis element found has to be multiplied. It is clear that is has to be initialized to 1 at top level. The fourth argument *kc_list* is a dotted pair, the *car* of which is the independent part, the *cdr* the a list of basis elements with coefficients, being build up so far. Hence *kc_list* has to be initialized to **nil . nil**. The RWEB source of *split_f* can be found in appendix A.

## 3. IMPLEMENTATION OF MULTILINEAR OPERATORS AND THE PROCEDURE OPERATORCOEFF

Algebraic expressions offered to REDUCE have to be evaluated. This evaluation is mainly taken care of by the procedure *simp*, which simplifies a given expression to a canonical standard quotient (cf. [1,4]). One of the ways in which one can influence the process of simplification is by specifying a procedure that has to take care of the simplication of some specific algebraic operator.

By using this fact and using the procedure *split_f* of the previous section we can easily outline the actions necessary for the simplification of *multilinear* operators: find the basis elements together with their coefficients of all arguments of the operator and return the sum of all possible combinations of basis elements. More specifically we will introduce the concept of multilinearity in REDUCE by implementing a simplification procedure *simp_multilinear* for *multilinear* operators. So, an algebraic operator $P$ will be *multilinear* if its *simpfn* is *simp_multilinear*.

Due to the nature of simplification procedures *simp_multilinear* must return a standard quotient and, moreover, this standard quotient will not be simplified again, so all monomial operator elements of $P$ formed during the process have to be simplified before adding them to the standard quotient. Simply applying the standard REDUCE simplification procedure *simp* to these monomial operator elements would be unwise, since $P$ is *multilinear*, i.e., $P$'s *simpfn* is *simp_multilinear*, and in this way we will get in an infinite loop. For ordinary cases applying the procedure *simpiden*, which checks if an operator element has a value and if so, returns this value as a standard quotient, otherwise the operator element itself, will suffice. But since we intend to use *multilinear* operators in special packages, where monomial operator elements may have to be simplified in a special way, we will add to each *multilinear* operator the property *resimpfn* which is the procedure to be used for the simplification of monomial operator elements.

With this information we can outline the two steps which together constitute the essential parts of *simp_multilinear*.

If we get a multilinear expression to be simplified by the simplification function *simp_multilinear* we must in the first place analyse all arguments using *split_f*, in order to split them into a list of basis elements with their coefficients. At the same time we have to keep record of the product of the denominators of all arguments, since the analysis of the arguments is performed on standard forms, and eventually the entire result has to be divided by this product of denominators. These actions are implemented in the (recursive) procedure *split_arguments(arg_list,oplist,splitted_list)*, where *arg_list* is the list of arguments of the multilinear expression to be analysed and *oplist* the list of operators, elements of which are allowed as basis elements. The third argument and final result *splitted_list* is a dotted pair, the *car* of which is the product of denominators of all arguments treated so far and the *cdr* is the list of splitted arguments in reverse order.

If we define a component of an argument as the independent part or a basis element together with its coefficient, an argument splitted by *split_f* may be looked at as a list of components and the result of the procedure *split_arguments* essentially as a stack of component lists completed with the product of denominators. From this stack we are to build a sum of monomial operator elements. We will do this with help of the (mutually recursive) procedures *process_arg_stack* and *process_comp_list*, which will described right away. However, before this, we will describe their arguments. In all procedures the argument *arg_stack* will be a stack of component lists, the *car* of which has to be treated next, *comp_list* the current component list to be treated, *op_name* the name of the *multilinear* operator and *arg_list* the list of arguments being build up for the current monomial operator element which has to be multiplied with a factor *fact*.

The procedure *process_arg_stack(arg_stack,op_name,arg_list, fact)* takes the following actions:

1. If *arg_stack* is empty this means that the argument list *arg_list* we are building is complete, i.e., does not contain any more arguments, so we can apply the *resimpfn* to the monomial operator element obtained in this way and return its value multiplied with the factor *fact*.
2. If the argument stack is not empty, we can extend the argument list with all components of the top of the stack and sum the results obtained by applying *process_arg_stack* to the rest of the stack. This is done by calling the procedure *process_comp_list*.

The procedure *process_comp_list* returns the sum of applying *process_independent_part* and *process_components* to the current component list.

Following our definition of multilinearity the procedure *process_independent_part* will multiply *fact* with the independent part and add 1 to the argument list. If, however, there is no independent part, the argument list obtained by adding the independent part would contain a zero, so due to the multilinearity such a term would not contribute, in which case the procedure can return **nil** . 1.

The procedure *process_components* returns the sum of applying *process_arg_stack* to the remaining *arg_stack* with *arg_list* extended with the basis element of a component and *fact* multiplied with its coefficient for each component.

The RWEB source of all procedures described above can be found in appendix A.

With the procedures explained above, the simplification procedure *simp_multilinear* can be described at once:

1. Apply the procedure *split_arguments* to the list of arguments of the multilinear operator to get a stack of component lists, as well as the product of the denominators of the arguments.
2. Apply *process_arg_stack* to the stack obtained in 1. to get the sum of simplified monomial operator elements.
3. Return the result of 2. divided by the product of denominators.

As a last step we have to implement a *multilinear* statement satisfying the syntax

*multilinear* $P$(operator | list of operators [,resimp. proc.]);

as to declare $P$ a multilinear operator, where the first argument is the operator or list of operators allowed as basis elements and the optional second argument is the name of the procedure used for the simplification of the resulting monomial operator elements. If the second argument is missing we will take *simpiden* as the default resimplification procedure.

The actions necessary to declare $P$ *multilinear* are the following:

1. assign the property *simpfn* to $P$ with value *simp_multilinear*.
2. assign the property *resimpfn* to $P$ with value *simpiden* if there is no resimplification procedure specified, otherwise the specified one.
3. flag $P$ as *full*. An operator that is flagged *full* will have its operatorname also passed to its simplification procedure, otherwise only the arguments of the operator are passed.
4. assign the property *oplist* to $P$ with value the list of operators allowed as basis elements.

A second and much simpler application of the procedure *split_f* is the procedure *operatorcoeff*, which is the counterpart for linear bases (in our concept) of the standard REDUCE procedure *coeff*. Due to the possibly more dimensional or sparse nature of the arguments of the operator elements, the result of applying *operatorcoeff* to some algebraic expression cannot, contrary to *coeff*, be only an algebraic list containing coefficients of basis elements, but should also give information about the basis element to which each coefficient belongs.

Therefore, if $X$ is an algebraic expression linear w.r.t. operator elements of some specified operators $P_1, \ldots, P_n$, applying *operatorcoeff* to $X$ w.r.t. the operators $P_1, \ldots, P_n$ will return an algebraic list containing:

1. as the first element of the list the part of $X$ not linearly depending on operator elements of one of the operators $P_1, \ldots, P_n$.
2. followed by zero or more algebraic lists containing a basis element with its coefficient.

The implementation of *operator_coeff* with help of *split_f* is essentially a matter of dividing all occuring coefficients by the denominator of the expression being analysed and replacing ordinary lists by algebraic lists.

As the implementation of the *multilinear* statement and the procedure *operatorcoeff* are rather straightforward we will not give these explicitly.

## 4. EXAMPLES OF APPLICATION IN MATHEMATICS

As a first example we shall introduce the ordinary tensor product as a *multilinear* operator in REDUCE. For this suppose that we use the operator $X$ to represent the basis elements of some linear space $E$ and let @ denote the tensor product. Then the declarations

*multilinear* @($X$);
*infix* @;
*precendence* @,*idifference*;

will turn @ into a multilinear infix operator, i.e., a tensor product w.r.t. $E$. The last statement is meant to take care that @ takes precedence over the ordinary multiplication, so that expressions like

$X(1) * 3@X(2)$ will be simplified to $3 * (X(1)@X(2))$ instead of $3 * X(1) * (1@X(2))$, which is also possible according to our definition of multilinearity.

From this point of view its is also very easy to define symmetric and alternating tensor products. First of all this can be done by specifying a resimplification procedure in the *multilinear* declaration that takes care of the symmetrization or antisymmetrization, respectively, but in this special case there is second, much simpler, solution, namely by declaring @ to be *symmetric* or *antisymmetric*. In doing so, the standard resimplification procedure *simpiden* will take care of the necessary actions. So we see that only four simple statements suffice to define a symmetric tensor algebra or an exterior algebra in REDUCE.

Our second example is the implementation of the two most important building stones for Hirota's bilinear formalism, which is used frequently in mathematical physics, in REDUCE (cf. [5]). These are tensor products, defined above, and operators like $D_x$ defined by

$$D_x(f \otimes g) = f_x \otimes g - f \otimes g_x$$

where $f$ and $g$ are functions depending on $x$, subscripts denoting differentation. If we use the operator *F* to represent functions and the operator *DX* to represent $D_x$, the most simple scheme for implementing Hirota's bilinear formalism is the set of statements

*multilinear @({F,DF}); infix @; precendence @,idifference;*
*multilinear DX(@,simpdx);*

where the procedure *simpdx* is defined by

```
lisp procedure simpdx exprn;
begin scalar arg1,arg2;
  arg1:=caddr exprn; arg2:=cadddr exprn;
  return subtrsq(simpiden list('!@,list('df,arg1,'x),arg2),
                 simpiden list('!@,arg1,list('df,arg2,'x)));
end;
```

In this way *DX(DX(F(1)@F(2)))* will be correctly simplified to *DF(F(1),X,2)@F(2)* $-$ 2∗*DF(F(1),X)@DF(F(2),X)* + *F(1)@ DF(F(2),X,2)*. It should however be noted that more sophisticated use of Hirota's formalism may require a somewhat more complicated setup.

A third example is the implementation of heighest weight modules of the Virasoro algebra, an important object in physics, in REDUCE (see e.g. [6]). This example is completely worked out in appendix B.

Further examples include the implementation of a package for computations in free Lie (super)algebras and a package for the computation of cohomology of Lie (super)algebras, both of which have been written at our site and make essential use of the concept of *multilinear* operators (cf. [7]).

## 5. CONCLUSIONS

At the cost of representing various kinds of expressions as operator elements of some operator, the procedures described in this paper offer a fast and flexible way to implement multilinear operators on these expressions and decompose algebraic expressions into various components.

Moreover simplification of multilinear operators is reasonably efficient. For instance, the simplification of linear operators, implemented with help of multilinear operators, will be executed 2 to 3 times as fast as the simplification of similar linear operators, as implemented in REDUCE.

The procedures described in this paper, are part of a package (which we call the TOOLS package) containing all kinds of procedures facilitating the use of algebraic operators in REDUCE. This package is available by E-mail at the address given on the first page.

Finally, the method of analysing standard forms applied in the procedure *split_f* can easily be adapted to generalize *coeff* for the decomposition of algebraic expressions in polynomial rings of more than one variable (actually such a procedure is also part of the TOOLS package).

## ACKNOWLEDGEMENTS

The first author would like to thank Gerhard Post for the numerous fruitful discussions on the use of REDUCE in mathematics, which brought about many of the useful ideas.

## REFERENCES

[1] A.C. Hearn, "REDUCE 2: A system and Language for Algebraic Manipulation", *Proceedings SYMSAM 2* (S.R. Petrick, ed.), 128–133. New York: ACM (1971).

[2] D.E. Knuth, "Literate Programming", *The Computer Journal* **27:2** (1984), 97–111.

[3] N. Ramsey, "Literate Programming: Weaving a Language-Independent WEB", *Comm. of the ACM* **32** (1989), 1051–1055.

[4] J. Moses, "Algebraic Simplification: A Guide for the Perplexed", *Proceedings SYMSAM 2* (S.R. Petrick, ed.), 282–304. New York: ACM (1971).

[5] R. Hirota, "Direct Methods in Soliton Theory", in *Solitons*, 157–176. Berlin: Springer (1980).

[6] V.G. Kac and A.K. Raina, "Bombay Lectures on Highest Weight Representations of Infinite Dimensional Lie Algebras". Singapore: World Scientific (1987).

[7] G.H.M. Roelofs, "The LIESUPER package for REDUCE", memorandum (to appear), Dept. of Appl. Math., University of Twente (1991).

## APPENDIX A: RWEB SOURCE OF SOME ESSENTIAL PROCEDURES

**1.** In this RWEB file we shall give the full implementation of some of the procedures described in section 2 and 3. First of all we will give the source of the main procedure *split_f*, which is underlying all other procedures. Following its description in section 2 we need not explain very much anymore. Recall that the factor *fact* is a standard form and that *kc_list* is the component list being build so far, i.e. is the result and has to be returned if the current standard form is analysed.

```
lisp procedure split_f(form, oplist, fact, kc_list);
  if null form then kc_list
  else
    if domainp form then
        addf(multf(fact, form), car kc_list) . cdr kc_list
    else ⟨ Examine mvar, lc and red for basis elements 2 ⟩$
```

**2.** The remaining part of *split_f* corresponds to points 3 and 4 of section 2.

```
⟨ Examine mvar, lc and red for basis elements 2 ⟩ ≡
if ¬atom mvar form ∧ member(car mvar form, oplist) then
  if ¬ldeg form = 1 ∨ get_first_kernel(lc form, oplist) then
        rederr "SPLIT_F: expression not linear"
  else split_f(red form, oplist, fact,
        update_kc_list(kc_list, mvar form,
        multf(fact, lc form)))
else split_f(red form, oplist, fact,
      split_f(lc form, oplist,
      multf(fact, !*p2f lpow form), kc_list))
```

This code is used in section 1.

**3.** For convenience we will write a surrounding procedure *split_form*, which can be called at top level and initializes the third and fourth argument of *split_f*.

```
lisp procedure split_form(form, oplist);
  split_f(form, oplist, 1, nil . nil)$
```

**4.** For updating the *kc_list* as efficient as possible we need an *assoc*-like procedure *list_assoc*. If applied to an association list $L$, this procedure returns the remainder of $L$, the *car* of which would be the result of *assoc* applied to $L$.

```
lisp procedure list_assoc(car_exprn, a_list);
  if null a_list then a_list
  else
    if caar a_list = car_exprn then a_list
    else list_assoc(car_exprn, cdr a_list)$
```

**5.** In order to update the *kc_list* we first have to find out if the kernel w.r.t. which we update the list, is already occuring on it. If so, we have to adjust its coefficient, otherwise we can *cons* the kernel and coefficient in front of the list. Adjusting a coefficient is performed by using the procedures *list_assoc* and *rplaca* in order to avoid rebuilding of the entire list. The reader should verify that *rplaca* cannot do any harm in this application, since it is replacing a list.

```
lisp procedure update_kc_list(kc_list, kernel, coefficient);
  (if rest_list then
    ≪ rplaca(rest_list, caar rest_list .
          addf(cdar rest_list, coefficient)); kc_list ≫
  else car kc_list . (kernel . coefficient) . cdr kc_list)
        where rest_list = list_assoc(kernel, cdr kc_list)$
```

**6.** Next we will give the essential procedures for the implementation of the procedure *simp_multilinear*: *split_arguments*, *process_arg_stack* and *process_comp_list*, the description of which can be found in section 3.

We start with the implementation of the procedure *split_arguments*. Notice that the arguments of *arg_list* need to be simplified, since *split_arguments* is called from within a simplification procedure.

```
lisp procedure split_arguments(arg_list, oplist, splitted_list);
  if null arg_list then splitted_list
  else split_arguments(cdr arg_list, oplist,
        multf(denr first_arg, car splitted_list) .
        split_form(numr first_arg, oplist) . cdr splitted_list)
        where first_arg = simp!* car arg_list$
```

**7.** For convenience we will write a surrounding procedure *split_operator*, in order to hide the last two arguments of *split_arguments*. Recall that the list of operators allowed basis elements is stored as the property *oplist* of the multilinear operator considered.

```
lisp procedure split_operator u;
  split_arguments(cdr u, get(car u, 'oplist), 1 . nil)$
```

**8.** The procedures *process_arg_stack* and *process_comp_list* are also thoroughly described in section 3, so there is no need to give too much explanation here neither. Notice, however, that the coefficients of any component and hence also *fact* are given as standard forms. As the result is a standard quotient we have to convert *fact* into a standard quotient using *!*f2q*.

```
lisp procedure process_arg_stack(arg_stack,
      op_name, arg_list, fact);
  if null arg_stack then multsq(!*f2q fact,
        apply1(get(op_name, 'resimp_fn), op_name . arg_list))
  else process_comp_list(car arg_stack,
        cdr arg_stack, op_name, arg_list, fact)$
```

**9.** Processing the component list consists of processing the independent part and all the components.

```
lisp procedure process_comp_list(comp_list, arg_stack,
      op_name, arg_list, fact);
  addsq(process_independent_part(car comp_list,
      arg_stack, op_name, arg_list, fact),
      process_components(cdr comp_list,
      arg_stack, op_name, arg_list, fact))$
```

**10.** Following our description of multilinearity, processing the independent part of an argument boils down to multiplying *fact* with it and adding the argument 1 to *arg_list*. If, however, the independent part is **nil**, we can return **nil** immediately.

```
lisp procedure process_independent_part(independent_part,
      arg_stack, op_name, arg_list, fact);
  if null independent_part then nil . 1
  else process_arg_stack(arg_stack, op_name,
        1 . arg_list, multf(fact, independent_part))$
```

**11.** The procedure *process_components* has to process the *comp_list* until there are no more components of the argument being processed.

```
lisp procedure process_components(comp_list, arg_stack,
        op_name, arg_list, fact);
   if null comp_list then nil . 1
   else addsq(process_components(cdr comp_list,
            arg_stack, op_name, arg_list, fact),
            process_arg_stack(arg_stack, op_name,
            caar comp_list . arg_list,
            multf(fact, cdar comp_list)))$
```

**12.** To hide the auxiliary arguments of *process_arg_stack* we will write a surrounding procedure *build_sum* for it. Recall that *arg_list* and *fact* have to be initialized to **nil** and 1, respectively.

```
lisp procedure build_sum(op_name, arg_stack);
   process_arg_stack(arg_stack, op_name, nil, 1)$
```

**13.** With the procedures written above, the simplification function *simp_multilinear* can be written at once. We recall that the result of *split_arguments* is a dotted pair, the *car* of which is the product of the denominators of all arguments, the *cdr* the list of splitted arguments, an argument stack. Moreover, notice that we are sure that the *car* of *u* is the name of the operator, since we flagged this operator *full*.

```
lisp procedure simp_multilinear u;
   quotsq(build_sum(car u, cdr splitted_list),
        !*f2q car splitted_list)
        where splitted_list = split_operator u$
```

## APPENDIX B: VERMA MODULES OF THE VIRASORO ALGEBRA

**1.** The Virasoro algebra $W$ is given by a basis $\{z\} \cup \{e_i \mid i \in \mathbb{Z}\}$ and relations

$$[z, e_i] = 0 \quad \text{and} \quad [e_i, e_j] = (j-i)e_{i+j} + \frac{1}{12}(j^3 - j)\delta_{i,-j}z. \quad (1)$$

A Verma module $V_{h,c}$ of $W$ is a heighest weight module of $W$ with heighest weight vector $v$ such that

$$e_0 \cdot v = hv, \qquad z \cdot v = cv, \qquad e_i \cdot v = 0 \quad (i < 0) \qquad (2)$$

and the action of $e_i e_j$ equals

$$e_i e_j = [e_i, e_j] + e_j e_i. \qquad (3)$$

One can proof that a basis of $V_{h,c}$ is given by

$$\{v_{i_1 \ldots i_k} = e_{i_1} \cdots e_{i_k} \cdot v \mid k \in \mathbb{N}, 0 < i_1 \leq \ldots \leq i_k\}.$$

The main problem of working with Verma modules is how to write an element $e_j \cdot v_{i_1 \ldots i_k}$ as a sum of basis elements using the rules described above, since the number of terms tends to explode rather quickly. In this file we will give an implementation of Verma modules of the Virasoro algebra in REDUCE.

**2.** For this purpose we represent the basis elements $v_{i_1 \ldots i_k}$ of $V_{h,c}$ as the algebraic operator elements $v(i_1, \ldots, i_k)$ (in this way the heighest weight vector $v$ will be represented by the operator element $v()$). The basis elements $e_i$ ($i \in \mathbb{Z}$) and $z$ of the Virasoro algebra are represented by the operator elements $x(i)$ and $c(0)$, respectively. Finally, the action $\cdot$ will be represented by an algebraic operator @. The reader can easily verify that the solution to the problem posed in the first section is the implementation of an appropriate simplification procedure for the action @.

The first thing to be noticed is the bilinearity of the action @ w.r.t. the operators $x$, $c$ and $v$. Therefore we will declare the operator @ to be multilinear using the *multilinear* statement of the TOOLS package. The argument *simp_Virasoro_action* is the name of the simplification procedure for the monomial action.

Moreover we declare @ to be an infix operator with precedence after the operator *idifference* and we flag @ as *right* in order to facilitate multiple action.

```
algebraic;
multilinear !@({x, c, v}, simp_Virasoro_action);
infix !@; precedence(!@, idifference);
lisp flag('(!@), 'right)$
```

**3.** At top level the action of a basis element $x$ of the Virasoro algebra on a basis element $v$ of the Verma module is rather simple: if $x = c(0)$ the action is just $cv$ due to the fact that $c(0)$ is central, otherwise we must compute the action as a sum of basis elements using rules (1), (2) and (3). This last step is performed by the procedure *merge_Virasoro_action*, which will be described in the next section. Notice that we expect the values of $c$ and $h$ to be stored in the algebraic variables $c$ and $h$.

```
lisp procedure simp_Virasoro_action u;
   (if ¬member(car x, '(c x)) ∨ car v ≠ 'v then
         rederr("VIRASORO: invalid arguments")
   else if car x = 'c then multsq(mksq('c, 1), simp v)
      else merge_Virasoro_action(cdr x, cdr v))
         where x = cadr u, v = caddr u$
```

**4.** If we have to compute the action of an element $x = x(i)$ on an element $v = v(j_1, \ldots, j_k)$ we have to take into account the following points:

a. if $i < 0$ or $i = 0$ then we have to switch $i$ over all of the $j_l$'s using rule (3), i.e.

$$x(i) @v(j_1, j_2, \ldots) = x(j_1)@(x(i) @v(j_2, \ldots)) + [x(i), x(j_1)] @v(j_2, \ldots)$$

etc. since $x(i)@v() = 0$ or $x(i)@v() = hv()$, respectively.

b. if $i > 0$ and $i > j_1$ we have to switch $i$ and $j_1$ using rule (3) in order to get a basis element of the Verma module.

The recursive procedure *merge_Virasoro_action(action_stack, basis_element)* essentially takes care of the points raised above and computes, for *action_stack* $=$ '$(i_1 \ \ldots \ i_l)$ and *basis_element* $=$ '$(j_1 \ \ldots \ j_k)$, the action $x(i_l)@ \cdots @x(i_1)@v(j_1, \ldots, j_k)$ as a sum of basis elements (in standard quotient form) in the following way (with $i = i_1$, $j = j_1$):

1. if *action_stack* is empty then return '*v . basis_element* as a standard quotient.
2. if *basis_element* is empty then return
   - **nil** if $i < 0$,
   - *h∗merge_Virasoro_action(cdr action_stack, basis_element)* if $i = 0$
   - *merge_Virasoro_action(cdr action_stack, i . basis_element)* if $i > 0$.
3. if $i < 0$, $i = 0$ or $i > 0$ and $j < i$ then return

   *merge_Virasoro_action(i . j . cdr action_stack, cdr basis_element)* + *merge_Virasoro_action(*$[x(i), x(j)]$ *. cdr action_stack, cdr basis_element)*

   If $[x(i), x(j)]$ contains the central element $c(0)$, it can be removed from the *action_stack* and applied directly to the *cdr* of *basis_element*, since $c(0)$ commutes with all $x(i)$, yielding a term *c∗merge_Virasoro_action(cdr action_stack, cdr basis_element)*. Notice that $c(0)$ may only occur in the commutator for $i < 0$.
4. if $i > 0$ and $j > i$ then return *merge_Virasoro_action(cdr action_stack, i . basis_element)*

```
lisp procedure merge_Virasoro_action(action_stack,
      basis_element);
   if null action_stack then mksq('v . basis_element, 1)
   else if null basis_element then
            ⟨ Return result for empty basis_element 5 ⟩
     else (if i < 0 ∨ i = 0 ∨ (i > 0 ∧ j < i) then
            ⟨ Cases for i < 0 ∨ i = 0 ∨ (i > 0 ∧ j < i) 6 ⟩
       else merge_Virasoro_action(
            cdr action_stack, i . basis_element))
            where i = car action_stack,
            j = car basis_element$
```

**5.** Recall that we expected the values of $h$ and $c$ to be stored in the algebraic variables $h$ and $c$.

```
⟨ Return result for empty basis_element 5 ⟩ ≡
   (if i = 0 then multsq(mksq('h, 1),
         merge_Virasoro_action(cdr action_stack, basis_element))
   else if i > 0 then merge_Virasoro_action(
            cdr action_stack, i . basis_element))
            where i = car action_stack
```

This code is used in section 4.

**6.** The central term of the commutator $(j^3 - j)\delta_{i,-j}z/12$ need only be added if $i = -j \wedge j \neq 1$.

```
⟨ Cases for i < 0 ∨ i = 0 ∨ (i > 0 ∧ j < i) 6 ⟩ ≡
   addsq(merge_Virasoro_action(
         i . j . cdr action_stack, cdr basis_element),
         addsq(multsq((j − i) . 1, merge_Virasoro_action(
         (i + j) . cdr action_stack, cdr basis_element)),
         if i = −j ∧ j ≠ 1 then
           multsq(mksq('c, 1), multsq((j↑3 − j) . 12,
           merge_Virasoro_action(cdr action_stack,
           cdr basis_element))) else nil . 1))
```

This code is used in section 4.