Faculty of Applied Mathematics

University of Twente

University for Technical and Social Sciences

P.O. Box 217
7500 AE Enschede
The Netherlands
Phone +31-53-893400
Fax +31-53-356695
Telex 44200

MEMORANDUM No. 1099

The SUPER VECTORFIELD package for REDUCE. Version 1.0

G.H.M. ROELOFS

NOVEMBER 1992

ISSN 0169-2690

The SUPER VECTORFIELD package for REDUCE

Version 1.0

Marcel Roelofs

Abstract: We give the WEB source of the SUPER VECTORFIELD package for REDUCE. The package implements $\mathbf{Z}_2$-graded vectorfields and their action on $\mathbf{Z}_2$-graded functions in local coordinates in REDUCE. It can be used for the computation of symmetries and prolongation structures of (supersymmetric) systems of partial differential equations. The package is based on a former package by Gragert and Kersten.

AMS subject classification (1991): 14A22, 58A50, 68N99, 68Q40.
Keywords: non-commutative algebraic geometry, computer algebra software.

1. Super vectorfields in REDUCE. In this `WEB` file we shall implement the action of $\mathbf{Z}_2$ graded vectorfields on $\mathbf{Z}_2$ graded functions. The package is partially based on a former package by Gragert and Kersten (TW-memorandum 680), which also implemented $\mathbf{Z}_2$ graded forms and operators like exterior differentiation, Lie derivatives, etc. Since our methods nowadays mainly consist of using vectorfields, there is no direct need for an implementation of these operators.

The "banner line" defined here is intended for indentification purposes on loading. It should be changed whenever this file is modified. System dependent changes, however, should be made in a separate change file.

define *banner* ≡ `"Super␣vectorfield␣package␣for␣REDUCE␣3.4,␣$Revision:␣1.0␣$"`

2. We define the following macros for clarity.

define *change_to_symbolic_mode* ≡ **symbolic**
define *change_to_algebraic_mode* ≡ **algebraic**
define *stop_with_error*(*string_1*, *expr_1*, *string_2*, *expr_2*) ≡
    msgpri(*string_1*, *expr_1*, *string_2*, *expr_2*, t)
define *message*(*string_1*, *expr_1*, *string_2*, *expr_2*) ≡
    msgpri(*string_1*, *expr_1*, *string_2*, *expr_2*, **nil**)
define *operator_name_of* ≡ *car*
define *arguments_of* ≡ *cdr*
define *first_argument_of* ≡ *cadr*
define *second_argument_of* ≡ *caddr*
define *first_element_of* ≡ *car*
define *rest_of* ≡ *cdr*
define *skip_list* ≡ *cdr*   { Skip the '*list* in front of an algebraic list }

3. The following macros are intended as common programming idioms.

define *incr*(x) ≡ ($x := x + 1$)
define *decr*(x) ≡ ($x := x - 1$)

4. A new REDUCE switch can be introduced using the following code.

define *initialize_global*(*global_name*, *value*) ≡
    global '(*global_name*)$
    global_name := *value*
define *new_switch*(*switch_name*, *value*) ≡
    initialize_global(!* `@&`*switch_name*, *value*)$
    flag('(*switch_name*), '*switch*)

5. We do all initializations in the beginning of the package.

change_to_symbolic_mode$
write banner$ *terpri*( )$
⟨ Lisp initializations 7 ⟩
change_to_algebraic_mode$

6. We shall start with a (very) short description of the local picture of a graded manifold and vectorfields on these graded manifolds. For a more detailed description we refer to B. Kostant, Lecture Notes in Mathematics 570 (1977).

The local picture of a *graded manifold* is $U \subset \mathbf{R}^m$ open together with the *graded commutative algebra* $C^\infty(U) \otimes \Lambda(n)$ where $\Lambda(n)$ is the antisymmetric (exterior) algebra on n elements $s_1, \ldots, s_n$, with $\mathbf{Z}_2$-degree $|s_i| = 1$ and $s_i s_j = -s_j s_i$. A particular element $f \in C^\infty(U) \otimes \Lambda(n)$ is represented by $f = \sum_\mu f_\mu s_\mu$ where

$$\mu \in M_n = \{\mu = (\mu_1, \ldots, \mu_k) \mid \mu_i \in \mathbf{N}, 1 \le \mu_1 < \mu_2 < \cdots < \mu_k \le n\},$$

$s_\mu = s_{\mu_1} s_{\mu_2} \cdots s_{\mu_k}$ and $f_\mu \in C^\infty(U)$.

Graded vectorfields on a graded manifold $(U, C^\infty(U) \otimes \Lambda(n))$ are introduced as graded derivations of the algebra $C^\infty(U) \otimes \Lambda(n)$. It can be shown that they constitute a left $C^\infty(U) \otimes \Lambda(n)$-module. Locally a graded vectorfield V is represented as

$$V = \sum_{i=1}^{m} f_i \frac{\partial}{\partial x_i} + \sum_{j=1}^{n} g_j \frac{\partial}{\partial s_j}$$

with $f_i, g_j \in C^\infty(U) \otimes \Lambda(n)$ and x_i $(i = 1, \ldots, m)$ a local coordinate system on U.

The derivations $\dfrac{\partial}{\partial x_i}$ are even, while the derivation $\dfrac{\partial}{\partial s_j}$ are odd; they satisfy the relations

$$\frac{\partial x_i}{\partial x_k} = \delta_{ik}, \qquad \frac{\partial s_j}{\partial x_k} = 0, \qquad \frac{\partial x_i}{\partial x_\ell} = 0, \qquad \frac{\partial s_j}{\partial s_\ell} = \delta_{j\ell}.$$

7. In REDUCE we shall represent the elements $s_\mu \in \Lambda(n)$ by $\mathrm{EXT}(\mu_1, \ldots, \mu_k)$. Thus elements of $C^\infty(U) \otimes \Lambda(n)$ can be implemented in REDUCE as ordinary algebraic expressions.

$\langle$ Lisp initializations 7 $\rangle \equiv$

put('ext, 'simpfn, 'simpiden)$

See also section 37.

This code is used in section 5.

8. Initializing vectorfields. In order to introduce graded vectorfields, we need to know the local coordinates x_i on U, as well as the components of $\frac{\partial}{\partial x_i}$ and $\frac{\partial}{\partial s_j}$.

In this file we want to implement vectorfields as algebraic operators with a simplification procedure which takes care of the action on a function. It is our purpose to keep the local coordinates and the components local to one vectorfield at a time.

The following procedure initializes a super vectorfield. The macro *make_oplist* is taken from the TOOLS package; it transforms algebraic and lisp lists and identifiers into the appropriate lisp lists.

We will not give all components of the vectorfield here: it is much easier to give them separately, as we shall see in the sequel. For this purpose a vectorfield gets a *setkfn setk_super_vectorfield*, to be explained later.

```
define make_oplist(op_list) ≡
        if null op_list then op_list
        else if atom op_list then list op_list
          else if car op_list = 'list then cdr op_list
            else op_list
  lisp operator super_vectorfield;
  lisp procedure super_vectorfield(operator_name, even_variables, odd_variables);
    begin scalar odd_dimension;
    if ¬idp operator_name then
      stop_with_error("SUPER_VECTORFIELD:", operator_name, "is␣not␣an␣identifier", nil);
    put(operator_name, 'simpfn, 'super_der_simp); flag(list(operator_name), 'full);
    even_variables := make_oplist(even_variables);
    odd_variables := make_oplist(odd_variables); odd_dimension := 0;
    ⟨Adapt odd_dimension according to odd_variables 9⟩;
    put(operator_name, 'variables, even_variables);
    put(operator_name, 'even_dimension, length even_variables);
    put(operator_name, 'odd_dimension, odd_dimension);
    put(operator_name, 'setkfn, 'setk_super_vectorfield);
    return list('list, length even_variables, odd_dimension);
    end$
```

9. The list of *odd_variables* should only contain kernels of the *ext* operator with one integer argument. The *odd_dimension* is the maximum of the all integer arguments.

```
⟨Adapt odd_dimension according to odd_variables 9⟩ ≡
  for each kernel in odd_variables do
    if length kernel ≠ 2 ∨ operator_name_of kernel ≠ 'ext ∨ ¬fixp first_argument_of kernel then
          stop_with_error("SUPER_VECTORFIELD:", kernel, "not␣a␣valid␣odd␣variable", nil)
    else odd_dimension := max(odd_dimension, first_argument_of kernel)
```

This code is used in sections 8 and 12.

10. For non-super applications we provide *vectorfield* as an alias which initializes the *odd_variables* of a *super_vectorfield* to **nil**.

```
  lisp operator vectorfield;
  lisp procedure vectorfield(operator_name, variables);
    super_vectorfield(operator_name, variables, nil)$
```

11. Finally we provide two straightforward procedures for extending the number of variables of a vectorfield or a super vectorfield.

```
lisp operator add_variables_to_vectorfield;
lisp procedure add_variables_to_vectorfield(operator_name, variables);
  if get(operator_name, 'simpfn) ≠ 'super_der_simp then
        stop_with_error("ADD_VARIABLE_TO_VECTORFIELD:", operator_name, "not␣a␣vectorfield", nil)
  else
    ≪ variables := append(get(operator_name, 'variables), make_oplist(variables));
      put(operator_name, 'variables, variables);
      put(operator_name, 'even_dimension, length variables) ≫$
```

12.

```
lisp operator add_odd_variables_to_vectorfield;
lisp procedure add_odd_variables_to_vectorfield(operator_name, odd_variables);
  if get(operator_name, 'simpfn) ≠ 'super_der_simp then
        stop_with_error("ADD_VARIABLE_TO_VECTORFIELD:", operator_name, "not␣a␣vectorfield", nil)
  else
    begin scalar odd_dimension;
    odd_variables := make_oplist(odd_variables);
    odd_dimension := get(operator_name, 'odd_dimension);
    ⟨Adapt odd_dimension according to odd_variables 9⟩;
    return put(operator_name, 'odd_dimension, odd_dimension);
    end$
```

13. Implementation of exterior multiplication. Before we can implement the action of a graded vectorfield on a graded function we need to have a function that computes the (exterior) multiplication of two elements of $\Lambda(n)$.

If we have two elements $\mathrm{EXT}(i_1,\ldots,i_n)$ and $\mathrm{EXT}(j_1,\ldots,j_m)$ then the product will be 0 or an expression of the form $\pm\mathrm{EXT}(\ldots)$. In order to find this result we need to merge the lists $(i_1,\ldots,i_n)$ and $(j_1,\ldots,j_m)$ into one ordered list, taking into account the signs that occur due to the switching of all pairs of elements of the lists.

In fact, since it is needed for cohomology computations by van den Hijligenberg and Post, we shall implement an even more general procedure: given two *ordered* lists $(i_1,\ldots,i_m)$ and $(j_1,\ldots,j_m)$, return the list which results from merging the two lists into one ordered lists, together with a sign due to the switching of indices. The elements of the list need, however, not only be positive integers anymore, but may also be negative integers, with the proviso that switching two negative integers does *not* cause a sign.

The algorithm is rather simple: given two lists *x1* and *x2* we construct the merged list *x2* as follows (the notation *cx1* is an abbreviation for *car x1*, and the same for all other lists):

1. reverse *x1* (*x1* is now ordered reversely) and move all the elements of *x2*, with which the first element of *x1* (i.e. the highest element) has to be interchanged for merging both lists, in reverse order on the list *lx2*. Keep track if the number of elements of *lx2* is odd or even with help of the boolean *oddskip*.
2. if either *x1* or *lx2* is empty return the appropriate result.
3. if *cx1* = *clx2* then we can return **nil** if both are positive, due to the anticommutativity.
4. if *cx1* > *clx2* put *cx1* in front of *x2* and adjust the sign according to *oddskip* only if *cx1* is positive: if *cx1* is negative, so are all elements of *lx2* and thus no sign need to be added. Continue with 2.
5. if *cx1* ≤ *clx2* put *clx2* in front of *x2* and adjust *oddskip*. Continue with 2.

Since it is used quite frequently, we shall implement this procedure using labels in order to prevent overhead caused by (recursive) function calls.

```
lisp procedure merge_lists(x1, x2);
   begin scalar cx1, cx2, lx2, clx2, oddskip, sign;
   ⟨Prepare x1, x2 and lx2, if ready goto b 14⟩;
b: ⟨Weave all elements of x1 and lx2 in front of x2, return if done 15⟩;
   end$
```

14. The implementation of step 1.

```
⟨Prepare x1, x2 and lx2, if ready goto b 14⟩ ≡
   sign := 1; x1 := reverse x1;
   if x1 then cx1 := car x1 else goto b;
a: if x2 then cx2 := car x2 else goto b;
   if cx1 < cx2 then goto b;
   lx2 := cx2 . lx2;
   oddskip := ¬oddskip;
   x2 := cdr x2;
   goto a
```

This code is used in section 13.

15. The implementation of steps 2 and 3.

```
⟨Weave all elements of x1 and lx2 in front of x2, return if done 15⟩ ≡
  if null x1 then return sign . nconc(reversip lx2, x2);
  if null lx2 then return sign . nconc(reversip x1, x2);
  clx2 := car lx2;
  if cx1 = clx2 ∧ cx1 > 0 then return nil;
  if cx1 > clx2 then goto b1;
  ⟨Move first element of lx2 to x2 and goto b 16⟩;
b1: ⟨Move first element of x1 to x2 and goto b 17⟩
```

This code is used in section 13.

16. The implementation of step 5.

```
⟨Move first element of lx2 to x2 and goto b 16⟩ ≡
  x2 := clx2 . x2;
  lx2 := cdr lx2;
  oddskip := ¬oddskip;
  goto b
```

This code is used in section 15.

17. And finally step 4.

```
⟨Move first element of x1 to x2 and goto b 17⟩ ≡
  x2 := cx1 . x2;
  x1 := cdr x1;
  if oddskip ∧ cx1 > 0 then sign := −sign;
  if x1 then cx1 := car x1;
  goto b
```

This code is used in section 15.

18. It's a piece of cake now the write a procedure for the multiplication of two "EXT" kernels. By definition *ext*( ) is equal to 1.

```
define sign_of ≡ car
define arg_list_of ≡ cdr
  lisp procedure ext_mult(x1, x2);
    (if null x then nil ./ 1
    else if null arg_list_of x then 1 ./ 1
      else (((!*a2k('ext . arg_list_of x) .↑ 1) .* sign_of x) .+ nil) ./ 1)
    where x = merge_lists(arguments_of x1, arguments_of x2)$
```

19. The simplification procedure for vectorfields. The only thing left now is to implement the action of a vectorfield on a function by means of the simplification procedure *super_der_simp*.

If V is a vectorfield we shall assume that the components of $\frac{\partial}{\partial x_i}$ and $\frac{\partial}{\partial s_j}$ are given by $V(0,i)$ and $V(1,j)$, respectively.

Since we want to be able to look at the value of the components, we have to make the following distinction: if a vectorfield has just one argument it is the action on a function, otherwise we just have to return the value of the kernel.

```
lisp procedure super_der_simp u;
  if length u = 2 then ⟨Return the action of the vectorfield on a function 20⟩
  else simpiden u$
```

20. The action is not very complicated: collect all the even and odd components of the vectorfield and apply the vectorfield to the numerator and denominator of the function, using the quotient rule.

Notice that we don't want denominators of any function to contain odd variables, since such an expression can always be rewritten to a finite expression without odd variables in the denominator.

```
⟨Return the action of the vectorfield on a function 20⟩ ≡
  begin scalar derivation_name, variables, even_components, odd_components,
      splitted_numr, splitted_denr;
  derivation_name := reval operator_name_of u;
  variables := get(derivation_name, 'variables);
  u := simp!* first_argument_of u;
  ⟨Get the lists splitted_numr, splitted_denr, even_components and odd_components 22⟩;
  return subtrsq(
        quotsq(addsq(even_action(even_components, splitted_numr),
        odd_action(odd_components, splitted_numr)), denr u ./ 1),
        quotsq(multsq(numr u ./ 1, even_action(even_components, splitted_denr)),
        multf(denr u, denr u) ./ 1));
  end
```

This code is used in section 19.

21. Getting the vectorfield components. Finding all linear kernels of an algebraic operator and their coefficients in a standard form is performed by the procedure *split_form* of the TOOLS package, which acts on standard forms. Since it is more convenient for the components of the vectorfield to have the coefficients returned by *split_form* as standard quotients instead of standard forms, the following procedure applies *split_form* to the numerator of a standard quotient and takes care of the necessary conversion of the coefficients to standard quotients.

In order to allow simple processing of the lists the independent part must be preceded by *ext*().

```
define independent_part_of ≡ car
define kc_list_of ≡ cdr
define kernel_of ≡ car
define coefficient_of ≡ cdr

  lisp procedure split_ext(sq, op_list);
    begin scalar denr_sq, splitted_form;
    denr_sq := denr sq; splitted_form := split_form(numr sq, op_list);
    return (list('ext) . cancel(independent_part_of splitted_form ./ denr_sq)) .
      for each kc_pair in kc_list_of splitted_form collect
        (kernel_of kc_pair . cancel(coefficient_of kc_pair ./ denr_sq))
    end$
```

22. For a proper action of *even_action* and *odd_action* all components need to be decomposed into "EXT" kernels and their coefficients. Since the action is most conveniently performed recursively on standard forms, the numerator and denominator are decomposed at standard form level.

```
⟨Get the lists splitted_numr, splitted_denr, even_components and odd_components 22⟩ ≡
  splitted_numr := split_form(numr u, '(ext));
  splitted_numr := (list('ext) . independent_part_of splitted_numr) . kc_list_of splitted_numr;
  splitted_denr := split_form(denr u, '(ext));
  splitted_denr := (list('ext) . independent_part_of splitted_denr) . kc_list_of splitted_denr;
  even_components := for i := 1:get(derivation_name, 'even_dimension) collect
      (nth(variables, i) . split_ext(component, '(ext)))
          where component = simp!* list(derivation_name, 0, i);
  odd_components := for i := 1:get(derivation_name, 'odd_dimension) collect
      (i . split_ext(component, '(ext)))
          where component = simp!* list(derivation_name, 1, i)
```

This code is used in section 20.

23. Action of the even components. The action of the even part of a vectorfield on a function is fairly simple at top level: just add the actions on all kernel-coefficient pairs.

```
lisp procedure even_action(components, splitted_form);
  begin scalar action;
  action := nil ./ 1;
  for each kc_pair in splitted_form do
    action := addsq(action, even_action_sf(components, coefficient_of kc_pair, kernel_of kc_pair, 1));
  return action;
  end$
```

24. The action on a standard form is the sum of the actions on all terms. If the last term is a domain element we don't have to take it into consideration.

```
lisp procedure even_action_sf(components, sf, ext_kernel, fac);
  begin scalar action;
  action := nil ./ 1;
  while ¬domainp sf do
    ≪ action := addsq(action, even_action_term(components, lt sf, ext_kernel, fac)); sf := red sf ≫;
  return action;
  end$
```

25. For the action on the leading term we use the derivation property: the action on the leading power has to be added to the action on the leading coefficient. The last argument of *even_action_sf* is the product of all leading powers which have already been treated and with which the result has to be multiplied.

For reasons of efficiency it is more convenient to have the factor as in standard quotient in *even_action_pow*.

```
define term_pow ≡ car
define term_coeff ≡ cdr

lisp procedure even_action_term(components, term, ext_kernel, fac);
  addsq(even_action_pow(components, term_pow term, ext_kernel, !*f2q multf(fac, term_coeff term)),
      even_action_sf(components, term_coeff term, ext_kernel, multf(fac, !*p2f term_pow term)))$
```

26. Finally we have to implement the action on leading powers. For this we have to find all dependencies of the main variable on local coordinates occuring in the vectorfield, and act accordingly.

```
lisp procedure even_action_pow(components, pow, ext_kernel, fac);
  begin scalar kernel, n, component, derivative, action, active_components;
  kernel := car pow; n := cdr pow;   { pow = kernel↑n }
  ⟨If kernel is one the even local coordinates, return the action on pow 27⟩;
  ⟨Find all the dependencies of kernel and construct active_components 31⟩;
  ⟨Return the sum of the actions of active_components on pow 32⟩;
  end$
```

27. We can check if *kernel* is one of the local coordinates by a simple *assoc* on *components*.

```
⟨If kernel is one the even local coordinates, return the action on pow 27⟩ ≡
  if (component := assoc(kernel, components)) then
    return
      ≪ derivative := if n = 1 then 1 ./ 1 else ((((kernel .↑ n − 1) .* n) .+ nil) ./ 1);
        action := component_action(component, ext_kernel, derivative);
        multsq(action, fac) ≫
```

This code is used in section 26.

28. The procedure *component_action* takes care of returning the sum of all products of the *kc_pairs* in *component* with *ext_kernel* and *derivative*.

Recall that super vectorfields have a left $C^\infty(U) \otimes \Lambda(n)$ module structure. This means that we have to take care that the arguments in the *ext_mult* call have to be in the right order: components of the vectorfield left and the *ext_kernel*'s from the function right. Of course, if the product of the two "EXT" kernels is zero, there is no need to consider the summand.

```
define combined_product(x, y, z) ≡
        multsq(multsq(x, y), z)
  lisp procedure component_action(component, ext_kernel, coefficient);
    begin scalar action;
    action := nil ./ 1;
    for each kc_pair in kc_list_of component do
      (if numr ext_product then
        action := addsq(action, combined_product(ext_product, even_coefficient, coefficient)))
            where ext_product = ext_mult(kernel_of kc_pair, ext_kernel),
            even_coefficient = coefficient_of kc_pair;
    return action;
    end$
```

29. If a kernel is not one of the local coordinates, it may still depend on them, in which case we can still differentiate it w.r.t. such a coordinate.

The following procedure tries finds all active components in *kernel* as completely as possible.

```
define get_dependencies_of(kernel) ≡
        ((if depl_entry then cdr depl_entry) where depl_entry = assoc(kernel, depl!*))
  lisp procedure find_active_components(kernel, components, components_found);
    begin components_found :=
        update_components(kernel . get_dependencies_of(kernel), components, components_found)$
    if ¬atom kernel then
      for each element in kernel do
        components_found := find_active_components(element, components, components_found);
    return components_found;
    end$
```

30. The procedure *update_components* takes care that *components_found* contains all active components just once.

```
  lisp procedure update_components(dependencies, components, components_found);
    begin scalar component;
    for each kernel in dependencies do
      if (component := assoc(kernel, components)) ∧ ¬assoc(kernel, components_found) then
        components_found := component . components_found;
    return components_found;
    end$
```

31.

```
⟨ Find all the dependencies of kernel and construct active_components 31 ⟩ ≡
  active_components := find_active_components(kernel, components, nil)
```

This code is used in section 26.

32. Once we know all active components we can simply apply *diffp* to compute the derivatives of *pow* and *component_action* to compute the action of the different components. Recall that the final result has to be multiplied with *fac*.

⟨ Return the sum of the actions of *active_components* on *pow* 32 ⟩ ≡

```
action := nil ./ 1;
for each component in active_components do
   << derivative := diffp(pow, kernel_of component);
      action := addsq(action, component_action(component, ext_kernel, derivative)) >>;
return multsq(action, fac)
```

This code is used in section 26.

33. Action of the odd components. The action of the odd components is much simpler than the action of the even components since the dependencies are clear at once: the only dependency on odd variables are the indices of the "EXT" kernels.

Odd differentiations can cause an additional sign:

$$\frac{\partial}{\partial s_{i_j}} s_{i_1} \dots s_{i_j} \dots s_{i_n} = (-1)^{j-1} s_{i_1} \dots \widehat{s_{i_j}} \dots s_{i_n}$$

Additional signs are governed by the boolean *sign*. After the deletion of one index we have to apply *!*a2k* in order to get a unique kernel.

```
lisp procedure odd_action(components, splitted_form);
  begin scalar action, sign, derivative, kernel, coefficient, component;
  action := nil ./ 1;
  for each kc_pair in splitted_form do
    ≪ kernel := kernel_of kc_pair;
      coefficient := !*f2q coefficient_of kc_pair;
      sign := t;
      for each i in arguments_of kernel do
        ≪ sign := ¬sign;
          derivative := !*a2k delete(i, kernel);
          component := assoc(i, components);
          action := addsq(action, component_action(component, derivative,
              if sign then negsq coefficient else coefficient)) ≫ ≫;
  return action;
  end$
```

34. Assigning values to vectorfield components. If V is a vectorfield we recall that the components of $\frac{\partial}{\partial x_i}$ and $\frac{\partial}{\partial s_j}$ are given by $V(0,i)$ and $V(1,j)$, respectively. However, assigning a value to, for instance, $V(0,i)$ has to be done very thoughtfully, since the correspondence between the index i and the i-th variable x_i is mostly not a logical one in practical situations. It would be much easier if one could say $V(x_i) := y$, if $V(0,i)$ has to become y.

Such a task can be easily accomplished by using a *setkfn*: if an algebraic operator possesses an indicator *setkfn*, this function is used for assignment instead of the default method, which is a call to *let2*. For vectorfields we introduce the *setkfn* *setk_super_vectorfield*, which takes care of the kind of assignments described above. This is fairly simple: if the number of arguments of *val* below is not 1, we can just apply the default call to *let2*, otherwise *val* apparently is of the form $V(x_i)$ or $V(s_j)$ and we must store *value* in $V(0,i)$ or $V(1,j)$, respectively.

lisp procedure *setk_super_vectorfield*(*val*, *value*);
  begin scalar *vectorfield*, *var*, *variables*, *i*, *tuple*;
  if *length val* $\neq$ 2 **then return** *let2*(*val*, *value*, **nil**, **t**);
  vectorfield := *operator_name_of val*; *var* := *first_argument_of val*;
  ⟨ If possible, translate *var* into an appropriate *tuple* 35 ⟩;
  return *let2*(*vectorfield* . *tuple*, *value*, **nil**, **t**);
  end$

35. If *var* = *ext*(j) then *tuple* must be $(1,j)$, else if *var* is the i-th entry of the even *variables* associated to v, *tuple* must be $(0,i)$. In all other cases no assignment is useful and we can return with an error.

⟨ If possible, translate *var* into an appropriate *tuple* 35 ⟩ ≡
  tuple := **if** ¬*atom var* ∧ *operator_name_of var* = '*ext* ∧ *length var* = 2 **then**
      list(1, *first_argument_of var*)
   else ≪ *variables* := *get*(*vectorfield*, '*variables*); *i* := 1;
     while *variables* ∧ *var* ≠ *first_element_of variables* **do**
       ≪ *variables* := *rest_of variables*; *incr*(*i*) ≫;
     if *null variables* **then**
       stop_with_error(`"SETK_SUPER_VECTORFIELD: "`, *var*, `"not␣a␣valid␣variable␣for"`, *vectorfield*)
     else *list*(0, *i*) ≫

This code is used in section 34.

36. Multiplication of graded expressions. Since it is useful in practical problems, we shall finally implement a procedure *super_product* for multiplying two graded expressions. Using some of the above procedures this is not difficult at all.

```
format product = car
  lisp operator super_product;
  lisp procedure super_product(x, y);
    begin scalar splitted_x, splitted_y, product;
    splitted_x := split_ext(simp x, '(ext)); splitted_y := split_ext(simp y, '(ext));
    product := nil ./ 1;
    for each term_x in splitted_x do
      for each term_y in splitted_y do
        product := addsq(product,
            combined_product(coefficient_of term_x, coefficient_of term_y,
            ext_mult(kernel_of term_x, kernel_of term_y)));
    return mk!*sq subs2 product;
    end$
```

37. In order to facilitate natural input we will implement a switch *natural_wedges* which introduce a new token *!^!^* in REDUCE that parses left associative to *super_product* and takes precedence over *times*. In conjunction with this token we assign a print function to the *ext* operator, which takes care of eventual aliases of *ext*-kernels, introduced by the *operator_representation* of the TOOLS package.

We start with the definition of the switch *natural_wedges*. By assigning the *simpfg* property to the switch *natural_wedges* we can make the appropriate call to the procedure *natural_wedges_handler* if it is put **on** or **off** , respectively.

```
⟨ Lisp initializations 7 ⟩ +≡
  new_switch(natural_wedges, nil)$
  put('natural_wedges, 'simpfg, '((t(natural_wedges_handler t)) (nil(natural_wedges_handler nil))))$
```

38. The handler *natural_wedges_handler* prepares and removes the token *!^!^* and the print function *wedge_print*.

```
  lisp procedure natural_wedges_handler on_off;
    begin scalar save_switch;
    if on_off then
      ≪ newtok '((!^ !^) super_product); precedence('super_product, 'times);
        put('ext, 'prifn, 'wedge_print) ≫
    else
      ≪ save_switch := get('!^, 'switch!*);
        save_switch := delete(assoc('!^, car save_switch), car save_switch) . cdr save_switch;
        put('!^, 'switch!*, save_switch); remprop('ext, 'prifn) ≫
    end$
```

39. The print function *wedge_print* is fairly simple: if the operator has one argument use *print_alias* for printing, which checks for aliases, otherwise apply *inprint* on the list of arguments surrounded by *ext*.

```
  lisp procedure wedge_print ext_kernel;
    if length ext_kernel ≤ 2 then print_alias ext_kernel
    else inprint('super_product, 0, kernels_on_list)
      where kernels_on_list = for each arg in arguments_of ext_kernel collect list('ext, arg)$
```

40. The end of a REDUCE input file must be marked with **end**.

```
  end;
```

41. Index. This section contains a cross reference index of all identifiers, together with the numbers of the mdules in which they are used. Underlined entries correspond to module numbers where the identifier was declared.