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. 1100

The INTEGRATOR package for REDUCE. Version 1.0

G.H.M. ROELOFS

NOVEMBER 1992

ISSN 0169-2690

THE INTEGRATOR PACKAGE FOR REDUCE

Version 1.0

MARCEL ROELOFS

Abstract: We give the WEB source of the INTEGRATOR package for REDUCE. The package can be used to solve overdetermined systems of partial differential equations, especially those occuring in the computation of symmetries and prolongation structures of (supersymmetric) systems of partial differential equations. The package is based on a former package by Kersten.

AMS subject classification (1991): 35N99, 58G37, 68N99, 68Q40, 70H33.
Keywords: overdetermined systems, computer algebra software, symmetries, prolongation structures.

1. Introduction. In this WEB file we shall describe a REDUCE package for the integration of overdetermined systems of partial differential equations (p.d.e.'s). This work is mainly based on a similar package by Paul Kersten for just the determination of symmetry groups (CWI tract 34, *Infinitesimal symmetries: a computational approach*, CWI, Amsterdam, 1987) and an extension by myself which also allows the determination of Wahlquist and Estabrook prolongation algebras.

The main reasons for the implementation of this package, are our improved insight in the internals of REDUCE, the wish to have one combined integrator for both cases and the availability of substantially improved versions of some the procedures used in the former packages.

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* ≡ `"Integrator␣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 }
format *function* = *identifier*

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 *initialize_fluid*(*fluid_name*, *value*) ≡
    fluid '(*fluid_name*)$
    fluid_name := *value*
define *new_switch*(*switch_name*, *value*) ≡
    initialize_fluid(*!* @&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 9 ⟩
change_to_algebraic_mode$

6. Integration of overdetermined systems of p.d.e.'s. For the determination of symmetry groups or prolongation structures of (systems of) partial differential equations, the defining relations give rise to an overdetermined system of p.d.e.'s. Finding the symmetry group or prolongation structure boils down to solving such a system.

There are, however, some differences between the determination of a symmetry group or the determination of a prolongation structure. These differences are:

1. The differential equations for the determination of the symmetry group are linear, the equations for the determination of a prolongation structure are nonlinear. This nonlinearity, however, is of a special kind, namely, the only occuring nonlinear terms are (possibly nested) liebrackets of the functions to be integrated.
2. For the determination of symmetry groups, the functions to be determined integrate to polynomials with constant coefficients. For the determination of prolongation structures, functions integrate to polynomials, coefficients of which are generators of some unknown Lie algebra. The defining relations of this algebra are the remaining (nonlinear) relations which have no dependency on the independent variables involved.

From the above it is clear that integration has to be treated slightly different in either of the cases. The differences are however small enough to allow the implementation of one integrator for both cases.

7. In order to explain all possible p.d.e.'s which can be integrated, we make the following assumptions:

1. Functions are represented by expressions $f(n)$, where f is some specified operator and n is an integer. Since we intend to use the package for computations for supersymmetric p.d.e.'s, we shall use the notion the elements with n positive must integrate to an even polynomial and elements with negative n must integrate to an odd polynomial (this is only useful for computations in prolongation theory, where coefficients can be even or odd Lie algebra generators).
2. The dependencies of functions are solely listed on the dependency list, i.e. must be stated by the 'depend' statement of REDUCE. Notice, however, that we do not allow dependencies of odd variables. The reason for this is a pragmatic one: due to the anticommutivity of odd variables, n odd variables can only produce 2^n different terms containing these variables, hence can be stated explicitly provided that n is not too big. On the other hand, if we allow dependencies of odd variables, a lot of additional operators have to be implemented to take care of e.g. partial differentation w.r.t. odd variables.

8. If f is the operator denoting functions, x the operator denoting Lie algebra generators (or, in the case of a symmetry group, just constants), then following the description above, a p.d.e. has the following possible terms (any coefficient c is always some polynomial in the independent variables):

A. terms of the form $c_n \mathrm{df}(f(n), \ldots)$.
B. terms of the form $c_n f(n)$.
C. terms of the form $c_{1,2}[z_1, z_2]$ where z_1, z_2 are either functions $f(n)$ or Lie algebra generators $x(n)$.
D. terms of the form $c_n x(n)$.

These possibilities lead, in a natural way, to the following strategy of solving the p.d.e.'s:

1. If there is only one term of type A, we can integrate this equation homogeneously, i.e. give a polynomial expression for $f(n)$ using the variables involved in the differential term.
2. If the p.d.e. is a polynomial in one or more independent variables on which none of the occuring functions depend, all coefficients of this polynomial have to be zero, i.e., the p.d.e. splits up into a set of smaller p.d.e.'s.
3. If there are only terms of type C and D we have a Lie algebra relation, which can be solved by the LIESUPER package, if solvable.
4. If there is a function of type B depending on all variables occuring in the p.d.e. and not occuring in a term of type A, we can solve for this function.
5. If there is one term of type A depending on all variables occuring the p.d.e. and the remaining terms are polynomial in the variables occuring in the derivative, the p.d.e. can be integrated inhomogeneously.
6. If there is just one function in the p.d.e. which depends on a variable only occuring polynomially in the rest of the p.d.e., such that the p.d.e. can not be integrated inhomogeneously since the dependencies of the various occuring functions do not match, we can introduce new equations of type 1 by appropriately differentiating the p.d.e.

9. Initializing an equation set. The integrator will be implemented in such a way that integration can be performed on different sets of p.d.e.'s at the same time. Different sets of p.d.e.'s will be distinguished by the name of the operator in which they are stored.

For each operator representing a set of p.d.e.'s we must know: the name of the operator(s) representing the functions and the operator that must be used to represent constant coefficients during the integration. If this last operator possesses the indicator *bracketname* we know that the operator is declared as a Lie algebra generator, hence that we are in the prolongation case. In this case the name of the associated liebracket is equal to the value of *bracketname*.

Moreover, we have to know the total number of equations used, in view of the additional equations that may be generated and which must be numbered subsequently. In connection with the integrations taking place we also have to know the number of functions, resp. constants (generators) being in use.

This is all taken care of by the procedure *initialize_equations*, which assigns to an operator *operator_name*, the total number of used equations *total_used*, the list *variable_list* of all occuring independent variables, the operator *constant_operator*, elements of which act as constants, and an arbitrary number of operators *function_operator* acting as functions. *constant_operator* and each *function_operator* should be given a an algebraic list of the form {operator, number of even elements used, number of odd elements used}.

In order to allow an arbitrary number of parameters we make *initialize_equations* a *psopfn*. How *psopfn*'s are dealt with internally is explained in the documentation of either the TOOLS package or the LIESUPER package.

⟨ Lisp initializations 9 ⟩ ≡

```
put('initialize_equations,'psopfn,'initialize_equations1)$
```

See also sections 13, 18, 33, 53, and 66.

This code is used in section 5.

10.

```
lisp procedure initialize_equations1 specification_list;
  begin scalar operator_name, total_used, variable_list, specification, even_used, odd_used,
      constant_operator, bracketname, function_name, function_list;
  if length specification_list < 5 then
      rederr("INITIALIZE_EQUATIONS: wrong number of parameters");
  if ¬idp(operator_name := first_element_of specification_list) then
      rederr("INITIALIZE_EQUATIONS: equations operator must be identifier");
  if ¬fixp(total_used := reval first_element_of(specification_list := rest_of
      specification_list)) ∨ total_used < 0 then
      rederr("INITIALIZE_EQUATIONS: total number of equations must be positive");
  put(operator_name,'total_used,total_used);
  variable_list := reval first_element_of(specification_list := rest_of specification_list);
  if atom variable_list ∨ operator_name_of variable_list ≠ 'list then
      rederr("INITIALIZE_EQUATIONS: variable list must be algebraic list");
  put(operator_name,'variable_list,skip_list variable_list);
  ⟨ Check and initialize constant_operator 11 ⟩;
  ⟨ Check and initialize function_list 12 ⟩;
  end$
```

11. The *constant_operator* can either be a Lie algebra generator or not. If so, we also have to assign the associated liebracket to *operator_name* and used the procedure *define_used* to take care of the assignment of the used dimensions to the liebracket. If *constant_operator* is not a Lie algebra generator, we store these dimensions in the same way as happens for liebrackets.

```
define check_valid_function_declaration(op_list, op_name) ≡
        if atom op_list ∨ length op_list ≠ 4 ∨ operator_name_of op_list ≠ 'list
                ∨ ¬idp(op_name := first_argument_of op_list) ∨ ¬fixp(even_used := reval caddr op_list)
                ∨ ¬fixp(odd_used := reval cadddr op_list) ∨ even_used < 0 ∨ odd_used < 0 then
          stop_with_error("INITIALIZE_EQUATIONS: invalid declaration of", op_list, nil, nil)
define put_used_dimensions(op_name, even_used, odd_used) ≡
        if get(op_name, 'bracketname) then define_used(bracketname, list('list, even_used, odd_used))
        else
          begin put(op_name, 'even_used, even_used);
          put(op_name, 'odd_used, odd_used);
          end
```

⟨ Check and initialize *constant_operator* 11 ⟩ ≡

```
  specification_list := rest_of specification_list; specification := first_element_of specification_list;
  check_valid_function_declaration(specification, constant_operator);
  put(operator_name, 'constant_operator, constant_operator);
  if (bracketname := get(constant_operator, 'bracketname)) then
    put(operator_name, 'bracketname, bracketname);
  put_used_dimensions(constant_operator, even_used, odd_used)
```

This code is used in section 10.

12.

⟨ Check and initialize *function_list* 12 ⟩ ≡

```
  for each function_specification in rest_of specification_list do
    begin check_valid_function_declaration(function_specification, function_name);
    put_used_dimensions(function_name, even_used, odd_used);
    function_list := function_name . function_list;
    end;
  put(operator_name, 'function_list, function_list)
```

This code is used in section 10.

13. Since we can apparently choose different sets of p.d.e.'s for solving, we must tell the integrator which set to take. This is done via a global variable *cur_eq_set!**. We will take the operator *equ* as the default *cur_eq_set!**. In this file we will use the abbreviation *ces!** for *cur_eq_set!**.

```
define ces!* ≡ cur_eq_set!*
```

⟨ Lisp initializations 9 ⟩ + ≡

```
  initialize_global(ces!*, 'equ)$
```

14.

```
  lisp operator use_equations;
  lisp procedure use_equations operator_name;
    begin
    if idp operator_name then ces!* := operator_name
    else rederr("USE_EQUATIONS: argument must be identifier");
    end$
```

15. The integration procedure. The implementation of the integrator follows the description of all the possible steps given above.
For the use of the fluid variable *listpri_depth!**, see below. Its local rebinding is necessary for a proper printing of the messages given by the procedure.

```
lisp operator integrate_equation;
lisp procedure integrate_equation n;
  begin scalar listpri_depth!*, total_used, equation, denominator, solvable_kernel, solvable_kernels,
      df_list, function_list, present_functions_list, variable_list, absent_variables, linear_functions_list,
      constants_list, bracketname, df_terms, df_functions, linear_functions, functions_and_constants_list,
      commutator_functions, present_variables, nr_of_variables, integration_variables;
  listpri_depth!* := 200; terpri!* t;
  ⟨Find the equation to be integrated 16⟩;
  ⟨Step 1: search for homogeneous integration 20⟩;
  ⟨Step 2: search for polynomial behaviour 29⟩;
  ⟨Step 3: search for a Lie relation 37⟩;
  ⟨Step 4: search for a solvable function 39⟩;
  ⟨Step 5: search for inhomogeneous integration 43⟩;
  ⟨Step 6: search for a useful differentation 54⟩;
  ⟨Step 7: print a "Not solved" message 59⟩;
solved:    { Go here when the equation is solved or its type is determined }
  end$
```

16. The part of the equation containing all necessary information is its numerator. For reasons that will become clear in the sequel we need, however, also know its denominator. If the equation is zero, no analysis has to be performed.

```
define nullify_equation(n) ≡
        setk(list(ces!*, n), 0)

⟨Find the equation to be integrated 16⟩ ≡
  if null(total_used := get(ces!*, 'total_used)) ∨ n > total_used then
        stop_with_error("INTEGRATE_EQUATIONS:␣properly␣initialize", ces!*, nil, nil);
  if null(equation := cadr assoc(list(ces!*, n), get(ces!*, 'kvalue))) then
        stop_with_error("INTEGRATE_EQUATION:", list(ces!*, n), "is␣non-existent", nil);
  denominator := denr(equation := simp!* equation); equation := numr equation;
  if null equation then
    ≪ write ces!*, "(", n, ")␣=␣0"; terpri!* t; nullify_equation(n); goto solved ≫
```

This code is used in section 15.

17. Homogeneous integration. Homogeneous integration must be performed if the equation consists of just one *df* term. In order to find all possible *df* terms we apply *split_form* to *equation*. This returns a list the *car* of which is the part of *equation* independent of the *df* operator, the *cdr* of which is a list of all linear *df* terms, together with their coefficients. *split_form* will return with an error if nonlinear *df* terms occur.

```
define independent_part_of ≡ car
define kc_list_of ≡ cdr
define kernel_of ≡ car    { For use with a kernel-coefficient list }
define coefficient_of ≡ cdr    { For use with a kernel-coefficient list }
```

18. If there is one *df* term, we only solve it if its coefficient is a number, by default. This behaviour is governed by the switch *coefficient_check*, which is **on** by default. In order to check the coefficient we will use the procedure *find_solvable_kernel* to be explained below.

```
⟨ Lisp initializations 9 ⟩ + ≡
  new_switch(coefficient_check, t)$
```

19. Before continuing we introduce some auxiliary macros and procedures.

```
define assoc_delete(kernel, assoc_list) ≡
        delete(assoc(kernel, assoc_list), assoc_list)
  lisp procedure successful_message_for(n, action, kernel);
    ≪ write ces!*, "(", n, "):␣", action; maprin kernel; terpri!*(¬!*nat);
      nullify_equation(n); t≫$
  lisp procedure not_a_number_message_for(n, action, kernel);
    ≪ write "***␣", ces!*, "(", n, "):␣", action, "␣failed:"; terpri!* t;
      write "␣␣␣␣coefficient␣not␣a␣number␣for␣"; maprin kernel; terpri!*(¬!*nat);
      write "␣␣␣␣Solvable␣with␣'off␣coefficient_check'";
      terpri!* t; t≫$
```

20.

```
⟨ Step 1: search for homogeneous integration 20 ⟩ ≡
  df_list := split_form(equation, '(df));
  if try_a_homogeneous_integration(n, denominator, df_list) then goto solved
```

This code is used in section 15.

21.

```
  lisp procedure try_a_homogeneous_integration(n, denominator, df_list);
    begin scalar solvable_kernel, solvable_kernels, df_kernel;
    return
      if null independent_part_of df_list ∧ (kc_list_of df_list) ∧ length(kc_list_of df_list) = 1 then
        if (solvable_kernel := find_solvable_kernel(
              solvable_kernels := list(kernel_of first_element_of kc_list_of df_list),
              kc_list_of df_list, denominator)) then
          ≪ df_kernel := first_argument_of solvable_kernel;
            setk(df_kernel, homogeneous_integration_of(solvable_kernel));
            depl!* := assoc_delete(df_kernel, depl!*);    { Remove df_kernel from the depl!* list }
            successful_message_for(n, "Homogeneous␣integration␣of␣", solvable_kernel) ≫
        else not_a_number_message_for(n, "Homogeneous␣integration", first_element_of solvable_kernels)
    end$
```

22. The procedure *find_solvable_kernel* tries to find the first element of *kernel_list* which has a number as coefficient. If *coefficient_check* is **off** we can simply take the first element of *kernel_list*, otherwise we can most conveniently implement a recursive procedure *first_solvable_kernel*, which finds the first element of *kernel_list* with a number as coefficient. We can check this by first checking if the numerator of the coefficient is a domain element, or if the whole coefficient is a number.

```
lisp procedure find_solvable_kernel(kernel_list, kc_list, denominator);
  if !*coefficient_check then first_solvable_kernel(kernel_list, kc_list, denominator)
  else first_element_of kernel_list$

lisp procedure first_solvable_kernel(kernel_list, kc_list, denominator);
  if kernel_list then
    (if domainp coefficient_of kc_pair ∨ numberp !*ff2a(coefficient_of kc_pair, denominator) then
      kernel_of kc_pair
    else first_solvable_kernel(rest_of kernel_list, kc_list, denominator))
        where kc_pair = assoc(first_element_of kernel_list, kc_list)$
```

23. The equation

$$\frac{\partial^{k_1}}{\partial {x_1}^{k_1}} \cdots \frac{\partial^{k_m}}{\partial {x_m}^{k_m}} f(x_1, \ldots, x_n) = 0 \qquad (m \leq n)$$

has general solution

$$f = \sum_{j=1}^{m} \sum_{i_j=0}^{k_j-1} x_j^{i_j} f_{j,i_j}(x_1, \ldots, \widehat{x_j}, \ldots, x_n).$$

Thus, given a homogenous p.d.e., *homogeneous_integration_of* has to return the REDUCE equivalent of the last expression.

If f depends on only one variable the f_{j,i_j} are constants, otherwise they are new functions with dependency on one less variable. In the Lie algebra case the constants are generators of the Lie algebra. Since the dimensions of a *liebracket* in REDUCE have to be given on beforehand, there may not be enough generators left to generate f. In this case, we have to enlarge the *liebracket*.

```
define get_dependencies_of(kernel) ≡
        ((if depl_entry then cdr depl_entry)
              where depl_entry = assoc(kernel, depl!*))

lisp procedure homogeneous_integration_of df_term;
  begin scalar df_function, function_number, dependency_list, integration_list, coefficient_name,
      bracketname, even_used, odd_used, integration_variable,
      number_of_integrations, solution, new_dependency_list;
  ⟨Check if df_term can be integrated, find df_function and function_number 24⟩;
  dependency_list := get_dependencies_of(df_function);
  if length dependency_list = 1 then coefficient_name := get(ces!*,'constant_operator)
  else coefficient_name := operator_name_of df_function;
  ⟨Get even_used, odd_used and if necessary bracketname 25⟩;
  integration_list := rest_of arguments_of df_term;
  ⟨Find the next integration_variable and number_of_integrations 26⟩;
  if bracketname then ⟨Check and possibly enlarge dimensions of bracketname 27⟩;
  ⟨Perform the integration 28⟩;
  return solution
  end$
```

24. We required *df_term* to be of the form $df(f(k),\ldots)$ where f is a function occuring on the *function_list* of *ces!** and k is an integer not equal to zero.

```
⟨ Check if df_term can be integrated, find df_function and function_number 24⟩ ≡
  df_function := first_argument_of df_term;
  if ¬member(operator_name_of df_function, get(ces!*, 'function_list))
        ∨ ¬fixp(function_number := first_argument_of df_function) ∨ function_number = 0 then
    stop_with_error("PERFORM_HOMOGENEOUS_INTEGRATION:␣integration␣of", df_function,
        "not␣allowed", nil)
```

This code is used in section 23.

25. In the liebracket case *even_used* and *odd_used* are stored as properties of *bracketname* instead of *coefficient_name*.

```
⟨ Get even_used, odd_used and if necessary bracketname 25⟩ ≡
  if (bracketname := get(coefficient_name, 'bracketname)) then
    begin even_used := get(bracketname, 'even_used);
    odd_used := get(bracketname, 'odd_used);
    end
  else
    begin even_used := get(coefficient_name, 'even_used);
    odd_used := get(coefficient_name, 'odd_used);
    end
```

This code is used in section 23.

26. Finding the integration variables is rather straightforward.

```
⟨ Find the next integration_variable and number_of_integrations 26⟩ ≡
  if integration_list then integration_variable := first_element_of integration_list
  else integration_variable := nil;
  if integration_variable ∧ (integration_list := rest_of integration_list)
        ∧ fixp first_element_of integration_list then
    ≪ number_of_integrations := first_element_of integration_list;
      integration_list := rest_of integration_list ≫
  else number_of_integrations := 1
```

This code is used in sections 23 and 28.

27. If *df_function* depends on only one variable, the number of constants being introduced is equal to the *number_of_integrations*. The even and odd dimension of *bracketname* are stored as the properties *even_dimension* and *odd_dimension*.

```
⟨ Check and possibly enlarge dimensions of bracketname 27⟩ ≡
  if function_number > 0 then
    (if even_used + number_of_integrations > get(bracketname, 'even_dimension) then
      change_dimensions_of(bracketname, even_used + number_of_integrations,
          get(bracketname, 'odd_dimension)))
  else
    (if odd_used + number_of_integrations > get(bracketname, 'odd_dimension) then
      change_dimensions_of(bracketname, get(bracketname, 'even_dimension),
          odd_used + number_of_integrations))
```

This code is used in section 23.

28. The actual integration is fairly straightforward by now: for all the possible integration variables we can simply add new terms to *solution*.

```
define new_coefficient ≡
        list(coefficient_name,
            if function_number > 0 then incr(even_used)
            else −incr(odd_used))
define ext_mksq(kernel, power) ≡
        if power = 0 then 1 ./ 1
        else mksq(kernel, power)
define depend_new_coefficient(dependency_list) ≡
        depl!* := (list(coefficient_name,
            if function_number > 0 then even_used
            else −odd_used) . dependency_list) . depl!*;
⟨ Perform the integration 28 ⟩ ≡
  solution := nil ./ 1;
  while integration_variable do
    begin new_dependency_list := delete(integration_variable, dependency_list);
    for i := 0:number_of_integrations − 1 do
      ≪ solution := addsq(solution, multsq(ext_mksq(integration_variable, i), mksq(new_coefficient, 1)));
        if new_dependency_list then depend_new_coefficient(new_dependency_list) ≫;
    ⟨ Find the next integration_variable and number_of_integrations 26 ⟩
    end;
  solution := mk!*sq subs2 solution;
  put_used_dimensions(coefficient_name, even_used, odd_used)
```

This code is used in section 23.

29. Splitting polynomial equations. For the polynomial behaviour of *equation* we need to know the dependencies of all the functions occuring in *equation* at any level. If there occur any other variables in *equation* and *equation* is polynomial in these variables, the coefficients of this polynomial give rise to a new set of equations.

```
define pc_list_of ≡ kc_list_of    { power-coefficient list }
define powers_of ≡ kernel_of
⟨ Step 2: search for polynomial behaviour 29 ⟩ ≡
  ⟨ Find present_functions_list and the absent_variables 30 ⟩;
  if split_equation_polynomially(n, total_used, equation, absent_variables) then goto solved
```

This code is used in section 15.

30. Finding all the functions in *equation* can be done by applying the procedure *get_recursive_kernels* of the TOOLS package.

```
⟨ Find present_functions_list and the absent_variables 30 ⟩ ≡
  function_list := get(ces!*, 'function_list);
  present_functions_list := get_recursive_kernels(equation, function_list);
  variable_list := get(ces!*, 'variable_list); absent_variables := variable_list;
  for each function in present_functions_list do
    for each variable in get_dependencies_of(function) do
      absent_variables := delete(variable, absent_variables)
```

This code is used in section 29.

31.

```
lisp procedure split_equation_polynomially(n, total_used, equation, absent_variables);
  begin scalar polynomial_variables, equations_list;
  ⟨ Find the polynomial_variables and test for polynomial behaviour 32 ⟩;
  ⟨ If possible, split up equation into smaller equations 35 ⟩
  end$
```

32. In most cases the equations under consideration are polynomial in any of the variables and therefore we shall by default not test for polynomial behaviour. This testing is governed by the switch *polynomial_check* which, be default, is **off** . If it is **on** testing is done by the procedure *polynomialp* to be defined below.

```
⟨ Find the polynomial_variables and test for polynomial behaviour 32 ⟩ ≡
  polynomial_variables := absent_variables;
  if !*polynomial_check then
    polynomial_variables := for each variable in polynomial_variables join
        if polynomialp(equation, variable) then list(variable)
```

This code is used in section 31.

33.

```
⟨ Lisp initializations 9 ⟩ + ≡
  new_switch(polynomial_check, nil)$
```

34. Checking a standard form for polynomial behaviour in some kernel can be done by checking the main variable, the leading coefficient and the reductum, respectively.

```
lisp procedure polynomialp(expression, kernel);
  if domainp expression then t
  else ((main_variable = kernel ∨ ¬depends(main_variable, kernel))
          ∧ polynomialp(lc expression, kernel) ∧ polynomialp(red expression, kernel))
        where main_variable = mvar expression$
```

35. The coefficients of a polynomial can be found by applying the procedure *multi_split_form* from the TOOLS package.

If *equation* can be split into smaller equations, *split_equation_polynomially* has to return **t**.

```
⟨ If possible, split up equation into smaller equations 35 ⟩ ≡
  equations_list := multi_split_form(equation, polynomial_variables);
  if length equations_list > 1 then
    ≪for each pc_pair in pc_list_of equations_list do
        setk(list(ces!*, incr(total_used)), mk!*sq((coefficient_of pc_pair) ./ 1));
      if independent_part_of equations_list then
        setk(list(ces!*, incr(total_used)), mk!*sq((independent_part_of equations_list) ./ 1));
      write ces!*, "(", n, ")␣breaks␣into␣", ces!*, "(", get(ces!*, 'total_used) + 1,
          "), . . . , ", ces!*, "(", total_used, ")␣by␣";
      maprin partial_list(polynomial_variables, 5); terpri!*(¬!*nat);
      nullify_equation(n); put(ces!*, 'total_used, total_used)≫;
  if length equations_list > 1 then
    return t
```

This code is used in section 31.

36. In order to get messages in a readable form, we sometimes need to print lists partially. This is taken care of the following procedures.

```
lisp procedure partial_list(printed_list, nr_of_items);
  'list . broken_list(printed_list, nr_of_items)$

lisp procedure broken_list(list, n);
  if list then
    if n = 0 then '(!.!.!.)
    else car list . broken_list(cdr list, n − 1)$
```

37. Solving Lie algebra relations. If the first two steps have failed, we need to analyze *equation* in a more drastic way: we need to find all functions occuring linearly in *equation*, and if a liebracket is specified, all commutators and algebra generators occuring in *equation* as well. Since we have already looked for *df* terms in *equation* in each next step we only have to examine the independent part of the previous step.

⟨ Step 3: search for a Lie relation 37 ⟩ ≡

```
linear_functions_list := split_form(independent_part_of df_list, function_list);
df_list := kc_list_of df_list;
constants_list := split_form(independent_part_of linear_functions_list, list get(ces!*, 'constant_operator));
linear_functions_list := kc_list_of linear_functions_list;
if (bracketname := get(ces!*, 'bracketname)) then ⟨ Solve equation if it is a Lie expression 38 ⟩
```

This code is used in section 15.

38. In the Lie algebra case we can try to solve the Lie expression if there are no *df* terms or linearly occuring functions. Solving Lie expression can be done using the procedure *relation_analysis* of the LIESUPER package. *relation_analysis* returns either the kernel for which the relation is solved or an atom indicating the nature of the non-solvability.

⟨ Solve *equation* if it is a Lie expression 38 ⟩ ≡

```
if length(df_list) = 0 ∧ length(linear_functions_list) = 0 then
  ≪ if atom(solvable_kernel := relation_analysis(!*ff2a(equation, denominator), bracketname)) then
      ≪ write ces!*, "(", n, ")␣is␣a␣non-solvable␣Lie␣relation"; terpri!* t ≫
    else
      ≪ write ces!*, "(", n, ")␣solved␣for␣"; maprin solvable_kernel; terpri!* t;
        nullify_equation(n) ≫;
    goto solved ≫
```

This code is used in section 37.

39. Solving a function. If *equation* is not a Lie expression, there may be a function or a constant for which we can solve it. In order to do this we need to

- find all variables *present_variables*, on which at least one of the present functions *recursive_functions_list* depends; of course it is the complement of *absent_variables* in *variable_list*.
- find all linearly occuring functions *solvable_kernels* which depend on all of the *present_variables*; these are the possible candidates for solving. If there are no *present_variables*, *equation* is apparently a relation between some constants and we can try to solve one.
- remove all functions from *solvable_kernels*, which also occur in a *df* term, or in the liebracket case, in a commutator.
- if *coefficient_check* is **on** we must only solve for those functions which have a number as coefficient. This is checked by the procedure *find_solvable_kernel*.

Before doing anything we shall, however, construct lists containing all functions occuring in *df* terms, occuring linearly (and the constants) and, if necessary, occuring in commutators. These lists will also come in handy in the next steps.

```
⟨ Step 4: search for a solvable function 39 ⟩ ≡
  ⟨ Construct df_terms, df_functions, linear_functions and commutator_functions 40 ⟩;
  ⟨ Get present_variables and nr_of_variables 41 ⟩;
  for each kernel in linear_functions do
    if length get_dependencies_of(kernel) = nr_of_variables then
      solvable_kernels := kernel . solvable_kernels;
  for each kernel in append(df_functions, commutator_functions) do
    solvable_kernels := delete(kernel, solvable_kernels);
  if solvable_kernels then ⟨ Try to solve a function 42 ⟩
```

This code is used in section 15.

40. Of course we are only interested in *df* terms of functions occuring on *function_list*.

```
⟨ Construct df_terms, df_functions, linear_functions and commutator_functions 40 ⟩ ≡
  df_terms := for each df_term in df_list join
      if member(operator_name_of first_argument_of kernel_of df_term, function_list) then
        list kernel_of df_term;
  for each df_term in df_terms do
    if ¬member(first_argument_of df_term, df_functions) then
      df_functions := first_argument_of(df_term) . df_functions;
  functions_and_constants_list := append(linear_functions_list, kc_list_of constants_list);
  linear_functions := for each linear_function in functions_and_constants_list collect
          kernel_of linear_function;
  if bracketname then commutator_functions :=
        get_recursive_kernels(independent_part_of constants_list, get(ces!*, 'function_list));
```

This code is used in section 39.

41.

```
⟨ Get present_variables and nr_of_variables 41 ⟩ ≡
  present_variables := variable_list;
  for each variable in absent_variables do present_variables := delete(variable, present_variables);
  nr_of_variables := length present_variables
```

This code is used in section 39.

42. ⟨ Try to solve a function 42 ⟩ ≡

```
≪solvable_kernel := find_solvable_kernel(solvable_kernels, functions_and_constants_list, denominator);
  if solvable_kernel then
    ≪linear_solve_and_assign(!*ff2a(equation, 1), solvable_kernel);
      depl!* := assoc_delete(solvable_kernel, depl!*);
          { Remove the dependencies of the solved function }
      successful_message_for(n, "Solved␣for␣", solvable_kernel); goto solved ≫
  else
    ≪not_a_number_message_for(n, "Solving␣a␣function", partial_list(solvable_kernels, 3));
      goto solved ≫≫
```

This code is used in section 39.

43. Inhomogeneous integration. For an inhomogeneous integration, we are looking for a maximal *df* term, i.e. which has dependency on all the *present_variables*, such that the remaining part of *equation* is polynomial in the variables, w.r.t. which the function in the *df* term is differentiated, i.e. *a*) we only have to look at *df* terms which are differentiated w.r.t. variables on which none of the non-maximally occuring functions in *equation* depend, and *b*) if *polynomial_check* is **on** , we must check explicitly if the rest of *equation* is polynomial in these variables.

We shall collect the list of "integrable" variables in the list *integration_variables*.

```
⟨ Step 5: search for inhomogeneous integration 43 ⟩ ≡
  ⟨ Find the possible integration_variables 44 ⟩;
  if try_an_inhomogeneous_integration(n, equation, denominator, df_list, df_terms, integration_variables,
        nr_of_variables) then goto solved
```

This code is used in section 15.

44. Finding the *integration_variables* is rather easy using the lists *df_functions*, *linear_functions* and *commutator_functions*. Starting with *present_variables* we have to delete all variables on which on of the *linear_functions* or *commutator_functions* depend, or one of the *df_functions*, which do not have maximal dependency, i.e. which do no depend on *nr_of_variables* variables.

```
⟨ Find the possible integration_variables 44 ⟩ ≡
  integration_variables := present_variables;
  for each kernel in append(linear_functions, commutator_functions) do
    for each variable in get_dependencies_of(kernel) do
      integration_variables := delete(variable, integration_variables);
  for each df_function in df_functions do
    if ¬length get_dependencies_of(df_function) = nr_of_variables then
      for each variable in get_dependencies_of(df_function) do
        integration_variables := delete(variable, integration_variables)
```

This code is used in section 43.

45. Finding the integrable *df* terms is rather easy know: find all the *df* terms which have maximal dependency and are only differentiated w.r.t. variables occuring on *integration_variables*. In order to check this last item we need to know the form of *df* term: it is a list '(*df* function differentiation_sequence), where differentiation_sequence is a sequence of variables, each variable optionally followed by a integer indicating the number of differentiations w.r.t. to that variable. The procedure *check_differentiation_sequence* checks whether all variables in a differentiation_sequence are member of the second argument *variable_list*.

```
lisp procedure check_differentiation_sequence(sequence, variable_list);
  if null sequence then t
  else if fixp first_element_of sequence ∨ member(first_element_of sequence, variable_list) then
      check_differentiation_sequence(rest_of sequence, variable_list)$
```

46.

```
lisp procedure try_an_inhomogeneous_integration(n, equation, denominator, df_list, df_terms,
      integration_variables, nr_of_variables);
  begin scalar solvable_kernel, solvable_kernels, forbidden_functions,
      df_kernel, inhomogeneous_term;
  ⟨ Find the integrable df_terms 47 ⟩;
  ⟨ Find a solvable_kernel, check the inhomogeneous_term and possibly integrate 48 ⟩
  end$
```

47. There one situation we have to take care of specifically: if there are more *df_terms* for the same function, only one of which is differentiated just w.r.t. *integration_variables*, we are not allowed to integrate, since the function would be expressed in itself. In this case, we will make *solvable_kernels* a list of at least length 2 in order to prevent integration.

```
⟨ Find the integrable df_terms 47 ⟩ ≡
  for each df_term in df_terms do
    ≪ if length get_dependencies_of(first_argument_of df_term) = nr_of_variables
          ∧ (check_differentiation_sequence(rest_of arguments_of df_term, integration_variables)
          ∨ member(first_argument_of df_term, forbidden_functions)) then
        solvable_kernels := if member(first_argument_of df_term, forbidden_functions) then list(nil, nil)
          else df_term . solvable_kernels;
      forbidden_functions := (first_argument_of df_term) . forbidden_functions ≫;
```

This code is used in section 46.

48. ⟨ Find a *solvable_kernel*, check the *inhomogeneous_term* and possibly integrate 48 ⟩ ≡

```
  return
    if solvable_kernels then
      if length(solvable_kernels) = 1 then
        if (solvable_kernel := find_solvable_kernel(solvable_kernels, df_list, denominator)) then
          if (inhomogeneous_term := linear_solve(mk!*sq(equation ./ 1), solvable_kernel))
                ∧ (¬!*polynomial_check
                ∨ check_polynomial_integration(solvable_kernel, inhomogeneous_term)) then
            ≪ df_kernel := first_argument_of solvable_kernel;
              setk(df_kernel, inhomogeneous_integration_of(solvable_kernel, inhomogeneous_term));
              depl!* := assoc_delete(df_kernel, depl!*);    { Remove df_kernel from the depl!* list }
              successful_message_for(n, "Inhomogeneous integration of ", solvable_kernel) ≫
          else
            ≪ write ces!*, "(", n, "): Inhomogeneous integration failed: "; terpri!* t;
              write "inhomogeneous term not polynomial in integration variables"; terpri!* t;
              t ≫
        else not_a_number_message_for(n, "Inhomogeneous integration",
              first_element_of solvable_kernels)
      else
        ≪ write ces!*, "(", n, "): Inhomogeneous integration failed: "; terpri!* t;
          write "more terms with maximal dependency"; terpri!* t; t ≫
```

This code is used in section 46.

49. Checking that the inhomogeneous term is polynomial in the integration variables is fairly easy. For all the integration variables we have to check that the denominator does not depend on it and the numerator should be polynomial.

```
lisp procedure check_polynomial_integration(df_term, integration_term);
  begin scalar numerator, denominator, integration_variables, variable, ok;
  numerator := numr simp integration_term; denominator := denr simp integration_term;
  integration_variables := for each argument in rest_of arguments_of df_term join
      if ¬fixp argument then list argument;
  ok := t;
  while ok ∧ integration_variables do
    ≪ variable := first_element_of integration_variables;
      ok := (¬depends(denominator, variable) ∧ polynomialp(numerator, variable));
      integration_variables := rest_of integration_variables ≫;
  return ok;
  end$
```

50. We can perform the inhomogeneous integration by applying *multi_split_form* to find all the polynomial components of the inhomogeneous term and *homogeneous_integration_of* for solving the homogeneous equation.

```
lisp procedure inhomogeneous_integration_of(df_term, inhomogeneous_term);
  begin scalar df_sequence, integration_variables, int_sequence, variable, nr_of_integrations,
      integration_terms, solution, powers, coefficient, int_factor, solution_term, n, k;
  df_sequence := rest_of arguments_of df_term;
  ⟨ Find the integration_variables and int_sequence 51 ⟩;
  integration_terms := multi_split_form(numr simp inhomogeneous_term, integration_variables);
  integration_terms := (nil . independent_part_of integration_terms) . pc_list_of integration_terms;
      { Make integration_terms a full blown pc_list }
  ⟨ Perform the inhomogeneous integration of the numerator of inhomogeneous_term 52 ⟩;
  solution := multsq(solution, 1 ./ denr simp inhomogeneous_term);
  solution := mk!*sq subs2 addsq(solution, simp homogeneous_integration_of df_term);
  return solution
  end$
```

51. We must analyze *df_sequence* to get all the integration variables, together with the number of integrations belonging to them.

```
⟨ Find the integration_variables and int_sequence 51 ⟩ ≡
  while df_sequence do
    ≪ variable := first_element_of df_sequence; df_sequence := rest_of df_sequence;
      if df_sequence ∧ fixp first_element_of df_sequence then
        ≪ nr_of_integrations := first_element_of df_sequence; df_sequence := rest_of df_sequence ≫
      else nr_of_integrations := 1;
      integration_variables := variable . integration_variables;
      int_sequence := (variable . nr_of_integrations) . int_sequence ≫
```

This code is used in section 50.

52. The particular solution of the equation $F^{(k)}(x) = x^n$ is

$$F(x) = \frac{1}{(n+1)\cdots(n+k)} x^{n+k}.$$

This process has to be performed for all the terms in *integration_terms* and for all integrations in *int_sequence*.

```
⟨ Perform the inhomogeneous integration of the numerator of inhomogeneous_term 52 ⟩ ≡
  solution := nil ./ 1;
  for each term in integration_terms do
    ≪ powers := powers_of term; coefficient := coefficient_of term; int_factor := 1;
      solution_term := 1 ./ 1;
      for each integration in int_sequence do
        ≪ variable := car integration; k := cdr integration;
          n := (if power then cdr power else 0) where power = assoc(variable, powers);
                { If variable does not occur in term, n = 0 }
          for i := 1:k do int_factor := (n + i)*int_factor;
          solution_term := multsq(solution_term, mksq(variable, n + k)) ≫;
      solution_term := multsq(solution_term, coefficient ./ int_factor);
      solution := addsq(solution, solution_term) ≫
```

This code is used in section 50.

53. Generation of new equations by differentiation. As a last method of solving we notice the following: if there is a variable, such that just one *df* term or just one linearly occuring function depends on it and all the other terms are polynomial in this variable, let's say of degree n, then we can differentiate *equation* $n+1$ times to get a new equation of type A.

Experience has proven, however, that applying the above mentioned method, generally will lead to multiple generation of equivalent terms in the answer. Therefore we will only generate a new equation if the switch *allow_differentiation* is **on** , otherwise we will only generate a message that it is possible to generate a new equation of type A. Solving of such a new equation is always left to the responsibility of the user.

⟨ Lisp initializations 9 ⟩ +≡
new_switch(*allow_differentiation*, **nil**)$

54. After this introduction it is clear what we have to do for step 6:

⟨ Step 6: search for a useful differentation 54 ⟩ ≡
if *try_a_differentiation*(*n*, *total_used*, *equation*, *present_variables*, *df_terms*, *linear_functions*, *commutator_functions*) **then goto** *solved*

This code is used in section 15.

55.

lisp procedure *try_a_differentiation*(*n*, *total_used*, *equation*, *present_variables*, *df_terms*, *linear_functions*, *commutator_functions*);
begin scalar *differentiations_list*, *polynomial_order*;
⟨ Count the number of occurences of all *present_variables* 56 ⟩;
⟨ If possible and allowed, generate new equations 57 ⟩
end$

56. Counting the occurence of variables is rather easy. For all functions in *df_terms*, *linear_functions* and *commutator_functions*, we have to count the occurences of all the variables in their respective entries on the dependency list *depl!**.

For this purpose we rebuild *present_variables* to an association list with entries of the form *variable* . *origin* . *number_of_occurences* where *origin* indicates the *df_term*, *linear_function* or *commutator_function* in which *variable* occured last.

The action of the following macros, which harmlessly make use of the procedure *rplacd*, is clear.

```
define reinitialize_present_variables ≡
        present_variables := for each variable in present_variables collect (variable . nil . 0)
define variable_of ≡ car
define origin_of ≡ cadr
define counter_of ≡ cddr
define update_variable(variable, origin) ≡
        rplacd(entry, origin . (counter_of entry + 1)) where entry = assoc(variable, present_variables)
define update_variables_using(kernel_list, kernel_selector, flag_function) ≡
        for each kernel in kernel_list do
          for each variable in get_dependencies_of(kernel_selector(kernel)) do
            update_variable(variable, flag_function(kernel));
define identity_function(kernel) ≡ kernel
define empty_function(kernel) ≡ nil
```

⟨ Count the number of occurences of all *present_variables* 56 ⟩ ≡

```
reinitialize_present_variables;
update_variables_using(df_terms, first_argument_of, identity_function);
update_variables_using(linear_functions, identity_function, identity_function);
update_variables_using(commutator_functions, identity_function, empty_function)
```

This code is used in section 55.

57. After the preceding step we can generate new equations by differentiating *equation* w.r.t. to all those variables which occur in only one *df_term* or *linear_function* and for which all other terms of *equation* are polynomial. Using the above code one can check that these variables are exactly the ones for which the *origin* has a value and the *counter* is 1.

```
⟨ If possible and allowed, generate new equations 57 ⟩ ≡
  differentiations_list := for each entry in present_variables join
      if origin_of entry ∧ counter_of entry = 1
            ∧ (polynomial_order :=
            get_polynomial_order(linear_solve(mk!*sq(equation ./ 1), origin_of entry), variable_of entry))
            then list(variable_of entry . origin_of entry . (polynomial_order + 1));
  return
    if differentiations_list then
      if !*allow_differentiation then
        ≪for each entry in differentiations_list do
            setk(list(ces!*, incr(total_used)),
                 mk!*sq simpdf list(mk!*sq(equation ./ 1), variable_of entry, counter_of entry));
          write ces!*, "(", n, "):␣Generation␣of␣", ces!*, "(", get(ces!*, 'total_used) + 1, "),...,",
              ces!*, "(", total_used, ")␣by␣differentiation␣w.r.t.␣"; terpri!* t;
          maprin partial_list(for each entry in differentiations_list collect
                list('list, variable_of entry, counter_of entry), 10);
          terpri!*(¬!*nat); put(ces!*, 'total_used, total_used); t≫
      else
        ≪write "***␣", ces!*, "(", n,
              "):␣Generation␣of␣new␣equations␣by␣differentiation␣possible."; terpri!* t;
          write "␣␣␣␣Solvable␣with␣'on␣allow_differentiation'"; terpri!* t; t≫
```

This code is used in section 55.

58. An algebraic expression is polynomial in a variable if the denominator does not depend on it and if the numerator is polynomial (we only have to check this if *polynomial_check* is **on**). The polynomial order we can obtain by simply reordering the numerator w.r.t. the variable involved.

```
lisp procedure get_polynomial_order(expression, variable);
  if ¬depends(denr(expression := simp expression), variable)
        ∧ (¬!*polynomial_check ∨ polynomialp(numr expression, variable)) then
    begin scalar kord!*;
    setkorder list !*a2k variable; expression := reorder numr expression;
    return if mvar expression = variable then ldeg expression else 0;
    end$
```

59. If none of the above methods can be applied, we cannot solve *equation*.

```
⟨ Step 7: print a "Not solved" message 59 ⟩ ≡
  write ces!*, "(", n, ")␣not␣solved"; terpri!* t
```

This code is used in section 15.

60. Additional tools. The following procedures are meant for solving more equations at a time or solving "exceptional" equations, which need the least restrictive setting of the switches *coefficient_check*, *polynomial_check* or *allow_differentiation*.

```
algebraic procedure integrate_equations(m, n);
  for i := m:n do integrate_equation(i)$

lisp operator integrate_exceptional_equation;
lisp procedure integrate_exceptional_equation(n);
  integrate_equation(n) where
      !*coefficient_check = nil,
      !*polynomial_check = nil,
      !*allow_differentiation = t$
```

61. For a system of equations which is not too difficult it may be possible to solve the system without intervenience of the user. For such systems the procedure *auto_solve* tries to solve a system automatically. If successful, it returns a message saying so, otherwise it returns the list of equation numbers left unsolved. The parameter *nr_list* gives either the equation number or the list of equation numbers to be considered: the other equations may contain conditions which should only be considered when all the higher equations are solved, for instance when we solve using some kind of grading and solve the system degree by degree.

```
lisp operator auto_solve;
lisp procedure auto_solve nr_list;
  begin scalar total, old_total, to_do, unsolved, old_unsolved, stuck;
  total := old_total := get(ces!*, 'total_used);
  to_do := if fixp nr_list then list nr_list
    else if car nr_list = 'list then cdr nr_list
      else nr_list;
  while ¬stuck ∧ to_do do
    begin
    for each eq_nr in to_do do
      ≪integrate_equation eq_nr;
        if cadr assoc(list(ces!*, eq_nr), get(ces!*, 'kvalue)) ≠ 0 then unsolved := eq_nr . unsolved ≫;
    total := get(ces!*, 'total_used);
    if total = old_total ∧ unsolved ∧ unsolved = old_unsolved then stuck := t
    else ≪old_unsolved := unsolved; to_do := reverse unsolved; unsolved := nil;
          to_do := append(for eq_nr := old_total + 1:total collect eq_nr, to_do); old_total := total ≫
    end;
  if stuck then return 'list . reverse unsolved
  else ≪terpri(); write "Successful integration of all equations"; terpri()≫;
  end$
```

62. As a last set of tools, we shall give a procedure to print an equation together with all the functions occuring in it and their dependencies, and some procedures for showing and changing the properties of an equation set and a the functions/constants used.

As a side effect the procedure *show_equation* will reassign the shown equation to its current value.

```
lisp operator show_equation;
lisp procedure show_equation n;
  begin scalar equation, total_used, function_list;
  if null(total_used := get(ces!*, 'total_used)) ∨ n > total_used then
        stop_with_error("SHOW_EQUATION: properly initialize", ces!*, nil, nil);
  if (equation := assoc(list(ces!*, n), get(ces!*, 'kvalue))) then
    begin equation := setk(list(ces!*, n), aeval cadr equation);
    varpri(equation, list('setk, mkquote list(ces!*, n), mkquote equation), 'only);
    function_list := get_recursive_kernels(numr simp equation, get(ces!*, 'function_list));
    if function_list then
      ≪ terpri!* t; write "Functions occurring:"; terpri!* t;
        for each fn in function_list do
          ≪ maprin(fn . get_dependencies_of(fn)); terpri!*(¬!*nat) ≫ ≫
    else terpri!* nil
    end
  end$
algebraic procedure show_equations(m, n);
  for i := m:n do show_equation i$
```

63.

```
lisp operator functions_used, put_functions_used, equations_used, put_equations_used;
lisp procedure functions_used function_name;
  list('list, get(function_name, 'even_used), get(function_name, 'odd_used))$
lisp procedure put_functions_used(function_name, even_used, odd_used);
  begin
  if ¬fixp even_used ∨ even_used < 0 ∨ ¬fixp odd_used ∨ odd_used < 0 then
    stop_with_error("PUT_FUNCTIONS_USED: used functions number invalid", nil, nil, nil);
  put(function_name, 'even_used, even_used); put(function_name, 'odd_used, odd_used);
  end$
lisp procedure equations_used;
  get(ces!*, 'total_used)$
lisp procedure put_equations_used(n);
  if ¬fixp n ∨ n < 0 then
    stop_with_error("PUT_EQUATIONS_USED: used equation number invalid", nil, nil, nil)
  else put(ces!*, 'total_used, n)$
```

64. There is one slight detail which we have not dealt with yet: in prolongation theory differentiation should act as a derivation on the arguments of a (eventually nested) commutator. In REDUCE 3.4 there is a hook which can take care of this situation. In the procedure *diffp*, which takes care of differentiation of standard powers, if this standard power is an operator kernel, the property *dfform* is checked for operator concerned. If this property has a value, it should be a function which takes care of the differentiation of such a standard power.

```
lisp operator df_acts_as_derivation_on;
lisp procedure df_acts_as_derivation_on operator_name;
  begin put(operator_name,'dfform,'df_as_derivation);
  end$
```

65. The procedure *df_as_derivation* is quite straightforward: apply *df* to all the arguments of the operator, one at a time, leaving the other ones untouched.

```
lisp procedure df_as_derivation(kernel, variable, power);
  begin scalar left_part, right_part, argument, derivative;
  if power ≠ 1 then stop_with_error("DF_AS_DERIVATION:", kernel, "must␣occur␣linearly", nil);
  left_part := list operator_name_of kernel;
  right_part := arguments_of kernel;
  derivative := nil . 1;
  while right_part do
    ≪ argument := first_element_of right_part;
      right_part := rest_of right_part;
      derivative := addsq(derivative, simp append(reverse left_part, list('df, argument,
          variable) . right_part));
      left_part := argument . left_part;≫;
  return derivative;
  end$
```

66. In order to get nice output of some of the messages given by *integrate_equation* we redefine the print function *listpri* for algebraic lists. Namely, we want don't want algebraic lists to split over multiple lines in the messages we give. For this purpose, we introduce a fluid variable *listpri_depth!** which governs the depth for which algebraic lists are split along lines. The default value is the same as the value in the used in REDUCE.

```
⟨ Lisp initializations 9 ⟩ + ≡
  initialize_fluid(listpri_depth!*, 40)$
```

67. The following procedure can be used at algebraic level to change *listpri_depth!**.

```
lisp operator listlength$
lisp procedure listlength l;
  listpri_depth!* := l$
```

68. The definition of *listpri* is basically that of *inprint*, except that it decides when to split at the comma by looking at the size of the argument, using the global variable *listpri_depth!**.

```
symbolic procedure listpri l;
  begin scalar orig, split, u;
  u := l; l := cdr l; prin2!* get('!*lcbkt!*, 'prtch);    { Do it this way so table can change }
  orig := orig!*;
  orig!* := if posn!* < 18 then posn!* else orig!* + 3;
  if null l then go to b;
  split := treesizep(l, listpri_depth!*);
a: maprint(negnumberchk car l, 0); l := cdr l;
  if null l then go to b;
  oprin '!*comma!*;
  if split then terpri!* t;
  go to a;
b: prin2!* get('!*rcbkt!*, 'prtch); orig!* := orig;
  return u
  end$
```

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

```
end;
```

70. 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.