REDUCE

User's and Contributed Packages Manual

Version 3.7

Anthony C. Hearn
Santa Monica, CA
and Codemist Ltd.

Email: reduce@rand.org

February 1999

Preface

This volume has been prepared by Codemist Ltd. from the LaTeX documentation sources distributed with REDUCE 3.7. It incorporates the User's Manual, and documentation for all the User Contributed Packages as a second Part. A common index and table of contents has been prepared. We hope that this single volume will be more convenient for REDUCE users than having two unrelated documents. Particularly in Part 2 the text of the authors has been extensively edited and modified and so the responsibility for any errors rests with us.

Parts I and III were written by Anthony C. Hearn. Part II is based on texts by:
Werner Antweiler, Victor Adamchik, Joachim Apel, Alan Barnes, Andreas Bernig, Yu. A. Blinkov, Russell Bradford, Chris Cannam, Hubert Caprasse, C. Dicrescenzo, Alain Dresse, Ladislav Drska, James W. Eastwood, John Fitch, Kerry Gaskell, Barbara L. Gates, Karin Gatermann, Hans-Gert Gräbe, David Harper, David Hartley, Anthony C. Hearn, J. A. van Hulzen, V. Ilyin, Stanley L. Kameny, Fujio Kako, C. Kazasov, Wolfram Koepf, A. Kryukov, Richard Liska, Kevin McIsaac, Malcolm A. H. MacCallum, Herbert Melenk, H. M. Möller, Winfried Neun, Julian Padget, Matt Rebbeck, F. Richard-Jung, A. Rodionov, Carsten and Franziska Schöbel, Rainer Schöpf, Stephen Scowcroft, Eberhard Schrüfer, Fritz Schwarz, M. Spiridonova, A. Taranov, Lisa Temme, Walter Tietze, V. Tomov, E. Tournier, Philip A. Tuckey, G. Üçoluk, Mathias Warns, Thomas Wolf, Francis J. Wright and A. Yu. Zharkov.

February 1999

Codemist Ltd
"Alta", Horsecombe Vale
Combe Down
Bath, England

Contents

Part I

REDUCE User's Manual

REDUCE

User's Manual

Version 3.7

Anthony C. Hearn
Santa Monica, CA, USA

Email: reduce@rand.org

March 1999

Abstract

This document provides the user with a description of the algebraic programming system REDUCE. The capabilities of this system include:

1. expansion and ordering of polynomials and rational functions,
2. substitutions and pattern matching in a wide variety of forms,
3. automatic and user controlled simplification of expressions,
4. calculations with symbolic matrices,
5. arbitrary precision integer and real arithmetic,
6. facilities for defining new functions and extending program syntax,
7. analytic differentiation and integration,
8. factorization of polynomials,
9. facilities for the solution of a variety of algebraic equations,
10. facilities for the output of expressions in a variety of formats,
11. facilities for generating numerical programs from symbolic input,
12. Dirac matrix calculations of interest to high energy physicists.

Acknowledgment

The production of this version of the manual has been the result of the contributions of a large number of individuals who have taken the time and effort to suggest improvements to previous versions, and to draft new sections. Particular thanks are due to Gerry Rayna, who provided a draft rewrite of most of the first half of the manual. Other people who have made significant contributions have included John Fitch, Martin Griss, Stan Kameny, Jed Marti, Herbert Melenk, Don Morrison, Arthur Norman, Eberhard Schrüfer, Larry Seward and Walter Tietze. Finally, Richard Hitt produced a TEX version of the REDUCE 3.3 manual, which has been a useful guide for the production of the LATEX version of this manual.

Chapter 1

Introductory Information

REDUCE is a system for carrying out algebraic operations accurately, no matter how complicated the expressions become. It can manipulate polynomials in a variety of forms, both expanding and factoring them, and extract various parts of them as required. REDUCE can also do differentiation and integration, but we shall only show trivial examples of this in this introduction. Other topics not considered include the use of arrays, the definition of procedures and operators, the specific routines for high energy physics calculations, the use of files to eliminate repetitious typing and for saving results, and the editing of the input text.

Also not considered in any detail in this introduction are the many options that are available for varying computational procedures, output forms, number systems used, and so on.

REDUCE is designed to be an interactive system, so that the user can input an algebraic expression and see its value before moving on to the next calculation. For those systems that do not support interactive use, or for those calculations, especially long ones, for which a standard script can be defined, REDUCE can also be used in batch mode. In this case, a sequence of commands can be given to REDUCE and results obtained without any user interaction during the computation.

In this introduction, we shall limit ourselves to the interactive use of REDUCE, since this illustrates most completely the capabilities of the system. When REDUCE is called, it begins by printing a banner message like:

```
REDUCE 3.7, 15-Jan-99 ...
```

where the version number and the system release date will change from time to time. It then prompts the user for input by:

```
1:
```

You can now type a REDUCE statement, terminated by a semicolon to indicate the end of the expression, for example:

```
(x+y+z)^2;
```

This expression would normally be followed by another character (a Return on an ASCII keyboard) to "wake up" the system, which would then input the expression, evaluate it, and return the result:

```
 2                    2            2
X  + 2*X*Y + 2*X*Z + Y  + 2*Y*Z + Z
```

Let us review this simple example to learn a little more about the way that REDUCE works. First, we note that REDUCE deals with variables, and constants like other computer languages, but that in evaluating the former, a variable can stand for itself. Expression evaluation normally follows the rules of high school algebra, so the only surprise in the above example might be that the expression was expanded. REDUCE normally expands expressions where possible, collecting like terms and ordering the variables in a specific manner. However, expansion, ordering of variables, format of output and so on is under control of the user, and various declarations are available to manipulate these.

Another characteristic of the above example is the use of lower case on input and upper case on output. In fact, input may be in either mode, but output is usually in lower case. To make the difference between input and output more distinct in this manual, all expressions intended for input will be shown in lower case and output in upper case. However, for stylistic reasons, we represent all single identifiers in the text in upper case.

Finally, the numerical prompt can be used to reference the result in a later computation.

As a further illustration of the system features, the user should try:

```
for i:= 1:40 product i;
```

The result in this case is the value of 40!,

```
8159152832478977343456112695961158942720000000000
```

You can also get the same result by saying

```
factorial 40;
```

Since we want exact results in algebraic calculations, it is essential that integer arithmetic be performed to arbitrary precision, as in the above example. Furthermore, the `FOR` statement in the above is illustrative of a whole range of combining forms that REDUCE supports for the convenience of the user.

Among the many options in REDUCE is the use of other number systems, such as multiple precision floating point with any specified number of digits — of use if roundoff in, say, the 100^{th} digit is all that can be tolerated.

In many cases, it is necessary to use the results of one calculation in succeeding calculations. One way to do this is via an assignment for a variable, such as

```
u := (x+y+z)^2;
```

If we now use `U` in later calculations, the value of the right-hand side of the above will be used.

The results of a given calculation are also saved in the variable `WS` (for WorkSpace), so this can be used in the next calculation for further processing.

For example, the expression

```
df(ws,x);
```

following the previous evaluation will calculate the derivative of `(x+y+z)^2` with respect to `X`. Alternatively,

```
int(ws,y);
```

would calculate the integral of the same expression with respect to y.

REDUCE is also capable of handling symbolic matrices. For example,

```
matrix m(2,2);
```

declares m to be a two by two matrix, and

```
m := mat((a,b),(c,d));
```

gives its elements values. Expressions that include `M` and make algebraic sense may now be evaluated, such as `1/m` to give the inverse, `2*m - u*m^2` to give us another matrix and `det(m)` to give us the determinant of `M`.

REDUCE has a wide range of substitution capabilities. The system knows about elementary functions, but does not automatically invoke many of their well-known properties. For example, products of trigonometrical functions are not converted automatically into multiple angle expressions, but if the user wants this, he can say, for example:

```
(sin(a+b)+cos(a+b))*(sin(a-b)-cos(a-b))
    where cos(~x)*cos(~y) = (cos(x+y)+cos(x-y))/2,
          cos(~x)*sin(~y) = (sin(x+y)-sin(x-y))/2,
          sin(~x)*sin(~y) = (cos(x-y)-cos(x+y))/2;
```

where the tilde in front of the variables `X` and `Y` indicates that the rules apply for all values of those variables. The result of this calculation is

```
-(COS(2*A) + SIN(2*B))
```

See also the user-contributed packages ASSIST (chapter 23), CAMAL (chapter 28) and TRIGSIMP (chapter 85).

Another very commonly used capability of the system, and an illustration of one of the many output modes of REDUCE, is the ability to output results in a FORTRAN compatible form. Such results can then be used in a FORTRAN based numerical calculation. This is particularly useful as a way of generating algebraic formulas to be used as the basis of extensive numerical calculations.

For example, the statements

```
on fort;
df(log(x)*(sin(x)+cos(x))/sqrt(x),x,2);
```

will result in the output

```
ANS=(-4.*LOG(X)*COS(X)*X**2-4.*LOG(X)*COS(X)*X+3.*
. LOG(X)*COS(X)-4.*LOG(X)*SIN(X)*X**2+4.*LOG(X)*
. SIN(X)*X+3.*LOG(X)*SIN(X)+8.*COS(X)*X-8.*COS(X)-8.
. *SIN(X)*X-8.*SIN(X))/(4.*SQRT(X)*X**2)
```

These algebraic manipulations illustrate the algebraic mode of REDUCE. REDUCE is based on Standard Lisp. A symbolic mode is also available for executing Lisp statements. These statements follow the syntax of Lisp, e.g.

```
symbolic car '(a);
```

Communication between the two modes is possible.

With this simple introduction, you are now in a position to study the material in the full REDUCE manual in order to learn just how extensive the range of facilities really is. If further tutorial material is desired, the seven REDUCE Interactive Lessons by David R. Stoutemyer are recommended. These are normally distributed with the system.

Chapter 2

Structure of Programs

A REDUCE program consists of a set of functional commands which are evaluated sequentially by the computer. These commands are built up from declarations, statements and expressions. Such entities are composed of sequences of numbers, variables, operators, strings, reserved words and delimiters (such as commas and parentheses), which in turn are sequences of basic characters.

2.1 The REDUCE Standard Character Set

The basic characters which are used to build REDUCE symbols are the following:

1. The 26 letters `a` through `z`

2. The 10 decimal digits `0` through `9`

3. The special characters _ ! ” $ % ’ ( ) * + , - . / : ; < > = { } <blank>

With the exception of strings and characters preceded by an exclamation mark, the case of characters is ignored: depending of the underlying LISP they will all be converted internally into lower case or upper case: `ALPHA`, `Alpha` and `alpha` represent the same symbol. Most implementations allow you to switch this conversion off. The operating instructions for a particular implementation should be consulted on this point. For portability, we shall limit ourselves to the standard character set in this exposition.

2.2 Numbers

There are several different types of numbers available in REDUCE. Integers consist of a signed or unsigned sequence of decimal digits written without a decimal point, for example:

```
-2, 5396, +32
```

In principle, there is no practical limit on the number of digits permitted as exact arithmetic is used in most implementations. (You should however check the specific instructions for your particular system implementation to make sure that this is true.) For example, if you ask for the value of 2^{2000} you get it displayed as a number of 603 decimal digits, taking up nine lines of output on an interactive display. It should be borne in mind of course that computations with such long numbers can be quite slow.

Numbers that aren't integers are usually represented as the quotient of two integers, in lowest terms: that is, as rational numbers.

In essentially all versions of REDUCE it is also possible (but not always desirable!) to ask REDUCE to work with floating point approximations to numbers again, to any precision. Such numbers are called *real*. They can be input in two ways:

1. as a signed or unsigned sequence of any number of decimal digits with an embedded or trailing decimal point.
2. as in 1. followed by a decimal exponent which is written as the letter `E` followed by a signed or unsigned integer.

e.g. `32.` `+32.0` `0.32E2` and `320.E-1` are all representations of 32.

The declaration `SCIENTIFIC_NOTATION` controls the output format of floating point numbers. At the default settings, any number with five or less digits before the decimal point is printed in a fixed-point notation, e.g., `12345.6`. Numbers with more than five digits are printed in scientific notation, e.g., `1.234567E+5`. Similarly, by default, any number with eleven or more zeros after the decimal point is printed in scientific notation. To change these defaults, `SCIENTIFIC_NOTATION` can be used in one of two ways. `SCIENTIFIC_NOTATION` m;, where m is a positive integer, sets the printing format so that a number with more than m digits before the decimal point, or m or more zeros after the decimal point, is printed in scientific notation.

`SCIENTIFIC_NOTATION` $\{m,n\}$, with m and n both positive integers, sets the format so that a number with more than m digits before the decimal point, or n or more zeros after the decimal point is printed in scientific notation.

CAUTION: The unsigned part of any number may *not* begin with a decimal point, as this causes confusion with the `CONS` (.) operator, i.e., NOT ALLOWED: `.5 -.23 +.12`; use `0.5 -0.23 +0.12` instead.

2.3 Identifiers

Identifiers in REDUCE consist of one or more alphanumeric characters (i.e. alphabetic letters or decimal digits) the first of which must be alphabetic. The maximum number of characters allowed is implementation dependent, although twenty-four is permitted in most implementations. In addition, the underscore character (_) is considered a letter if it is *within* an identifier. For example,

```
a az p1 q23p  a_very_long_variable
```

are all identifiers, whereas

```
_a
```

is not.

A sequence of alphanumeric characters in which the first is a digit is interpreted as a product. For example, `2ab3c` is interpreted as `2*ab3c`. There is one exception to this: If the first letter after a digit is `E`, the system will try to interpret that part of the sequence as a real number, which may fail in some cases. For example, `2E12` is the real number $2.0 * 10^{12}$, `2e3c` is 2000.0*C, and `2ebc` gives an error.

Special characters, such as $-$, *, and blank, may be used in identifiers too, even as the first character, but each must be preceded by an exclamation mark in input. For example:

```
light!-years    d!*!*n         good! morning
!$sign          !5goldrings
```

CAUTION: Many system identifiers have such special characters in their names (especially * and =). If the user accidentally picks the name of one of them for his own purposes it may have catastrophic consequences for his

REDUCE run. Users are therefore advised to avoid such names.

Identifiers are used as variables, labels and to name arrays, operators and procedures.

Restrictions

The reserved words listed in another section may not be used as identifiers. No spaces may appear within an identifier, and an identifier may not extend over a line of text. (Hyphenation of an identifier, by using a reserved character as a hyphen before an end-of-line character is possible in some versions of REDUCE).

2.4 Variables

Every variable is named by an identifier, and is given a specific type. The type is of no concern to the ordinary user. Most variables are allowed to have the default type, called *scalar*. These can receive, as values, the representation of any ordinary algebraic expression. In the absence of such a value, they stand for themselves.

Reserved Variables

Several variables in REDUCE have particular properties which should not be changed by the user. These variables include:

`E` Intended to represent the base of the natural logarithms. `log(e)`, if it occurs in an expression, is automatically replaced by 1. If `ROUNDED` is on, `E` is replaced by the value of E to the current degree of floating point precision.

`I` Intended to represent the square root of -1. `i^2` is replaced by -1, and appropriately for higher powers of `I`. This applies only to the symbol `I` used on the top level, not as a formal parameter in a procedure, a local variable, nor in the context `for i:= ...`

`INFINITY` Intended to represent ∞ in limit and power series calculations for example. Note however that the current system does *not* do proper arithmetic on ∞. For example, `infinity + infinity` is `2*infinity`.

`NIL` In REDUCE (algebraic mode only) taken as a synonym for zero. Therefore `NIL` cannot be used as a variable.

`PI` Intended to represent the circular constant. With `ROUNDED` on, it is replaced by the value of π to the current degree of floating point precision.

`T` Should not be used as a formal parameter or local variable in procedures, since conflict arises with the symbolic mode meaning of T as *true*.

Other reserved variables, such as `LOW_POW`, described in other sections, are listed in Appendix A.

Using these reserved variables inappropriately will lead to errors.

There are also internal variables used by REDUCE that have similar restrictions. These usually have an asterisk in their names, so it is unlikely a casual user would use one. An example of such a variable is `K!*` used in the asymptotic command package.

Certain words are reserved in REDUCE. They may only be used in the manner intended. A list of these is given in the section "Reserved Identifiers". There are, of course, an impossibly large number of such names to keep in mind. The reader may therefore want to make himself a copy of the list, deleting the names he doesn't think he is likely to use by mistake.

2.5 Strings

Strings are used in `WRITE` statements, in other output statements (such as error messages), and to name files. A string consists of any number of characters enclosed in double quotes. For example:

```
"A String".
```

Lower case characters within a string are not converted to upper case.

The string `""` represents the empty string. A double quote may be included in a string by preceding it by another double quote. Thus `"a""b"` is the string `a"b`, and `""""` is the string `"`.

2.6 Comments

Text can be included in program listings for the convenience of human readers, in such a way that REDUCE pays no attention to it. There are two ways to do this:

1. Everything from the word `COMMENT` to the next statement terminator, normally ; or \$, is ignored. Such comments can be placed anywhere a blank could properly appear. (Note that `END` and >> are *not* treated as `COMMENT` delimiters!)

2. Everything from the symbol `%` to the end of the line on which it appears is ignored. Such comments can be placed as the last part of any line. Statement terminators have no special meaning in such comments. Remember to put a semicolon before the `%` if the earlier part of the line is intended to be so terminated. Remember also to begin each line of a multi-line `%` comment with a `%` sign.

2.7 Operators

Operators in REDUCE are specified by name and type. There are two types, infix and prefix. Operators can be purely abstract, just symbols with no properties; they can have values assigned (using `:=` or simple `LET` declarations) for specific arguments; they can have properties declared for some collection of arguments (using more general `LET` declarations); or they can be fully defined (usually by a procedure declaration).

Infix operators have a definite precedence with respect to one another, and normally occur between their arguments. For example:

`a + b - c`	(spaces optional)
`x<y and y=z`	(spaces required where shown)

Spaces can be freely inserted between operators and variables or operators and operators. They are required only where operator names are spelled out with letters (such as the `AND` in the example) and must be unambiguously separated from another such or from a variable (like `Y`). Wherever one space can be used, so can any larger number.

Prefix operators occur to the left of their arguments, which are written as a list enclosed in parentheses and separated by commas, as with normal mathematical functions, e.g.,

```
cos(u)
df(x^2,x)
q(v+w)
```

Unmatched parentheses, incorrect groupings of infix operators and the like, naturally lead to syntax errors. The parentheses can be omitted (replaced by a space following the operator name) if the operator is unary and the argument is a single symbol or begins with a prefix operator name:

`cos y`	means cos(y)
`cos (-y)`	– parentheses necessary
`log cos y`	means log(cos(y))
`log cos (a+b)`	means log(cos(a+b))

but

`cos a*b`	means (cos a)*b
`cos -y`	is erroneous (treated as a variable "cos" minus the variable y)

A unary prefix operator has a precedence higher than any infix operator, including unary infix operators. In other words, REDUCE will always interpret `cos y + 3` as `(cos y) + 3` rather than as `cos(y + 3)`.

Infix operators may also be used in a prefix format on input, e.g., `+(a,b,c)`. On output, however, such expressions will always be printed in infix form (i.e., `a + b + c` for this example).

A number of prefix operators are built into the system with predefined properties. Users may also add new operators and define their rules for simplification. The built in operators are described in another section.

Built-In Infix Operators

The following infix operators are built into the system. They are all defined internally as procedures.

```
<infix operator>::= where|:=|or|and|member|memq|=|neq|eq|
                    >=|>|<=|<|+|-|*|/|^|**|.
```

These operators may be further divided into the following subclasses:

```
<assignment operator>   ::= :=
<logical operator>      ::= or|and|member|memq
<relational operator>   ::= =|neq|eq|>=|>|<=|<
<substitution operator> ::= where
<arithmetic operator>   ::= +|-|*|/|^|**
<construction operator> ::= .
```

`MEMQ` and `EQ` are not used in the algebraic mode of REDUCE. They are explained in the section on symbolic mode. `WHERE` is described in the section on substitutions.

In previous versions of REDUCE, *not* was also defined as an infix operator. In the present version it is a regular prefix operator, and interchangeable with *null*.

For compatibility with the intermediate language used by REDUCE, each special character infix operator has an alternative alphanumeric identifier associated with it. These identifiers may be used interchangeably with the corresponding special character names on input. This correspondence is as follows:

`:= setq`	(the assignment operator)
`= equal`	
`>= geq`	
`> greaterp`	
`<= leq`	
`< lessp`	
`+ plus`	
`- difference`	(if unary, `minus`)
`* times`	
`/ quotient`	(if unary, `recip`)
`^ or ** expt`	(raising to a power)
`. cons`	

Note: `NEQ` is used to mean *not equal.* There is no special symbol provided for it.

The above operators are binary, except `NOT` which is unary and `+` and `*` which are nary (i.e., taking an arbitrary number of arguments). In addition, `-` and `/` may be used as unary operators, e.g., /2 means the same as 1/2. Any other operator is parsed as a binary operator using a left association rule. Thus `a/b/c` is interpreted as `(a/b)/c`. There are two exceptions to this rule: `:=` and `.` are right associative. Example: `a:=b:=c` is interpreted as `a:=(b:=c)`. Unlike ALGOL and PASCAL, `^` is left associative. In other words, `a^b^c` is interpreted as `(a^b)^c`.

The operators $<$, $<$=, $>$, $>$= can only be used for making comparisons between numbers. No meaning is currently assigned to this kind of comparison between general expressions.

Parentheses may be used to specify the order of combination. If parentheses are omitted then this order is by the ordering of the precedence list defined by the right-hand side of the `<infix operator>` table at the beginning of this section, from lowest to highest. In other words, `WHERE` has the lowest precedence, and `.` (the dot operator) the highest.

Chapter 3

Expressions

REDUCE expressions may be of several types and consist of sequences of numbers, variables, operators, left and right parentheses and commas. The most common types are as follows:

3.1 Scalar Expressions

Using the arithmetic operations `+ - * / ^` (power) and parentheses, scalar expressions are composed from numbers, ordinary "scalar" variables (identifiers), array names with subscripts, operator or procedure names with arguments and statement expressions.

Examples:

```
x
x^3 - 2*y/(2*z^2 - df(x,z))
(p^2 + m^2)^(1/2)*log (y/m)
a(5) + b(i,q)
```

The symbol ** may be used as an alternative to the caret symbol (^) for forming powers, particularly in those systems that do not support a caret symbol.

Statement expressions, usually in parentheses, can also form part of a scalar expression, as in the example

```
w + (c:=x+y) + z .
```

When the algebraic value of an expression is needed, REDUCE determines it, starting with the algebraic values of the parts, roughly as follows:

Variables and operator symbols with an argument list have the algebraic values they were last assigned, or if never assigned stand for themselves. However, array elements have the algebraic values they were last assigned, or, if never assigned, are taken to be 0.

Procedures are evaluated with the values of their actual parameters.

In evaluating expressions, the standard rules of algebra are applied. Unfortunately, this algebraic evaluation of an expression is not as unambiguous as is numerical evaluation. This process is generally referred to as "simplification" in the sense that the evaluation usually but not always produces a simplified form for the expression.

There are many options available to the user for carrying out such simplification. If the user doesn't specify any method, the default method is used. The default evaluation of an expression involves expansion of the expression and collection of like terms, ordering of the terms, evaluation of derivatives and other functions and substitution for any expressions which have values assigned or declared (see assignments and `LET` statements). In many cases, this is all that the user needs.

The declarations by which the user can exercise some control over the way in which the evaluation is performed are explained in other sections. For example, if a real (floating point) number is encountered during evaluation, the system will normally convert it into a ratio of two integers. If the user wants to use real arithmetic, he can effect this by the command `on rounded;`. Other modes for coefficient arithmetic are described elsewhere.

If an illegal action occurs during evaluation (such as division by zero) or functions are called with the wrong number of arguments, and so on, an appropriate error message is generated.

3.2 Integer Expressions

These are expressions which, because of the values of the constants and variables in them, evaluate to whole numbers.

Examples:

```
2,      37 * 999,       (x + 3)^2 - x^2 - 6*x
```

are obviously integer expressions.

```
j + k - 2 * j^2
```

is an integer expression when J and K have values that are integers, or if not integers are such that "the variables and fractions cancel out", as in

```
k - 7/3 - j + 2/3 + 2*j^2.
```

3.3 Boolean Expressions

A boolean expression returns a truth value. In the algebraic mode of REDUCE, boolean expressions have the syntactical form:

```
<expression> <relational operator> <expression>
```

or

```
<boolean operator> (<arguments>)
```

or

```
<boolean expression> <logical operator>
<boolean expression>.
```

Parentheses can also be used to control the precedence of expressions.

In addition to the logical and relational operators defined earlier as infix operators, the following boolean operators are also defined:

`EVENP(U)`	determines if the number `U` is even or not;
`FIXP(U)`	determines if the expression `U` is integer or not;
`FREEOF(U,V)`	determines if the expression `U` does not contain the kernel `V` anywhere in its structure;
`NUMBERP(U)`	determines if `U` is a number or not;
`ORDP(U,V)`	determines if `U` is ordered ahead of `V` by some canonical ordering (based on the expression structure and an internal ordering of identifiers);
`PRIMEP(U)`	true if `U` is a prime object.

Examples:

```
j<1
x>0  or  x=-2
numberp x
fixp x and evenp x
numberp x and x neq 0
```

Boolean expressions can only appear directly within `IF`, `FOR`, `WHILE`, and `UNTIL` statements, as described in other sections. Such expressions cannot be used in place of ordinary algebraic expressions, or assigned to a variable.

NB: For those familiar with symbolic mode, the meaning of some of these operators is different in that mode. For example, `NUMBERP` is true only for integers and reals in symbolic mode.

When two or more boolean expressions are combined with `AND`, they are evaluated one by one until a *false* expression is found. The rest are not evaluated. Thus

```
numberp x and numberp y and x>y
```

does not attempt to make the `x>y` comparison unless `X` and `Y` are both verified to be numbers.

Similarly, evaluation of a sequence of boolean expressions connected by `OR` stops as soon as a *true* expression is found.

NB: In a boolean expression, and in a place where a boolean expression is expected, the algebraic value 0 is interpreted as *false*, while all other algebraic values are converted to *true*. So in algebraic mode a procedure can be written for direct usage in boolean expressions, returning say 1 or 0 as its value as in

```
procedure polynomialp(u,x);
   if den(u)=1 and deg(u,x)>=1 then 1 else 0;
```

One can then use this in a boolean construct, such as

```
if polynomialp(q,z) and not polynomialp(q,y) then ...
```

In addition, any procedure that does not have a defined return value (for example, a block without a RETURN statement in it) has the boolean value *false*.

3.4 Equations

Equations are a particular type of expression with the syntax

```
<expression> = <expression>.
```

In addition to their role as boolean expressions, they can also be used as arguments to several operators (e.g., SOLVE), and can be returned as values.

Under normal circumstances, the right-hand-side of the equation is evaluated but not the left-hand-side. This also applies to any substitutions made by the SUB operator. If both sides are to be evaluated, the switch EVALLHSEQP should be turned on.

To facilitate the handling of equations, two selectors, LHS and RHS, which return the left- and right-hand sides of a equation respectively, are provided. For example,

```
lhs(a+b=c) -> a+b
```

and

```
rhs(a+b=c) -> c.
```

3.5 Proper Statements as Expressions

Several kinds of proper statements deliver an algebraic or numerical result of some kind, which can in turn be used as an expression or part of an expression. For example, an assignment statement itself has a value, namely the value assigned. So

```
2 * (x := a+b)
```

is equal to `2*(a+b)`, as well as having the "side-effect" of assigning the value `a+b` to `X`. In context,

```
y := 2 * (x := a+b);
```

sets `X` to `a+b` and `Y` to `2*(a+b)`.

The sections on the various proper statement types indicate which of these statements are also useful as expressions.

Chapter 4

Lists

A list is an object consisting of a sequence of other objects (including lists themselves), separated by commas and surrounded by braces. Examples of lists are:

```
{a,b,c}

{1,a-b,c=d}

{{a},{{b,c},d},e}.
```

The empty list is represented as

```
{}.
```

4.1 Operations on Lists

Several operators in the system return their results as lists, and a user can create new lists using braces and commas. Alternatively, one can use the operator LIST to construct a list. An important class of operations on lists are MAP and SELECT operations. For details, please refer to the chapters on MAP, SELECT and the FOR command. See also the documentation on the ASSIST package.

To facilitate the use of lists, a number of operators are also available for manipulating them. `PART(<list>,n)` for example will return the n^{th} element of a list. `LENGTH` will return the length of a list. Several operators are also defined uniquely for lists. For those familiar with them, these operators in fact mirror the operations defined for Lisp lists. These operators are as follows:

4.1.1 LIST

The operator LIST is an alternative to the usage of curly brackets. LIST accepts an arbitrary number of arguments and returns a list of its arguments. This operator is useful in cases where operators have to be passed as arguments. E.g.,

```
list(a,list(list(b,c),d),e);       ->  {{a},{{b,c},d},e}
```

4.1.2 FIRST

This operator returns the first member of a list. An error occurs if the argument is not a list, or the list is empty.

4.1.3 SECOND

`SECOND` returns the second member of a list. An error occurs if the argument is not a list or has no second element.

4.1.4 THIRD

This operator returns the third member of a list. An error occurs if the argument is not a list or has no third element.

4.1.5 REST

`REST` returns its argument with the first element removed. An error occurs if the argument is not a list, or is empty.

4.1.6 . (Cons) Operator

This operator adds ("conses") an expression to the front of a list. For example:

```
a . {b,c}     ->   {a,b,c}.
```

4.1.7 APPEND

This operator appends its first argument to its second to form a new list. *Examples:*

```
append({a,b},{c,d})     ->     {a,b,c,d}
append({{a,b}},{c,d})   ->     {{a,b},c,d}.
```

4.1.8 REVERSE

The operator `REVERSE` returns its argument with the elements in the reverse order. It only applies to the top level list, not any lower level lists that may occur. Examples are:

```
reverse({a,b,c})        ->     {c,b,a}
reverse({{a,b,c},d})    ->     {d,{a,b,c}}.
```

4.1.9 List Arguments of Other Operators

If an operator other than those specifically defined for lists is given a single argument that is a list, then the result of this operation will be a list in which that operator is applied to each element of the list. For example, the result of evaluating `log{a,b,c}` is the expression `{LOG(A),LOG(B),LOG(C)}`.

There are two ways to inhibit this operator distribution. Firstly, the switch `LISTARGS`, if on, will globally inhibit such distribution. Secondly, one can inhibit this distribution for a specific operator by the declaration `LISTARGP`. For example, with the declaration `listargp log`, `log{a,b,c}` would evaluate to `LOG({A,B,C})`.

If an operator has more than one argument, no such distribution occurs.

4.1.10 Caveats and Examples

Some of the natural list operations such as *member* or *delete* are available only after loading the package *ASSIST*.

Please note that a non-list as second argument to CONS (a "dotted pair" in LISP terms) is not allowed and causes an "invalid as list" error.

```
 a := 17 . 4;

***** 17 4 invalid as list
```

Also, the initialization of a scalar variable is not the empty list – one has to set list type variables explicitly, as in the following example:

```
 load_package assist;

 procedure lotto (n,m);
  begin scalar list_1_n, luckies, hit;
     list_1_n := {};
     luckies := {};
     for k:=1:n do list_1_n := k . list_1_n;
     for k:=1:m do
       << hit := part(list_1_n,random(n-k+1) + 1);
          list_1_n := delete(hit,list_1_n);
          luckies := hit . luckies >>;
     return luckies;
  end;                             % In Germany, try lotto (49,6);
```

Another example: Find all coefficients of a multivariate polynomial with respect to a list of variables:

```
procedure allcoeffs(q,lis); % q : polynomial, lis: list of vars
   allcoeffs1 (list q,lis);

procedure allcoeffs1(q,lis);
  if lis={} then q else
    allcoeffs1(foreach qq in q join coeff(qq,first lis),rest lis);
```

Chapter 5

Statements

A statement is any combination of reserved words and expressions, and has the syntax

```
<statement> ::= <expression>|<proper statement>
```

A REDUCE program consists of a series of commands which are statements followed by a terminator:

```
<terminator> ::= ;|$
```

The division of the program into lines is arbitrary. Several statements can be on one line, or one statement can be freely broken onto several lines. If the program is run interactively, statements ending with ; or $ are not processed until an end-of-line character is encountered. This character can vary from system to system, but is normally the Return key on an ASCII terminal. Specific systems may also use additional keys as statement terminators.

If a statement is a proper statement, the appropriate action takes place.

Depending on the nature of the proper statement some result or response may or may not be printed out, and the response may or may not depend on the terminator used.

If a statement is an expression, it is evaluated. If the terminator is a semicolon, the result is printed. If the terminator is a dollar sign, the result is not printed. Because it is not usually possible to know in advance how large an expression will be, no explicit format statements are offered to the user. However, a variety of output declarations are available so that the output can be produced in different forms. These output declarations are explained in Section 8.3.3.

The following sub-sections describe the types of proper statements in REDUCE.

5.1 Assignment Statements

These statements have the syntax

```
<assignment statement> ::= <expression> := <expression>
```

The `<expression>` on the left side is normally the name of a variable, an operator symbol with its list of arguments filled in, or an array name with the proper number of integer subscript values within the array bounds. For example:

```
a1 := b + c
h(l,m) := x-2*y                (where h is an operator)
k(3,5) := x-2*y                (where k is a 2-dim. array)
```

More general assignments such as `a+b := c` are also allowed. The effect of these is explained in Section 10.2.5.

An assignment statement causes the expression on the right-hand-side to be evaluated. If the left-hand-side is a variable, the value of the right-hand-side is assigned to that unevaluated variable. If the left-hand-side is an operator or array expression, the arguments of that operator or array are evaluated, but no other simplification done. The evaluated right-hand-side is then assigned to the resulting expression. For example, if `A` is a single-dimensional array, `a(1+1) := b` assigns the value `B` to the array element `a(2)`.

If a semicolon is used as the terminator when an assignment is issued as a command (i.e. not as a part of a group statement or procedure or other similar construct), the left-hand side symbol of the assignment statement is printed out, followed by a "`:=`", followed by the value of the expression on the right.

It is also possible to write a multiple assignment statement:

```
<expression> := ... := <expression> := <expression>
```

In this form, each `<expression>` but the last is set to the value of the last `<expression>`. If a semicolon is used as a terminator, each expression except the last is printed followed by a "`:=`" ending with the value of the last expression.

5.1.1 Set Statement

In some cases, it is desirable to perform an assignment in which *both* the left- and right-hand sides of an assignment are evaluated. In this case, the `SET` statement can be used with the syntax:

```
SET(<expression>,<expression>);
```

For example, the statements

```
j := 23;
set(mkid(a,j),x);
```

assigns the value `X` to `A23`.

5.2 Group Statements

The group statement is a construct used where REDUCE expects a single statement, but a series of actions needs to be performed. It is formed by enclosing one or more statements (of any kind) between the symbols $<<$ and $>>$, separated by semicolons or dollar signs – it doesn't matter which. The statements are executed one after another.

Examples will be given in the sections on `IF` and other types of statements in which the $<< \ldots >>$ construct is useful.

If the last statement in the enclosed group has a value, then that is also the value of the group statement. Care must be taken not to have a semicolon or dollar sign after the last grouped statement, if the value of the group is relevant: such an extra terminator causes the group to have the value NIL or zero.

5.3 Conditional Statements

The conditional statement has the following syntax:

```
<conditional statement> ::=
   IF <boolean expression> THEN <statement> [ELSE <statement>]
```

The boolean expression is evaluated. If this is *true*, the first `<statement>` is executed. If it is *false*, the second is.

Examples:

```
if x=5 then a:=b+c else d:=e+f

if x=5 and numberp y
   then <<ff:=q1; a:=b+c>>
   else <<ff:=q2; d:=e+f>>
```

Note the use of the group statement.
Conditional statements associate to the right; i.e.,

```
IF <a> THEN <b> ELSE IF <c> THEN <d> ELSE <e>
```

is equivalent to:

```
IF <a> THEN <b> ELSE (IF <c> THEN <d> ELSE <e>)
```

In addition, the construction

```
IF <a> THEN IF <b> THEN <c> ELSE <d>
```

parses as

```
IF <a> THEN (IF <b> THEN <c> ELSE <d>).
```

If the value of the conditional statement is of primary interest, it is often called a conditional expression instead. Its value is the value of whichever statement was executed. (If the executed statement has no value, the conditional expression has no value or the value 0, depending on how it is used.)

Examples:

```
a:=if x<5 then 123 else 456;
```

```
b:=u + v^(if numberp z then 10*z  else 1) + w;
```

If the value is of no concern, the ELSE clause may be omitted if no action is required in the *false* case.

```
if x=5 then a:=b+c;
```

Note: As explained in Section 3.3,a if a scalar or numerical expression is used in place of the boolean expression – for example, a variable is written there – the *true* alternative is followed unless the expression has the value 0.

5.4 FOR Statements

The FOR statement is used to define a variety of program loops. Its general syntax is as follows:

$$\texttt{FOR}\left\{\begin{array}{c} \langle var\rangle \ \texttt{:=}\ \langle number\rangle \left\{\begin{array}{c}\texttt{STEP}\ \langle number\rangle\ \texttt{UNTIL}\\ \texttt{:}\end{array}\right\}\langle number\rangle \\ \texttt{EACH}\ \langle var\rangle \left\{\begin{array}{c}\texttt{IN}\\ \texttt{ON}\end{array}\right\}\langle list\rangle\end{array}\right\}\langle action\rangle\ \langle exprn\rangle$$

where

$$\langle action\rangle \ \texttt{::= do|product|sum|collect|join.}$$

The assignment form of the FOR statement defines an iteration over the indicated numerical range. If expressions that do not evaluate to numbers are used in the designated places, an error will result.

The FOR EACH form of the FOR statement is designed to iterate down a list. Again, an error will occur if a list is not used.

The action DO means that <exprn> is simply evaluated and no value kept; the statement returning 0 in this case (or no value at the top level). COLLECT means that the results of evaluating <exprn> each time are linked together to make a list, and JOIN means that the values of <exprn> are themselves lists that are joined to make one list (similar to CONC in Lisp). Finally, PRODUCT and SUM form the respective combined value out of the values of <exprn>.

In all cases, <exprn> is evaluated algebraically within the scope of the current value of <var>. If <action> is DO, then nothing else happens. In other cases, `<action>` is a binary operator that causes a result to be built up and returned by `FOR`. In those cases, the loop is initialized to a default value (`0` for `SUM`, `1` for `PRODUCT`, and an empty list for the other actions). The test for the end condition is made before any action is taken. As in Pascal, if the variable is out of range in the assignment case, or the `<list>` is empty in the `FOR EACH` case, `<exprn>` is not evaluated at all.

Examples:

1. If `A`, `B` have been declared to be arrays, the following stores 5^2 through 10^2 in `A(5)` through `A(10)`, and at the same time stores the cubes in the `B` array:

   ```
   for i := 5 step 1 until 10 do <<a(i):=i^2; b(i):=i^3>>
   ```

2. As a convenience, the common construction

   ```
   STEP 1 UNTIL
   ```

   may be abbreviated to a colon. Thus, instead of the above we could write:

   ```
   for i := 5:10 do <<a(i):=i^2; b(i):=i^3>>
   ```

3. The following sets `C` to the sum of the squares of 1,3,5,7,9; and `D` to the expression `x*(x+1)*(x+2)*(x+3)*(x+4)`:

   ```
   c := for j:=1 step 2 until 9 sum j^2;
   d := for k:=0 step 1 until 4 product (x+k);
   ```

4. The following forms a list of the squares of the elements of the list `{a,b,c}`:

   ```
   for each x in {a,b,c} collect x^2;
   ```

5. The following forms a list of the listed squares of the elements of the list `{a,b,c}` (i.e., `{{A^2},{B^2},{C^2}}`):

   ```
   for each x in {a,b,c} collect {x^2};
   ```

6. The following also forms a list of the squares of the elements of the list `{a,b,c}`, since the `JOIN` operation joins the individual lists into one list:

```
for each x in {a,b,c} join {x^2};
```

The control variable used in the `FOR` statement is actually a new variable, not related to the variable of the same name outside the `FOR` statement. In other words, executing a statement `for i:=` ... doesn't change the system's assumption that $i^2 = -1$. Furthermore, in algebraic mode, the value of the control variable is substituted in `<exprn>` only if it occurs explicitly in that expression. It will not replace a variable of the same name in the value of that expression. For example:

```
b := a; for a := 1:2 do write b;
```

prints `A` twice, not 1 followed by 2.

5.5 WHILE . . . DO

The `FOR ...DO` feature allows easy coding of a repeated operation in which the number of repetitions is known in advance. If the criterion for repetition is more complicated, `WHILE ...DO` can often be used. Its syntax is:

```
WHILE <boolean expression> DO <statement>
```

The `WHILE ...DO` controls the single statement following `DO`. If several statements are to be repeated, as is almost always the case, they must be grouped using the $<< \ldots >>$ or `BEGIN ...END` as in the example below.

The `WHILE` condition is tested each time *before* the action following the `DO` is attempted. If the condition is false to begin with, the action is not performed at all. Make sure that what is to be tested has an appropriate value initially.

Example:

Suppose we want to add up a series of terms, generated one by one, until we reach a term which is less than 1/1000 in value. For our simple example, let us suppose the first term equals 1 and each term is obtained from the one before by taking one third of it and adding one third its square. We would write:

```
ex:=0; term:=1;
while num(term - 1/1000) >= 0  do
        <<ex := ex+term; term:=(term + term^2)/3>>;
ex;
```

As long as `TERM` is greater than or equal to (`>=`) 1/1000 it will be added to `EX` and the next `TERM` calculated. As soon as `TERM` becomes less than 1/1000 the `WHILE` test fails and the `TERM` will not be added.

5.6 REPEAT ... UNTIL

`REPEAT ...UNTIL` is very similar in purpose to `WHILE ...DO`. Its syntax is:

```
REPEAT <statement> UNTIL <boolean expression>
```

(PASCAL users note: Only a single statement – usually a group statement – is allowed between the `REPEAT` and the `UNTIL`.)

There are two essential differences:

1. The test is performed *after* the controlled statement (or group of statements) is executed, so the controlled statement is always executed at least once.
2. The test is a test for when to stop rather than when to continue, so its "polarity" is the opposite of that in `WHILE ...DO`.

As an example, we rewrite the example from the `WHILE ...DO` section:

```
ex:=0; term:=1;
repeat <<ex := ex+term; term := (term + term^2)/3>>
   until num(term - 1/1000) < 0;
ex;
```

In this case, the answer will be the same as before, because in neither case is a term added to `EX` which is less than 1/1000.

5.7 Compound Statements

Often the desired process can best (or only) be described as a series of steps to be carried out one after the other. In many cases, this can be achieved by use of the group statement. However, each step often provides some intermediate result, until at the end we have the final result wanted. Alternatively, iterations on the steps are needed that are not possible with constructs such as `WHILE` or `REPEAT` statements. In such cases the steps of the process must be enclosed between the words `BEGIN` and `END` forming what is technically called a *block* or *compound* statement. Such a compound statement can in fact be used wherever a group statement appears. The converse is not true: `BEGIN ...END` can be used in ways that << ...>> cannot.

If intermediate results must be formed, local variables must be provided in which to store them. *Local* means that their values are deleted as soon as the block's operations are complete, and there is no conflict with variables outside the block that happen to have the same name. Local variables are created by a `SCALAR` declaration immediately after the `BEGIN`:

```
scalar a,b,c,z;
```

If more convenient, several `SCALAR` declarations can be given one after another:

```
scalar a,b,c;
scalar z;
```

In place of `SCALAR` one can also use the declarations `INTEGER` or `REAL`. In the present version of REDUCE variables declared `INTEGER` are expected to have only integer values, and are initialized to 0. `REAL` variables on the other hand are currently treated as algebraic mode `SCALAR`s.

CAUTION: `INTEGER`, `REAL` and `SCALAR` declarations can only be given immediately after a `BEGIN`. An error will result if they are used after other statements in a block (including `ARRAY` and `OPERATOR` declarations, which are global in scope), or outside the top-most block (e.g., at the top level). All variables declared `SCALAR` are automatically initialized to zero in algebraic mode (`NIL` in symbolic mode).

Any symbols not declared as local variables in a block refer to the variables of the same name in the current calling environment. In particular, if they are not so declared at a higher level (e.g., in a surrounding block or as parameters in a calling procedure), their values can be permanently changed.

Following the `SCALAR` declaration(s), if any, write the statements to be executed, one after the other, separated by delimiters (e.g., `;` or `$`) (it doesn't matter which). However, from a stylistic point of view, `;` is preferred.

The last statement in the body, just before `END`, need not have a terminator (since the `BEGIN ...END` are in a sense brackets confining the block statements). The last statement must also be the command RETURN followed by the variable or expression whose value is to be the value returned by the procedure. If the RETURN is omitted (or nothing is written after the word RETURN) the procedure will have no value or the value zero, depending on how it is used (and NIL in symbolic mode). Remember to put a terminator after the END.

Example:

Given a previously assigned integer value for N, the following block will compute the Legendre polynomial of degree N in the variable X:

```
begin scalar seed,deriv,top,fact;
   seed:=1/(y^2 - 2*x*y +1)^(1/2);
   deriv:=df(seed,y,n);
   top:=sub(y=0,deriv);
   fact:=for i:=1:n product i;
   return top/fact
end;
```

5.7.1 Compound Statements with GO TO

It is possible to have more complicated structures inside the BEGIN ...END brackets than indicated in the previous example. That the individual lines of the program need not be assignment statements, but could be almost any other kind of statement or command, needs no explanation. For example, conditional statements, and WHILE and REPEAT constructions, have an obvious role in defining more intricate blocks.

If these structured constructs don't suffice, it is possible to use labels and GO TOs within a compound statement, and also to use RETURN in places within the block other than just before the END. The following subsections discuss these matters in detail. For many readers the following example, presenting one possible definition of a process to calculate the factorial of N for preassigned N will suffice:

Example:

```
begin scalar m;
    m:=1;
 l: if n=0 then return m;
    m:=m*n;
    n:=n-1;
```

```
    go to l
end;
```

5.7.2 Labels and GO TO Statements

Within a BEGIN ...END compound statement it is possible to label statements, and transfer to them out of sequence using GO TO statements. Only statements on the top level inside compound statements can be labeled, not ones inside subsidiary constructions like << ...>>, IF ...THEN ..., WHILE ...DO ..., etc.

Labels and GO TO statements have the syntax:

```
<go to statement> ::= GO TO <label> | GOTO <label>
<label> ::= <identifier>
<labeled statement> ::= <label>:<statement>
```

Note that statement names cannot be used as labels.

While GO TO is an unconditional transfer, it is frequently used in conditional statements such as

```
if x>5 then go to abcd;
```

giving the effect of a conditional transfer.

Transfers using GO TOs can only occur within the block in which the GO TO is used. In other words, you cannot transfer from an inner block to an outer block using a GO TO. However, if a group statement occurs within a compound statement, it is possible to jump out of that group statement to a point within the compound statement using a GO TO.

5.7.3 RETURN Statements

The value corresponding to a BEGIN ...END compound statement, such as a procedure body, is normally 0 (NIL in symbolic mode). By executing a RETURN statement in the compound statement a different value can be returned. After a RETURN statement is executed, no further statements within the compound statement are executed.

Examples:

```
return x+y;
return m;
return;
```

Note that parentheses are not required around the x+y, although they are permitted. The last example is equivalent to return 0 or return nil, depending on whether the block is used as part of an expression or not.

Since RETURN actually moves up only one block level, in a sense the casual user is not expected to understand, we tabulate some cautions concerning its use.

1. RETURN can be used on the top level inside the compound statement, i.e. as one of the statements bracketed together by the BEGIN ...END

2. RETURN can be used within a top level << ... >> construction within the compound statement. In this case, the RETURN transfers control out of both the group statement and the compound statement.

3. RETURN can be used within an IF ... THEN ... ELSE ... on the top level within the compound statement.

NOTE: At present, there is no construct provided to permit early termination of a FOR, WHILE, or REPEAT statement. In particular, the use of RETURN in such cases results in a syntax error. For example,

```
begin scalar y;
   y := for i:=0:99 do if a(i)=x then return b(i);
   ...
```

will lead to an error.

Chapter 6

Commands and Declarations

A command is an order to the system to do something. Some commands cause visible results (such as calling for input or output); others, usually called declarations, set options, define properties of variables, or define procedures. Commands are formally defined as a statement followed by a terminator

```
<command> ::= <statement> <terminator>
<terminator> ::= ;|$
```

Some REDUCE commands and declarations are described in the following sub-sections.

6.1 Array Declarations

Array declarations in REDUCE are similar to FORTRAN dimension statements. For example:

```
array a(10),b(2,3,4);
```

Array indices each range from 0 to the value declared. An element of an array is referred to in standard FORTRAN notation, e.g. `A(2)`.

We can also use an expression for defining an array bound, provided the value of the expression is a positive integer. For example, if `X` has the value 10 and `Y` the value 7 then `array c(5*x+y)` is the same as `array c(57)`.

If an array is referenced by an index outside its range, an error occurs. If the array is to be one-dimensional, and the bound a number or a variable (not a more general expression) the parentheses may be omitted:

```
array a 10, c 57;
```

The operator `LENGTH` applied to an array name returns a list of its dimensions.

All array elements are initialized to 0 at declaration time. In other words, an array element has an *instant evaluation* property and cannot stand for itself. If this is required, then an operator should be used instead.

Array declarations can appear anywhere in a program. Once a symbol is declared to name an array, it can not also be used as a variable, or to name an operator or a procedure. It can however be re-declared to be an array, and its size may be changed at that time. An array name can also continue to be used as a parameter in a procedure, or a local variable in a compound statement, although this use is not recommended, since it can lead to user confusion over the type of the variable.

Arrays once declared are global in scope, and so can then be referenced anywhere in the program. In other words, unlike arrays in most other languages, a declaration within a block (or a procedure) does not limit the scope of the array to that block, nor does the array go away on exiting the block (use `CLEAR` instead for this purpose).

6.2 Mode Handling Declarations

The `ON` and `OFF` declarations are available to the user for controlling various system options. Each option is represented by a *switch* name. `ON` and `OFF` take a list of switch names as argument and turn them on and off respectively, e.g.,

```
on time;
```

causes the system to print a message after each command giving the elapsed CPU time since the last command, or since `TIME` was last turned off, or the session began. Another useful switch with interactive use is `DEMO`, which causes the system to pause after each command in a file (with the exception of comments) until a $\boxed{\text{Return}}$ is typed on the terminal. This enables a user to set up a demonstration file and step through it command by command.

As with most declarations, arguments to `ON` and `OFF` may be strung together separated by commas. For example,

```
off time,demo;
```

will turn off both the time messages and the demonstration switch.

We note here that while most `ON` and `OFF` commands are obeyed almost instantaneously, some trigger time-consuming actions such as reading in necessary modules from secondary storage.

A diagnostic message is printed if `ON` or `OFF` are used with a switch that is not known to the system. For example, if you misspell `DEMO` and type

```
on demq;
```

you will get the message

```
***** DEMQ not defined as switch.
```

6.3 END

The identifier `END` has two separate uses.

1) Its use in a `BEGIN ...END` bracket has been discussed in connection with compound statements.

2) Files to be read using `IN` should end with an extra `END`; command. The reason for this is explained in the section on the `IN` command. This use of `END` does not allow an immediately preceding `END` (such as the `END` of a procedure definition), so we advise using `;END;` there.

6.4 BYE Command

The command `BYE`; (or alternatively `QUIT`;) stops the execution of REDUCE, closes all open output files, and returns you to the calling program (usually the operating system). Your REDUCE session is normally destroyed.

6.5 SHOWTIME Command

`SHOWTIME`; prints the elapsed time since the last call of this command or, on its first call, since the current REDUCE session began. The time is normally given in milliseconds and gives the time as measured by a system clock. The operations covered by this measure are system dependent.

6.6 DEFINE Command

The command `DEFINE` allows a user to supply a new name for any identifier or replace it by any well-formed expression. Its argument is a list of expressions of the form

```
<identifier> = <number>|<identifier>|<operator>|
                  <reserved word>|<expression>
```

Example:

```
define be==,x=y+z;
```

means that `BE` will be interpreted as an equal sign, and `X` as the expression `y+z` from then on. This renaming is done at parse time, and therefore takes precedence over any other replacement declared for the same identifier. It stays in effect until the end of the REDUCE run.

The identifiers `ALGEBRAIC` and `SYMBOLIC` have properties which prevent `DEFINE` from being used on them. To define `ALG` to be a synonym for `ALGEBRAIC`, use the more complicated construction

```
put('alg,'newnam,'algebraic);
```

Chapter 7

Built-in Prefix Operators

In the following subsections are descriptions of the most useful prefix operators built into REDUCE that are not defined in other sections (such as substitution operators). Some are fully defined internally as procedures; others are more nearly abstract operators, with only some of their properties known to the system.

In many cases, an operator is described by a prototypical header line as follows. Each formal parameter is given a name and followed by its allowed type. The names of classes referred to in the definition are printed in lower case, and parameter names in upper case. If a parameter type is not commonly used, it may be a specific set enclosed in brackets { ...}. Operators that accept formal parameter lists of arbitrary length have the parameter and type class enclosed in square brackets indicating that zero or more occurrences of that argument are permitted. Optional parameters and their type classes are enclosed in angle brackets.

7.1 Numerical Operators

REDUCE includes a number of functions that are analogs of those found in most numerical systems. With numerical arguments, such functions return the expected result. However, they may also be called with non-numerical arguments. In such cases, except where noted, the system attempts to simplify the expression as far as it can. In such cases, a residual expression involving the original operator usually remains. These operators are as follows:

7.1.1 ABS

ABS returns the absolute value of its single argument, if that argument has a numerical value. A non-numerical argument is returned as an absolute value, with an overall numerical coefficient taken outside the absolute value operator. For example:

```
abs(-3/4)      ->  3/4
abs(2a)        ->  2*ABS(A)
abs(i)         ->  1
abs(-x)        ->  ABS(X)
```

7.1.2 CEILING

This operator returns the ceiling (i.e., the least integer greater than the given argument) if its single argument has a numerical value. A non-numerical argument is returned as an expression in the original operator. For example:

```
ceiling(-5/4) ->  -1
ceiling(-a)   ->  CEILING(-A)
```

7.1.3 CONJ

This returns the complex conjugate of an expression, if that argument has an numerical value. A non-numerical argument is returned as an expression in the operators REPART and IMPART. For example:

```
conj(1+i)     -> 1-I
conj(a+i*b)   -> REPART(A) - REPART(B)*I - IMPART(A)*I
                 - IMPART(B)
```

7.1.4 FACTORIAL

If the single argument of FACTORIAL evaluates to a non-negative integer, its factorial is returned. Otherwise an expression involving FACTORIAL is returned. For example:

```
factorial(5)  ->  120
```

```
factorial(a)  ->  FACTORIAL(A)
```

7.1.5 FIX

This operator returns the fixed value (i.e., the integer part of the given argument) if its single argument has a numerical value. A non-numerical argument is returned as an expression in the original operator. For example:

```
fix(-5/4)   ->  -1
fix(a)      ->  FIX(A)
```

7.1.6 FLOOR

This operator returns the floor (i.e., the greatest integer less than the given argument) if its single argument has a numerical value. A non-numerical argument is returned as an expression in the original operator. For example:

```
floor(-5/4)   ->  -2
floor(a)      ->  FLOOR(A)
```

7.1.7 IMPART

This operator returns the imaginary part of an expression, if that argument has an numerical value. A non-numerical argument is returned as an expression in the operators `REPART` and `IMPART`. For example:

```
impart(1+i)   -> 1
impart(a+i*b) -> REPART(B) + IMPART(A)
```

7.1.8 MAX/MIN

`MAX` and `MIN` can take an arbitrary number of expressions as their arguments. If all arguments evaluate to numerical values, the maximum or minimum of the argument list is returned. If any argument is non-numeric, an appropriately reduced expression is returned. For example:

```
max(2,-3,4,5) ->  5
min(2,-2)     ->  -2.
max(a,2,3)    ->  MAX(A,3)
```

```
min(x)        ->  X
```

`MAX` or `MIN` of an empty list returns 0.

7.1.9 NEXTPRIME

`NEXTPRIME` returns the next prime greater than its integer argument, using a probabilistic algorithm. A type error occurs if the value of the argument is not an integer. For example:

```
nextprime(5)      ->  7
nextprime(-2)     ->  2
nextprime(-7)     -> -5
nextprime 1000000 -> 1000003
```

whereas `nextprime(a)` gives a type error.

7.1.10 RANDOM

`random(`n`)` returns a random number r in the range $0 \leq r < n$. A type error occurs if the value of the argument is not a positive integer in algebraic mode, or positive number in symbolic mode. For example:

```
random(5)         ->    3
random(1000)      ->  191
```

whereas `random(a)` gives a type error.

7.1.11 RANDOM_NEW_SEED

`random_new_seed(`n`)` reseeds the random number generator to a sequence determined by the integer argument n. It can be used to ensure that a repeatable pseudo-random sequence will be delivered regardless of any previous use of `RANDOM`, or can be called early in a run with an argument derived from something variable (such as the time of day) to arrange that different runs of a REDUCE program will use different random sequences. When a fresh copy of REDUCE is first created it is as if `random_new_seed(1)` has been obeyed.

A type error occurs if the value of the argument is not a positive integer.

7.1.12 REPART

This returns the real part of an expression, if that argument has an numerical value. A non-numerical argument is returned as an expression in the operators `REPART` and `IMPART`. For example:

```
repart(1+i)   -> 1
repart(a+i*b) -> REPART(A) - IMPART(B)
```

7.1.13 ROUND

This operator returns the rounded value (i.e, the nearest integer) of its single argument if that argument has a numerical value. A non-numeric argument is returned as an expression in the original operator. For example:

```
round(-5/4)   ->  -1
round(a)      ->  ROUND(A)
```

7.1.14 SIGN

`SIGN` tries to evaluate the sign of its argument. If this is possible `SIGN` returns one of 1, 0 or -1. Otherwise, the result is the original form or a simplified variant. For example:

```
sign(-5)      ->  -1
sign(-a^2*b)  ->  -SIGN(B)
```

Note that even powers of formal expressions are assumed to be positive only as long as the switch `COMPLEX` is off.

7.2 Mathematical Functions

REDUCE knows that the following represent mathematical functions that can take arbitrary scalar expressions as their single argument:

```
ACOS ACOSH ACOT ACOTH ACSC ACSCH ASEC ASECH ASIN ASINH
ATAN ATANH ATAN2 COS COSH COT COTH CSC CSCH DILOG EI EXP
HYPOT LN LOG LOGB LOG10 SEC SECH SIN SINH SQRT TAN TANH
```

where `LOG` is the natural logarithm (and equivalent to `LN`), and `LOGB` has two arguments of which the second is the logarithmic base.

The derivatives of all these functions are also known to the system.

REDUCE knows various elementary identities and properties of these functions. For example:

```
cos(-x) = cos(x)              sin(-x) = - sin (x)
cos(n*pi) = (-1)^n            sin(n*pi) = 0
log(e)  = 1                   e^(i*pi/2) = i
log(1)  = 0                   e^(i*pi) = -1
log(e^x) = x                  e^(3*i*pi/2) = -i
```

Beside these identities, there are a lot of simplifications for elementary functions defined in the REDUCE system as rulelists. In order to view these, the SHOWRULES operator can be used, e.g.

```
      SHOWRULES tan;

{tan(~n*arbint(~i)*pi + ~(~ x)) => tan(x) when fixp(n),

 tan(~x)

  => trigquot(sin(x),cos(x)) when knowledge_about(sin,x,tan)

 ,

      ~x + ~(~ k)*pi
 tan(----------------)
            ~d

            x                              k    1
  =>  - cot(---) when x freeof pi and abs(---)=---,
            d                              d    2

      ~(~ w) + ~(~ k)*pi           w + remainder(k,d)*pi
 tan(--------------------) => tan(-----------------------)
            ~(~ d)                           d

                              k
 when w freeof pi and ratnump(---) and fixp(k)
                              d

          k
```

```
  and abs(---)>=1,
          d

 tan(atan(~x)) => x,

                                  2
 df(tan(~x),~x) => 1 + tan(x) }
```

For further simplification, especially of expressions involving trigonometric functions, see the TRIGSIMP package documentation.

Functions not listed above may be defined in the special functions package SPECFN.

The user can add further rules for the reduction of expressions involving these operators by using the `LET` command.

In many cases it is desirable to expand product arguments of logarithms, or collect a sum of logarithms into a single logarithm. Since these are inverse operations, it is not possible to provide rules for doing both at the same time and preserve the REDUCE concept of idempotent evaluation. As an alternative, REDUCE provides two switches `EXPANDLOGS` and `COMBINELOGS` to carry out these operations. Both are off by default. Thus to expand `LOG(X*Y)` into a sum of logs, one can say

```
ON EXPANDLOGS; LOG(X*Y);
```

and to combine this sum into a single log:

```
ON COMBINELOGS; LOG(X) + LOG(Y);
```

At the present time, it is possible to have both switches on at once, which could lead to infinite recursion. However, an expression is switched from one form to the other in this case. Users should not rely on this behavior, since it may change in the next release.

The current version of REDUCE does a poor job of simplifying surds. In particular, expressions involving the product of variables raised to non-integer powers do not usually have their powers combined internally, even though they are printed as if those powers were combined. For example, the expression

```
x^(1/3)*x^(1/6);
```

will print as

```
SQRT(X)
```

but will have an internal form containing the two exponentiated terms. If you now subtract `sqrt(x)` from this expression, you will *not* get zero. Instead, the confusing form

```
SQRT(X) - SQRT(X)
```

will result. To combine such exponentiated terms, the switch `COMBINEEXPT` should be turned on.

The square root function can be input using the name `SQRT`, or the power operation `^(1/2)`. On output, unsimplified square roots are normally represented by the operator `SQRT` rather than a fractional power. With the default system switch settings, the argument of a square root is first simplified, and any divisors of the expression that are perfect squares taken outside the square root argument. The remaining expression is left under the square root. Thus the expression

```
sqrt(-8a^2*b)
```

becomes

```
2*a*sqrt(-2*b).
```

Note that such simplifications can cause trouble if `A` is eventually given a value that is a negative number. If it is important that the positive property of the square root and higher even roots always be preserved, the switch `PRECISE` should be set on (the default value). This causes any non-numerical factors taken out of surds to be represented by their absolute value form. With `PRECISE` on then, the above example would become

```
2*abs(a)*sqrt(-2*b).
```

The statement that REDUCE knows very little about these functions applies only in the mathematically exact `off rounded` mode. If `ROUNDED` is on, any of the functions

```
ACOS ACOSH ACOT ACOTH ACSC ACSCH ASEC ASECH ASIN ASINH
ATAN ATANH ATAN2 COS COSH COT COTH CSC CSCH EXP HYPOT
LN LOG LOGB LOG10 SEC SECH SIN SINH SQRT TAN TANH
```

when given a numerical argument has its value calculated to the current degree of floating point precision. In addition, real (non-integer valued) powers of numbers will also be evaluated.

If the COMPLEX switch is turned on in addition to ROUNDED, these functions will also calculate a real or complex result, again to the current degree of floating point precision, if given complex arguments. For example, with `on rounded,complex;`

```
2.3^(5.6i)    ->    -0.0480793490914 - 0.998843519372*I
cos(2+3i)     ->    -4.18962569097 - 9.10922789376*I
```

7.3 DF Operator

The operator DF is used to represent partial differentiation with respect to one or more variables. It is used with the syntax:

```
DF(EXPRN:algebraic[,VAR:kernel<,NUM:integer>]):algebraic.
```

The first argument is the expression to be differentiated. The remaining arguments specify the differentiation variables and the number of times they are applied.

The number NUM may be omitted if it is 1. For example,

`df(y,x)`	$= \partial y/\partial x$
`df(y,x,2)`	$= \partial^2 y/\partial x^2$
`df(y,x1,2,x2,x3,2)`	$= \partial^5 y/\partial x_1^2\ \partial x_2 \partial x_3^2.$

The evaluation of `df(y,x)` proceeds as follows: first, the values of Y and X are found. Let us assume that X has no assigned value, so its value is X. Each term or other part of the value of Y that contains the variable X is differentiated by the standard rules. If Z is another variable, not X itself, then its derivative with respect to X is taken to be 0, unless Z has previously been declared to DEPEND on X, in which case the derivative is reported as the symbol `df(z,x)`.

7.3.1 Adding Differentiation Rules

The LET statement can be used to introduce rules for differentiation of user-defined operators. Its general form is

```
FOR ALL <var1>,...,<varn>
     LET DF(<operator><varlist>,<vari>)=<expression>
```

where `<varlist>` ::= (`<var1>`,...,`<varn>`), and `<var1>`,...,`<varn>` are the dummy variable arguments of `<operator>`.

An analogous form applies to infix operators.

Examples:

```
for all x let df(tan x,x)= 1 + tan(x)^2;
```

(This is how the tan differentiation rule appears in the REDUCE source.)

```
for all x,y let df(f(x,y),x)=2*f(x,y),
                df(f(x,y),y)=x*f(x,y);
```

Notice that all dummy arguments of the relevant operator must be declared arbitrary by the `FOR ALL` command, and that rules may be supplied for operators with any number of arguments. If no differentiation rule appears for an argument in an operator, the differentiation routines will return as result an expression in terms of `DF`. For example, if the rule for the differentiation with respect to the second argument of `F` is not supplied, the evaluation of `df(f(x,z),z)` would leave this expression unchanged. (No `DEPEND` declaration is needed here, since `f(x,z)` obviously "depends on" `Z`.)

Once such a rule has been defined for a given operator, any future differentiation rules for that operator must be defined with the same number of arguments for that operator, otherwise we get the error message

```
Incompatible DF rule argument length for <operator>
```

7.4 INT Operator

`INT` is an operator in REDUCE for indefinite integration using a combination of the Risch-Norman algorithm and pattern matching. It is used with the syntax:

```
INT(EXPRN:algebraic,VAR:kernel):algebraic.
```

This will return correctly the indefinite integral for expressions comprising polynomials, log functions, exponential functions and tan and atan. The arbitrary constant is not represented. If the integral cannot be done in closed terms, it returns a formal integral for the answer in one of two ways:

1. It returns the input, `INT(...,...)` unchanged.
2. It returns an expression involving `INT`s of some other functions (sometimes more complicated than the original one, unfortunately).

Rational functions can be integrated when the denominator is factorizable by the program. In addition it will attempt to integrate expressions involving error functions, dilogarithms and other trigonometric expressions. In these cases it might not always succeed in finding the solution, even if one exists.

Examples:

```
int(log(x),x) ->  X*(LOG(X) - 1),
int(e^x,x)    ->  E**X.
```

The program checks that the second argument is a variable and gives an error if it is not.

Note: If the `int` operator is called with 4 arguments, REDUCE will implicitly call the definite integration package (DEFINT) and this package will interpret the third and fourth arguments as the lower and upper limit of integration, respectively. For details, consult the documentation on the DEFINT package.

7.4.1 Options

The switch `TRINT` when on will trace the operation of the algorithm. It produces a great deal of output in a somewhat illegible form, and is not of much interest to the general user. It is normally off.

If the switch `FAILHARD` is on the algorithm will terminate with an error if the integral cannot be done in closed terms, rather than return a formal integration form. `FAILHARD` is normally off.

The switch `NOLNR` suppresses the use of the linear properties of integration in cases when the integral cannot be found in closed terms. It is normally off.

7.4.2 Advanced Use

If a function appears in the integrand that is not one of the functions `EXP, ERF, TAN, ATAN, LOG, DILOG` then the algorithm will make an attempt to integrate the argument if it can, differentiate it and reach a known function. However the answer cannot be guaranteed in this case. If a function is known to be algebraically independent of this set it can be flagged transcendental by

```
flag('(trilog),'transcendental);
```

in which case this function will be added to the permitted field descriptors for a genuine decision procedure. If this is done the user is responsible for the mathematical correctness of his actions.

The standard version does not deal with algebraic extensions. Thus integration of expressions involving square roots and other like things can lead to trouble. A contributed package that supports integration of functions involving square roots is available, however (ALGINT, chapter 20). In addition there is a definite integration package, DEFINT(chapter 34).

7.4.3 References

A. C. Norman & P. M. A. Moore, "Implementing the New Risch Algorithm", Proc. 4th International Symposium on Advanced Comp. Methods in Theor. Phys., CNRS, Marseilles, 1977.

S. J. Harrington, "A New Symbolic Integration System in Reduce", Comp. Journ. 22 (1979) 2.

A. C. Norman & J. H. Davenport, "Symbolic Integration — The Dust Settles?", Proc. EUROSAM 79, Lecture Notes in Computer Science 72, Springer-Verlag, Berlin Heidelberg New York (1979) 398-407.

7.5 LENGTH Operator

`LENGTH` is a generic operator for finding the length of various objects in the system. The meaning depends on the type of the object. In particular, the length of an algebraic expression is the number of additive top-level terms its expanded representation.

Examples:

```
length(a+b)    ->  2
length(2)      ->  1.
```

Other objects that support a length operator include arrays, lists and matrices. The explicit meaning in these cases is included in the description of these objects.

7.6 MAP Operator

The `MAP` operator applies a uniform evaluation pattern to all members of a composite structure: a matrix, a list, or the arguments of an operator expression. The evaluation pattern can be a unary procedure, an operator, or an algebraic expression with one free variable.

It is used with the syntax:

```
MAP(U:function,V:object)
```

Here `object` is a list, a matrix or an operator expression. `Function` can be one of the following:

1. the name of an operator for a single argument: the operator is evaluated once with each element of `object` as its single argument;
2. an algebraic expression with exactly one free variable, that is a variable preceded by the tilde symbol. The expression is evaluated for each element of `object`, where the element is substituted for the free variable;
3. a replacement rule of the form `var => rep` where `var` is a variable (a kernel without a subscript) and `rep` is an expression that contains `var`. `Rep` is evaluated for each element of `object` where the element is substituted for `var`. `Var` may be optionally preceded by a tilde.

The rule form for `function` is needed when more than one free variable occurs.

Examples:

```
map(abs,{1,-2,a,-a})  ->  {1,2,ABS(A),ABS(A)}
map(int(~w,x), mat((x^2,x^5),(x^4,x^5))) ->

        [  3     6 ]
        [ x     x  ]
        [----  ----]
        [ 3     6  ]
        [          ]
        [  5     6 ]
        [ x     x  ]
        [----  ----]
        [ 5     6  ]

map(~w*6, x^2/3 = y^3/2 -1) -> 2*X^2=3*(Y^3-2)
```

You can use MAP in nested expressions. However, you cannot apply MAP to a non-composed object, e.g. an identifier or a number.

7.7 MKID Operator

In many applications, it is useful to create a set of identifiers for naming objects in a consistent manner. In most cases, it is sufficient to create such names from two components. The operator MKID is provided for this purpose. Its syntax is:

```
MKID(U:id,V:id|non-negative integer):id
```

for example

```
mkid(a,3)       -> A3
mkid(apple,s)   -> APPLES
```

while mkid(a+b,2) gives an error.

The SET operator can be used to give a value to the identifiers created by MKID, for example

```
set(mkid(a,3),3);
```

will give A3 the value 2.

7.8 PF Operator

PF(<exp>,<var>) transforms the expression <exp> into a list of partial fractions with respect to the main variable, <var>. PF does a complete partial fraction decomposition, and as the algorithms used are fairly unsophisticated (factorization and the extended Euclidean algorithm), the code may be unacceptably slow in complicated cases.

Example: Given `2/((x+1)^2*(x+2))` in the workspace, `pf(ws,x);` gives the result

```
       2       - 2           2
   {-------,-------,-------------} .
     X + 2   X + 1    2
                     X  + 2*X + 1
```

If you want the denominators in factored form, use `off exp;`. Thus, with `2/((x+1)^2*(x+2))` in the workspace, the commands `off exp; pf(ws,x);` give the result

```
       2       - 2       2
   {-------,-------,----------} .
     X + 2   X + 1           2
                     (X + 1)
```

To recombine the terms, `FOR EACH ...SUM` can be used. So with the above list in the workspace, `for each j in ws sum j;` returns the result

```
        2
------------------
                 2
 (X + 2)*(X + 1)
```

Alternatively, one can use the operations on lists to extract any desired term.

7.9 SELECT Operator

The `SELECT` operator extracts from a list, or from the arguments of an n–ary operator, elements corresponding to a boolean predicate. It is used with the syntax:

```
SELECT(U:function,V:list)
```

`Function` can be one of the following forms:

1. the name of an operator for a single argument: the operator is evaluated once with each element of `object` as its single argument;
2. an algebraic expression with exactly one free variable, that is a variable preceded by the tilde symbol. The expression is evaluated for each element of ⟨*object*⟩, where the element is substituted for the free variable;
3. a replacement rule of the form ⟨*var* => *rep*⟩ where `var` is a variable (a kernel without subscript) and `rep` is an expression that contains `var`. `Rep` is evaluated for each element of `object` where the element is substituted for `var`. `var` may be optionally preceded by a tilde.

The rule form for `function` is needed when more than one free variable occurs.

The result of evaluating `function` is interpreted as a boolean value corresponding to the conventions of REDUCE. These values are composed with the leading operator of the input expression.

Examples:

```
select( ~w>0 , {1,-1,2,-3,3}) -> {1,2,3}
select(evenp deg(~w,y),part((x+y)^5,0):=list)
       -> {X^5 ,10*X^3*Y^2 ,5*X*Y^4}
select(evenp deg(~w,x),2x^2+3x^3+4x^4) -> 4X^4 + 2X^2
```

7.10 SOLVE Operator

SOLVE is an operator for solving one or more simultaneous algebraic equations. It is used with the syntax:

```
SOLVE(EXPRN:algebraic[,VAR:kernel|,VARLIST:list of kernels])
       :list.
```

`EXPRN` is of the form `<expression>` or { `<expression1>`,`<expression2>`, ...}. Each expression is an algebraic equation, or is the difference of the two sides of the equation. The second argument is either a kernel or a list of kernels representing the unknowns in the system. This argument may be omitted if the number of distinct, non-constant, top-level kernels equals the number of unknowns, in which case these kernels are presumed to be the unknowns.

For one equation, `SOLVE` recursively uses factorization and decomposition, together with the known inverses of `LOG`, `SIN`, `COS`, `^`, `ACOS`, `ASIN`, and linear, quadratic, cubic, quartic, or binomial factors. Solutions of equations built with exponentials or logarithms are often expressed in terms of Lambert's `W` function. This function is (partially) implemented in the special functions package.

Linear equations are solved by the multi-step elimination method due to Bareiss, unless the switch `CRAMER` is on, in which case Cramer's method is used. The Bareiss method is usually more efficient unless the system is large and dense.

Non-linear equations are solved using the Groebner basis package. Users should note that this can be quite a time consuming process.

Examples:

```
solve(log(sin(x+3))^5 = 8,x);
solve(a*log(sin(x+3))^5 - b, sin(x+3));
solve({a*x+y=3,y=-2},{x,y});
```

`SOLVE` returns a list of solutions. If there is one unknown, each solution is an equation for the unknown. If a complete solution was found, the unknown will appear by itself on the left-hand side of the equation. On the other hand, if the solve package could not find a solution, the "solution" will be an equation for the unknown in terms of the operator `ROOT_OF`. If there are several unknowns, each solution will be a list of equations for the unknowns. For example,

```
solve(x^2=1,x);            -> {X=-1,X=1}

solve(x^7-x^6+x^2=1,x)
                                6
       -> {X=ROOT_OF(X_  + X_ + 1,X_,TAG_1),X=1}

solve({x+3y=7,y-x=1},{x,y}) -> {{X=1,Y=2}}.
```

The TAG argument is used to uniquely identify those particular solutions. Solution multiplicities are stored in the global variable `ROOT_MULTIPLICITIES` rather than the solution list. The value of this variable is a list of the multiplicities of the solutions for the last call of `SOLVE`. For example,

```
solve(x^2=2x-1,x); root_multiplicities;
```

gives the results

```
{X=1}

{2}
```

If you want the multiplicities explicitly displayed, the switch MULTIPLICITIES can be turned on. For example

```
on multiplicities; solve(x^2=2x-1,x);
```

yields the result

```
{X=1,X=1}
```

7.10.1 Handling of Undetermined Solutions

When SOLVE cannot find a solution to an equation, it normally returns an equation for the relevant indeterminates in terms of the operator ROOT_OF. For example, the expression

```
solve(cos(x) + log(x),x);
```

returns the result

```
{X=ROOT_OF(COS(X_) + LOG(X_),X_,TAG_1)} .
```

An expression with a top-level ROOT_OF operator is implicitly a list with an unknown number of elements (since we don't always know how many solutions an equation has). If a substitution is made into such an expression, closed form solutions can emerge. If this occurs, the ROOT_OF construct is replaced by an operator ONE_OF. At this point it is of course possible to transform the result of the original SOLVE operator expression into a standard SOLVE solution. To effect this, the operator EXPAND_CASES can be used.

The following example shows the use of these facilities:

```
solve(-a*x^3+a*x^2+x^4-x^3-4*x^2+4,x);
               2      3
{X=ROOT_OF(A*X_  - X_  + 4*X_ + 4,X_,TAG_2),X=1}

sub(a=-1,ws);

{X=ONE_OF({2,-1,-2},TAG_2),X=1}

expand_cases ws;

{X=2,X=-1,X=-2,X=1}
```

7.10.2 Solutions of Equations Involving Cubics and Quartics

Since roots of cubics and quartics can often be very messy, a switch FULLROOTS is available, that, when off (the default), will prevent the production of a result in closed form. The ROOT_OF construct will be used in this case instead.

In constructing the solutions of cubics and quartics, trigonometrical forms are used where appropriate. This option is under the control of a switch TRIGFORM, which is normally on.

The following example illustrates the use of these facilities:

```
let xx = solve(x^3+x+1,x);

xx;
            3
{X=ROOT_OF(X_  + X_ + 1,X_)}

on fullroots;

xx;

                              - SQRT(31)*I
                       ATAN(---------------)
                                3*SQRT(3)
{X=(I*(SQRT(3)*SIN(-----------------------)
                                  3
```

```
                  - SQRT(31)*I
           ATAN(---------------)
                   3*SQRT(3)
      - COS(-----------------------)))/SQRT(3),
                      3

                          - SQRT(31)*I
                   ATAN(---------------)
                           3*SQRT(3)
 X=( - I*(SQRT(3)*SIN(-----------------------)
                              3

                     - SQRT(31)*I
              ATAN(---------------)
                      3*SQRT(3)
         + COS(-----------------------)))/SQRT(
                         3

     3),

                 - SQRT(31)*I
          ATAN(---------------)
                  3*SQRT(3)
    2*COS(-----------------------)*I
                     3
 X=----------------------------------}
               SQRT(3)

off trigform;

xx;
                             2/3
{X=( - (SQRT(31) - 3*SQRT(3))   *SQRT(3)*I

                             2/3    2/3
     - (SQRT(31) - 3*SQRT(3))    - 2   *SQRT(3)*I

        2/3                          1/3  1/3
     + 2   )/(2*(SQRT(31) - 3*SQRT(3))   *6

               1/6
              *3   ),

                          2/3
 X=((SQRT(31) - 3*SQRT(3))   *SQRT(3)*I
```

```
                            2/3    2/3
 - (SQRT(31) - 3*SQRT(3))    + 2   *SQRT(3)*I

    2/3                          1/3  1/3
 + 2   )/(2*(SQRT(31) - 3*SQRT(3))   *6

            1/6
         *3   ),

                          2/3    2/3
  (SQRT(31) - 3*SQRT(3))    - 2
X=------------------------------------}
                         1/3  1/3  1/6
 (SQRT(31) - 3*SQRT(3))   *6   *3
```

7.10.3 Other Options

If `SOLVESINGULAR` is on (the default setting), degenerate systems such as `x+y=0`, `2x+2y=0` will be solved by introducing appropriate arbitrary constants. The consistent singular equation 0=0 or equations involving functions with multiple inverses may introduce unique new indeterminant kernels `ARBCOMPLEX(j)`, or `ARBINT(j)`, (j=1,2,...), representing arbitrary complex or integer numbers respectively. To automatically select the principal branches, do `off allbranch;` . To avoid the introduction of new indeterminant kernels do `OFF ARBVARS` – then no equations are generated for the free variables and their original names are used to express the solution forms. To suppress solutions of consistent singular equations do `OFF SOLVESINGULAR`.

To incorporate additional inverse functions do, for example:

```
put('sinh,'inverse,'asinh);
put('asinh,'inverse,'sinh);
```

together with any desired simplification rules such as

```
for all x let sinh(asinh(x))=x, asinh(sinh(x))=x;
```

For completeness, functions with non-unique inverses should be treated as `^`, `SIN`, and `COS` are in the `SOLVE` module source.

Arguments of `ASIN` and `ACOS` are not checked to ensure that the absolute value of the real part does not exceed 1; and arguments of LOG are not checked to ensure that the absolute value of the imaginary part does not exceed π; but checks (perhaps involving user response for non-numerical arguments) could be introduced using LET statements for these operators.

7.10.4 Parameters and Variable Dependency

The proper design of a variable sequence supplied as a second argument to SOLVE is important for the structure of the solution of an equation system. Any unknown in the system not in this list is considered totally free. E.g. the call

```
solve({x=2*z,z=2*y},{z});
```

produces an empty list as a result because there is no function $z = z(x, y)$ which fulfills both equations for arbitrary x and y values. In such a case the share variable requirements displays a set of restrictions for the parameters of the system:

```
requirements;

{x - 4*y}
```

The non-existence of a formal solution is caused by a contradiction which disappears only if the parameters of the initial system are set such that all members of the requirements list take the value zero. For a linear system the set is complete: a solution of the requirements list makes the initial system solvable. E.g. in the above case a substitution $x = 4y$ makes the equation set consistent. For a non-linear system only one inconsistency is detected. If such a system has more than one inconsistency, you must reduce them one after the other. [1] The set shows you also the dependency among the parameters: here one of x and y is free and a formal solution of the system can be computed by adding it to the variable list of solve. The requirement set is not unique – there may be other such sets.

A system with parameters may have a formal solution, e.g.

```
solve({x=a*z+1,0=b*z-y},{z,x});
```

[1] The difference between linear and non–linear inconsistent systems is based on the algorithms which produce this information as a side effect when attempting to find a formal solution; example: $solve(\{x = a, x = b, y = c, y = d\}, \{x, y\}$ gives a set $\{a-b, c-d\}$ while $solve(\{x^2 = a, x^2 = b, y^2 = c, y^2 = d\}, \{x, y\}$ leads to $\{a - b\}$.

```
       y      a*y + b
{{z=---,x=---------}}
       b         b
```

which is not valid for all possible values of the parameters. The variable `assumptions` contains then a list of restrictions: the solutions are valid only as long as none of these expressions vanishes. Any zero of one of them represents a special case that is not covered by the formal solution. In the above case the value is

```
assumptions;

{b}
```

which excludes formally the case $b = 0$; obviously this special parameter value makes the system singular. The set of assumptions is complete for both, linear and non–linear systems.

SOLVE rearranges the variable sequence to reduce the (expected) computing time. This behavior is controlled by the switch `varopt`, which is on by default. If it is turned off, the supplied variable sequence is used or the system kernel ordering is taken if the variable list is omitted. The effect is demonstrated by an example:

```
s:= {y^3+3x=0,x^2+y^2=1};

solve(s,{y,x});

               6       2
{{y=root_of(y_  + 9*y_  - 9,y_),

          3
       - y
  x=-------}}
        3

off varopt; solve(s,{y,x});

               6       4        2
{{x=root_of(x_  - 3*x_  + 12*x_  - 1,x_),

             4      2
     x*( - x  + 2*x  - 10)
  y=-----------------------}}
               3
```

In the first case, `solve` forms the solution as a set of pairs $(y_i, x(y_i))$ because the degree of x is higher – such a rearrangement makes the internal computation of the Gröbner basis generally faster. For the second case the explicitly given variable sequence is used such that the solution has now the form $(x_i, y(x_i))$. Controlling the variable sequence is especially important if the system has one or more free variables. As an alternative to turning off `varopt`, a partial dependency among the variables can be declared using the `depend` statement: `solve` then rearranges the variable sequence but keeps any variable ahead of those on which it depends.

```
on varopt;
s:={a^3+b,b^2+c}$
solve(s,{a,b,c});

                        3       6
{{a=arbcomplex(1),b= - a ,c= - a }}

depend a,c; depend b,c; solve(s,{a,b,c});

{{c=arbcomplex(2),

              6
  a=root_of(a_  + c,a_),

        3
  b= - a }}
```

Here `solve` is forced to put c after a and after b, but there is no obstacle to interchanging a and b.

7.11 Even and Odd Operators

An operator can be declared to be *even* or *odd* in its first argument by the declarations `EVEN` and `ODD` respectively. Expressions involving an operator declared in this manner are transformed if the first argument contains a minus sign. Any other arguments are not affected. In addition, if say `F` is declared odd, then `f(0)` is replaced by zero unless `F` is also declared *non zero* by the declaration `NONZERO`. For example, the declarations

```
even f1; odd f2;
```

mean that

```
f1(-a)    ->    F1(A)
f2(-a)    ->   -F2(A)
f1(-a,-b) ->    F1(A,-B)
f2(0)     ->    0.
```

To inhibit the last transformation, say `nonzero f2;`.

7.12 Linear Operators

An operator can be declared to be linear in its first argument over powers of its second argument. If an operator F is so declared, F of any sum is broken up into sums of Fs, and any factors that are not powers of the variable are taken outside. This means that F must have (at least) two arguments. In addition, the second argument must be an identifier (or more generally a kernel), not an expression.

Example:

If F were declared linear, then

```
                                   5
f(a*x^5+b*x+c,x) ->  F(X ,X)*A + F(X,X)*B + F(1,X)*C
```

More precisely, not only will the variable and its powers remain within the scope of the F operator, but so will any variable and its powers that had been declared to DEPEND on the prescribed variable; and so would any expression that contains that variable or a dependent variable on any level, e.g. cos(sin(x)).

To declare operators F and G to be linear operators, use:

```
linear f,g;
```

The analysis is done of the first argument with respect to the second; any other arguments are ignored. It uses the following rules of evaluation:

```
f(0) -> 0
f(-y,x) -> -F(Y,X)
f(y+z,x) -> F(Y,X)+F(Z,X)
f(y*z,x) -> Z*F(Y,X)        if Z does not depend on X
f(y/z,x) -> F(Y,X)/Z        if Z does not depend on X
```

To summarize, Y "depends" on the indeterminate X in the above if either of the following hold:

1. Y is an expression that contains X at any level as a variable, e.g.: cos(sin(x))
2. Any variable in the expression Y has been declared dependent on X by use of the declaration DEPEND.

The use of such linear operators can be seen in the paper Fox, J.A. and A. C. Hearn, "Analytic Computation of Some Integrals in Fourth Order Quantum Electrodynamics" Journ. Comp. Phys. 14 (1974) 301-317, which contains a complete listing of a program for definite integration of some expressions that arise in fourth order quantum electrodynamics.

7.13 Non-Commuting Operators

An operator can be declared to be non-commutative under multiplication by the declaration `NONCOM`.

Example:

After the declaration
`noncom u,v;`
the expressions `u(x)*u(y)-u(y)*u(x)` and `u(x)*v(y)-v(y)*u(x)` will remain unchanged on simplification, and in particular will not simplify to zero.

Note that it is the operator (`U` and `V` in the above example) and not the variable that has the non-commutative property.

The `LET` statement may be used to introduce rules of evaluation for such operators. In particular, the boolean operator `ORDP` is useful for introducing an ordering on such expressions.

Example:

The rule

```
for all x,y such that x neq y and ordp(x,y)
   let u(x)*u(y)= u(y)*u(x)+comm(x,y);
```

would introduce the commutator of `u(x)` and `u(y)` for all `X` and `Y`. Note that since `ordp(x,x)` is *true*, the equality check is necessary in the degenerate case to avoid a circular loop in the rule.

7.14 Symmetric and Antisymmetric Operators

An operator can be declared to be symmetric with respect to its arguments by the declaration `SYMMETRIC`. For example

```
symmetric u,v;
```

means that any expression involving the top level operators `U` or `V` will have its arguments reordered to conform to the internal order used by REDUCE. The user can change this order for kernels by the command `KORDER`.

For example, `u(x,v(1,2))` would become `u(v(2,1),x)`, since numbers are ordered in decreasing order, and expressions are ordered in decreasing order of complexity.

Similarly the declaration `ANTISYMMETRIC` declares an operator antisymmetric. For example,

```
antisymmetric l,m;
```

means that any expression involving the top level operators `L` or `M` will have its arguments reordered to conform to the internal order of the system, and the sign of the expression changed if there are an odd number of argument interchanges necessary to bring about the new order.

For example, `l(x,m(1,2))` would become `-l(-m(2,1),x)` since one interchange occurs with each operator. An expression like `l(x,x)` would also be replaced by 0.

7.15 Declaring New Prefix Operators

The user may add new prefix operators to the system by using the declaration `OPERATOR`. For example:

```
operator h,g1,arctan;
```

adds the prefix operators `H`, `G1` and `ARCTAN` to the system.

This allows symbols like `h(w), h(x,y,z), g1(p+q), arctan(u/v)` to be used in expressions, but no meaning or properties of the operator are implied. The same operator symbol can be used equally well as a 0-, 1-, 2-, 3-, etc.-place operator.

To give a meaning to an operator symbol, or express some of its properties, `LET` statements can be used, or the operator can be given a definition as a procedure.

If the user forgets to declare an identifier as an operator, the system will prompt the user to do so in interactive mode, or do it automatically in non-interactive mode. A diagnostic message will also be printed if an identifier is declared `OPERATOR` more than once.

Operators once declared are global in scope, and so can then be referenced anywhere in the program. In other words, a declaration within a block (or a procedure) does not limit the scope of the operator to that block, nor does the operator go away on exiting the block (use `CLEAR` instead for this purpose).

7.16 Declaring New Infix Operators

Users can add new infix operators by using the declarations `INFIX` and `PRECEDENCE`. For example,

```
infix mm;
precedence mm,-;
```

The declaration `infix mm;` would allow one to use the symbol `MM` as an infix operator:

`a mm b` instead of `mm(a,b)`.

The declaration `precedence mm,-;` says that `MM` should be inserted into the infix operator precedence list just *after* the − operator. This gives it higher precedence than − and lower precedence than * . Thus

`a - b mm c - d` means `a - (b mm c) - d`,

while

`a * b mm c * d` means `(a * b) mm (c * d)`.

Both infix and prefix operators have no transformation properties unless `LET` statements or procedure declarations are used to assign a meaning.

We should note here that infix operators so defined are always binary:

`a mm b mm c` means `(a mm b) mm c`.

7.17 Creating/Removing Variable Dependency

There are several facilities in REDUCE, such as the differentiation operator and the linear operator facility, that can utilize knowledge of the dependency between various variables, or kernels. Such dependency may be expressed by the command `DEPEND`. This takes an arbitrary number of arguments and sets up a dependency of the first argument on the remaining arguments. For example,

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

says that `X` is dependent on both `Y` and `Z`.

```
depend z,cos(x),y;
```

says that `Z` is dependent on `COS(X)` and `Y`.

Dependencies introduced by `DEPEND` can be removed by `NODEPEND`. The arguments of this are the same as for `DEPEND`. For example, given the above dependencies,

```
nodepend z,cos(x);
```

says that `Z` is no longer dependent on `COS(X)`, although it remains dependent on `Y`.

Chapter 8

Display and Structuring of Expressions

In this section, we consider a variety of commands and operators that permit the user to obtain various parts of algebraic expressions and also display their structure in a variety of forms. Also presented are some additional concepts in the REDUCE design that help the user gain a better understanding of the structure of the system.

8.1 Kernels

REDUCE is designed so that each operator in the system has an evaluation (or simplification) function associated with it that transforms the expression into an internal canonical form. This form, which bears little resemblance to the original expression, is described in detail in Hearn, A. C., "REDUCE 2: A System and Language for Algebraic Manipulation," Proc. of the Second Symposium on Symbolic and Algebraic Manipulation, ACM, New York (1971) 128-133.

The evaluation function may transform its arguments in one of two alternative ways. First, it may convert the expression into other operators in the system, leaving no functions of the original operator for further manipulation. This is in a sense true of the evaluation functions associated with the operators `+`, `*` and `/` , for example, because the canonical form does not include these operators explicitly. It is also true of an operator such as the determinant operator `DET` because the relevant evaluation function calculates the appropriate determinant, and the operator `DET` no longer appears. On the other hand, the evaluation process may leave some residual functions of the relevant operator. For example, with the operator `COS`, a residual expression like `COS(X)` may remain after evaluation unless a rule for the reduction of cosines into exponentials, for example, were introduced. These residual functions of an operator are termed *kernels* and are stored uniquely like variables. Subsequently, the kernel is carried through the calculation as a variable unless transformations are introduced for the operator at a later stage.

In those cases where the evaluation process leaves an operator expression with non-trivial arguments, the form of the argument can vary depending on the state of the system at the point of evaluation. Such arguments are normally produced in expanded form with no terms factored or grouped in any way. For example, the expression `cos(2*x+2*y)` will normally be returned in the same form. If the argument `2*x+2*y` were evaluated at the top level, however, it would be printed as `2*(X+Y)`. If it is desirable to have the arguments themselves in a similar form, the switch `INTSTR` (for “internal structure”), if on, will cause this to happen.

In cases where the arguments of the kernel operators may be reordered, the system puts them in a canonical order, based on an internal intrinsic ordering of the variables. However, some commands allow arguments in the form of kernels, and the user has no way of telling what internal order the system will assign to these arguments. To resolve this difficulty, we introduce the notion of a *kernel form* as an expression that transforms to a kernel on evaluation.

Examples of kernel forms are:

```
a
cos(x*y)
log(sin(x))
```

whereas

```
a*b
(a+b)^4
```

are not.

We see that kernel forms can usually be used as generalized variables, and most algebraic properties associated with variables may also be associated with kernels.

8.2 The Expression Workspace

Several mechanisms are available for saving and retrieving previously evaluated expressions. The simplest of these refers to the last algebraic expression simplified. When an assignment of an algebraic expression is made, or an expression is evaluated at the top level, (i.e., not inside a compound statement or procedure) the results of the evaluation are automatically saved in a variable `WS` that we shall refer to as the workspace. (More precisely, the expression is assigned to the variable `WS` that is then available for further manipulation.)

Example:

If we evaluate the expression `(x+y)^2` at the top level and next wish to differentiate it with respect to `Y`, we can simply say

```
df(ws,y);
```

to get the desired answer.

If the user wishes to assign the workspace to a variable or expression for later use, the `SAVEAS` statement can be used. It has the syntax

```
SAVEAS <expression>
```

For example, after the differentiation in the last example, the workspace holds the expression `2*x+2*y`. If we wish to assign this to the variable `Z` we can now say

```
saveas z;
```

If the user wishes to save the expression in a form that allows him to use some of its variables as arbitrary parameters, the `FOR ALL` command can be used.

Example:

```
for all x saveas h(x);
```

with the above expression would mean that `h(z)` evaluates to `2*Y+2*Z`.

A further method for referencing more than the last expression is described in the section on interactive use of REDUCE.

8.3 Output of Expressions

A considerable degree of flexibility is available in REDUCE in the printing of expressions generated during calculations. No explicit format statements are supplied, as these are in most cases of little use in algebraic calculations, where the size of output or its composition is not generally known in advance. Instead, REDUCE provides a series of mode options to the user that should enable him to produce his output in a comprehensible and possibly pleasing form.

The most extreme option offered is to suppress the output entirely from any top level evaluation. This is accomplished by turning off the switch `OUTPUT` which is normally on. It is useful for limiting output when loading large files or producing "clean" output from the prettyprint programs.

In most circumstances, however, we wish to view the output, so we need to know how to format it appropriately. As we mentioned earlier, an algebraic expression is normally printed in an expanded form, filling the whole output line with terms. Certain output declarations, however, can be used to affect this format. To begin with, we look at an operator for changing the length of the output line.

8.3.1 LINELENGTH Operator

This operator is used with the syntax

```
LINELENGTH(NUM:integer):integer
```

and sets the output line length to the integer `NUM`. It returns the previous output line length (so that it can be stored for later resetting of the output line if needed).

8.3.2 Output Declarations

We now describe a number of switches and declarations that are available for controlling output formats. It should be noted, however, that the transformation of large expressions to produce these varied output formats can take a lot of computing time and space. If a user wishes to speed up the printing of the output in such cases, he can turn off the switch `PRI`. If this is done, then output is produced in one fixed format, which basically reflects the internal form of the expression, and none of the options below apply. `PRI` is normally on.

With `PRI` on, the output declarations and switches available are as follows:

ORDER Declaration

The declaration `ORDER` may be used to order variables on output. The syntax is:

```
order v1,...vn;
```

where the `vi` are kernels. Thus,

```
order x,y,z;
```

orders `X` ahead of `Y`, `Y` ahead of `Z` and all three ahead of other variables not given an order. `order nil;` resets the output order to the system default. The order of variables may be changed by further calls of `ORDER`, but then the reordered variables would have an order lower than those in earlier `ORDER` calls. Thus,

```
order x,y,z;
order y,x;
```

would order `Z` ahead of `Y` and `X`. The default ordering is usually alphabetic.

FACTOR Declaration

This declaration takes a list of identifiers or kernels as argument. `FACTOR` is not a factoring command (use `FACTORIZE` or the `FACTOR` switch for this purpose); rather it is a separation command. All terms involving fixed powers of the declared expressions are printed as a product of the fixed powers and a sum of the rest of the terms.

All expressions involving a given prefix operator may also be factored by putting the operator name in the list of factored identifiers. For example:

```
factor x,cos,sin(x);
```

causes all powers of `X` and `SIN(X)` and all functions of `COS` to be factored.

Note that `FACTOR` does not affect the order of its arguments. You should also use `ORDER` if this is important.

The declaration `remfac v1,...,vn;` removes the factoring flag from the expressions `v1` through `vn`.

8.3.3 Output Control Switches

In addition to these declarations, the form of the output can be modified by switching various output control switches using the declarations `ON` and `OFF`. We shall illustrate the use of these switches by an example, namely the printing of the expression

```
x^2*(y^2+2*y)+x*(y^2+z)/(2*a) .
```

The relevant switches are as follows:

ALLFAC Switch

This switch will cause the system to search the whole expression, or any sub-expression enclosed in parentheses, for simple multiplicative factors and print them outside the parentheses. Thus our expression with `ALLFAC` off will print as

```
    2  2         2           2
(2*X *Y *A + 4*X *Y*A + X*Y  + X*Z)/(2*A)
```

and with `ALLFAC` on as

```
         2                 2
X*(2*X*Y *A + 4*X*Y*A + Y  + Z)/(2*A) .
```

`ALLFAC` is normally on, and is on in the following examples, except where otherwise stated.

DIV Switch

This switch makes the system search the denominator of an expression for simple factors that it divides into the numerator, so that rational fractions and negative powers appear in the output. With `DIV` on, our expression would print as

```
      2                2  (-1)         (-1)
X*(X*Y  + 2*X*Y + 1/2*Y *A     + 1/2*A    *Z) .
```

`DIV` is normally off.

LIST Switch

This switch causes the system to print each term in any sum on a separate line. With `LIST` on, our expression prints as

```
        2
X*(2*X*Y *A

    + 4*X*Y*A

       2
    + Y

    + Z)/(2*A) .
```

`LIST` is normally off.

NOSPLIT Switch

Under normal circumstances, the printing routines try to break an expression across lines at a natural point. This is a fairly expensive process. If you are not overly concerned about where the end-of-line breaks come, you can speed up the printing of expressions by turning off the switch `NOSPLIT`. This switch is normally on.

RAT Switch

This switch is only useful with expressions in which variables are factored with `FACTOR`. With this mode, the overall denominator of the expression is printed with each factored sub-expression. We assume a prior declaration `factor x;` in the following output. We first print the expression with `RAT` `off`:

```
    2                    2
(2*X *Y*A*(Y + 2) + X*(Y  + Z))/(2*A) .
```

With `RAT` on the output becomes:

```
 2                 2
X *Y*(Y + 2) + X*(Y  + Z)/(2*A) .
```

RAT is normally off.

Next, if we leave X factored, and turn on both DIV and RAT, the result becomes

```
 2                      (-1)   2
X *Y*(Y + 2) + 1/2*X*A    *(Y  + Z) .
```

Finally, with X factored, RAT on and ALLFAC off we retrieve the original structure

```
 2   2              2
X *(Y  + 2*Y) + X*(Y  + Z)/(2*A) .
```

RATPRI Switch

If the numerator and denominator of an expression can each be printed in one line, the output routines will print them in a two dimensional notation, with numerator and denominator on separate lines and a line of dashes in between. For example, (a+b)/2 will print as

```
A + B
-----
  2
```

Turning this switch off causes such expressions to be output in a linear form.

REVPRI Switch

The normal ordering of terms in output is from highest to lowest power. In some situations (e.g., when a power series is output), the opposite ordering is more convenient. The switch REVPRI if on causes such a reverse ordering of terms. For example, the expression y*(x+1)^2+(y+3)^2 will normally print as

```
 2              2
X *Y + 2*X*Y + Y  + 7*Y + 9
```

whereas with REVPRI on, it will print as

```
               2              2
9 + 7*Y + Y  + 2*X*Y + X *Y.
```

8.3.4 WRITE Command

In simple cases no explicit output command is necessary in REDUCE, since the value of any expression is automatically printed if a semicolon is used as a delimiter. There are, however, several situations in which such a command is useful.

In a `FOR`, `WHILE`, or `REPEAT` statement it may be desired to output something each time the statement within the loop construct is repeated.

It may be desired for a procedure to output intermediate results or other information while it is running. It may be desired to have results labeled in special ways, especially if the output is directed to a file or device other than the terminal.

The `WRITE` command consists of the word `WRITE` followed by one or more items separated by commas, and followed by a terminator. There are three kinds of items that can be used:

1. Expressions (including variables and constants). The expression is evaluated, and the result is printed out.
2. Assignments. The expression on the right side of the `:=` operator is evaluated, and is assigned to the variable on the left; then the symbol on the left is printed, followed by a “`:=`”, followed by the value of the expression on the right – almost exactly the way an assignment followed by a semicolon prints out normally. (The difference is that if the `WRITE` is in a `FOR` statement and the left-hand side of the assignment is an array position or something similar containing the variable of the `FOR` iteration, then the value of that variable is inserted in the printout.)
3. Arbitrary strings of characters, preceded and followed by double-quote marks (e.g., `"string"`).

The items specified by a single `WRITE` statement print side by side on one line. (The line is broken automatically if it is too long.) Strings print exactly as quoted. The `WRITE` command itself however does not return a value.

The print line is closed at the end of a `WRITE` command evaluation. Therefore the command `WRITE "";` (specifying nothing to be printed except the empty string) causes a line to be skipped.

Examples:

1. If `A` is `X+5`, `B` is itself, `C` is 123, `M` is an array, and `Q`=3, then

   ```
   write m(q):=a," ",b/c," THANK YOU";
   ```

   will set `M(3)` to `x+5` and print

   ```
   M(Q) := X + 5 B/123 THANK YOU
   ```

   The blanks between the `5` and `B`, and the `3` and `T`, come from the blanks in the quoted strings.

2. To print a table of the squares of the integers from 1 to 20:

   ```
   for i:=1:20 do write i," ",i^2;
   ```

3. To print a table of the squares of the integers from 1 to 20, and at the same time store them in positions 1 to 20 of an array `A`:

   ```
   for i:=1:20 do <<a(i):=i^2; write i," ",a(i)>>;
   ```

   This will give us two columns of numbers. If we had used

   ```
   for i:=1:20 do write i," ",a(i):=i^2;
   ```

   we would also get `A(i) :=` repeated on each line.

4. The following more complete example calculates the famous f and g series, first reported in Sconzo, P., LeSchack, A. R., and Tobey, R., "Symbolic Computation of f and g Series by Computer", Astronomical Journal 70 (May 1965).

   ```
   x1:= -sig*(mu+2*eps)$
   x2:= eps - 2*sig^2$
   x3:= -3*mu*sig$
   f:= 1$
   g:= 0$
   for i:= 1 step 1 until 10 do begin
      f1:= -mu*g+x1*df(f,eps)+x2*df(f,sig)+x3*df(f,mu);
   ```

```
  write "f(",i,") := ",f1;
  g1:= f+x1*df(g,eps)+x2*df(g,sig)+x3*df(g,mu);
  write "g(",i,") := ",g1;
  f:=f1$
  g:=g1$
 end;
```

A portion of the output, to illustrate the printout from the WRITE command, is as follows:

```
                ... <prior output> ...

                                2
 F(4) := MU*(3*EPS - 15*SIG  + MU)

 G(4) := 6*SIG*MU

                                         2
 F(5) := 15*SIG*MU*( - 3*EPS + 7*SIG  - MU)

                                2
 G(5) := MU*(9*EPS - 45*SIG  + MU)

                ... <more output> ...
```

8.3.5 Suppression of Zeros

It is sometimes annoying to have zero assignments (i.e. assignments of the form `<expression> := 0`) printed, especially in printing large arrays with many zero elements. The output from such assignments can be suppressed by turning on the switch NERO.

8.3.6 FORTRAN Style Output Of Expressions

It is naturally possible to evaluate expressions numerically in REDUCE by giving all variables and sub-expressions numerical values. However, as we pointed out elsewhere the user must declare real arithmetical operation by turning on the switch ROUNDED. However, it should be remembered that arithmetic in REDUCE is not particularly fast, since results are interpreted rather than evaluated in a compiled form. The user with a large amount of numerical computation after all necessary algebraic manipulations have been performed is therefore well advised to perform these calculations in a FORTRAN or similar system. For this purpose, REDUCE offers facilities for users to produce FORTRAN compatible files for numerical processing.

First, when the switch `FORT` is on, the system will print expressions in a FORTRAN notation. Expressions begin in column seven. If an expression extends over one line, a continuation mark (.) followed by a blank appears on subsequent cards. After a certain number of lines have been produced (according to the value of the variable `CARD_NO`), a new expression is started. If the expression printed arises from an assignment to a variable, the variable is printed as the name of the expression. Otherwise the expression is given the default name `ANS`. An error occurs if identifiers or numbers are outside the bounds permitted by FORTRAN.

A second option is to use the `WRITE` command to produce other programs.

Example:

The following REDUCE statements

```
 on fort;
 out "forfil";
 write "C     this is a fortran program";
 write " 1    format(e13.5)";
 write "      u=1.23";
 write "      v=2.17";
 write "      w=5.2";
 x:=(u+v+w)^11;
 write "C     it was foolish to expand this expression";
 write "      print 1,x";
 write "      end";
 shut "forfil";
 off fort;
```

will generate a file `forfil` that contains:

```
c this is a fortran program
 1    format(e13.5)
      u=1.23
      v=2.17
      w=5.2
      ans1=1320.*u**3*v*w**7+165.*u**3*w**8+55.*u**2*v**9+495.*u
     . **2*v**8*w+1980.*u**2*v**7*w**2+4620.*u**2*v**6*w**3+
```

```
     . 6930.*u**2*v**5*w**4+6930.*u**2*v**4*w**5+4620.*u**2*v**3*
     . w**6+1980.*u**2*v**2*w**7+495.*u**2*v*w**8+55.*u**2*w**9+
     . 11.*u*v**10+110.*u*v**9*w+495.*u*v**8*w**2+1320.*u*v**7*w
     . **3+2310.*u*v**6*w**4+2772.*u*v**5*w**5+2310.*u*v**4*w**6
     . +1320.*u*v**3*w**7+495.*u*v**2*w**8+110.*u*v*w**9+11.*u*w
     . **10+v**11+11.*v**10*w+55.*v**9*w**2+165.*v**8*w**3+330.*
     . v**7*w**4+462.*v**6*w**5+462.*v**5*w**6+330.*v**4*w**7+
     . 165.*v**3*w**8+55.*v**2*w**9+11.*v*w**10+w**11
      x=u**11+11.*u**10*v+11.*u**10*w+55.*u**9*v**2+110.*u**9*v*
     . w+55.*u**9*w**2+165.*u**8*v**3+495.*u**8*v**2*w+495.*u**8
     . *v*w**2+165.*u**8*w**3+330.*u**7*v**4+1320.*u**7*v**3*w+
     . 1980.*u**7*v**2*w**2+1320.*u**7*v*w**3+330.*u**7*w**4+462.
     . *u**6*v**5+2310.*u**6*v**4*w+4620.*u**6*v**3*w**2+4620.*u
     . **6*v**2*w**3+2310.*u**6*v*w**4+462.*u**6*w**5+462.*u**5*
     . v**6+2772.*u**5*v**5*w+6930.*u**5*v**4*w**2+9240.*u**5*v
     . **3*w**3+6930.*u**5*v**2*w**4+2772.*u**5*v*w**5+462.*u**5
     . *w**6+330.*u**4*v**7+2310.*u**4*v**6*w+6930.*u**4*v**5*w
     . **2+11550.*u**4*v**4*w**3+11550.*u**4*v**3*w**4+6930.*u**
     . 4*v**2*w**5+2310.*u**4*v*w**6+330.*u**4*w**7+165.*u**3*v
     . **8+1320.*u**3*v**7*w+4620.*u**3*v**6*w**2+9240.*u**3*v**
     . 5*w**3+11550.*u**3*v**4*w**4+9240.*u**3*v**3*w**5+4620.*u
     . **3*v**2*w**6+ans1
c     it was foolish to expand this expression
      print 1,x
      end
```

If the arguments of a WRITE statement include an expression that requires continuation records, the output will need editing, since the output routine prints the arguments of WRITE sequentially, and the continuation mechanism therefore generates its auxiliary variables after the preceding expression has been printed.

Finally, since there is no direct analog of *list* in FORTRAN, a comment line of the form

```
c ***** invalid fortran construct (list) not printed
```

will be printed if you try to print a list with FORT on.

FORTRAN Output Options

There are a number of methods available to change the default format of the FORTRAN output.

The breakup of the expression into subparts is such that the number of continuation lines produced is less than a given number. This number can be modified by the assignment

```
card_no := <number>;
```

where `<number>` is the *total* number of cards allowed in a statement. The default value of `CARD_NO` is 20.

The width of the output expression is also adjustable by the assignment

```
fort_width := <integer>;
```

which sets the total width of a given line to `<integer>`. The initial FORTRAN output width is 70.

REDUCE automatically inserts a decimal point after each isolated integer coefficient in a FORTRAN expression (so that, for example, 4 becomes `4.`). To prevent this, set the `PERIOD` mode switch to `OFF`.

FORTRAN output is normally produced in lower case. If upper case is desired, the switch `FORTUPPER` should be turned on.

Finally, the default name `ANS` assigned to an unnamed expression and its subparts can be changed by the operator `VARNAME`. This takes a single identifier as argument, which then replaces `ANS` as the expression name. The value of `VARNAME` is its argument.

Further facilities for the production of FORTRAN and other language output are provided by the SCOPE and GENTRAN packagesdescribed in chapters 42 and 73.

8.3.7 Saving Expressions for Later Use as Input

It is often useful to save an expression on an external file for use later as input in further calculations. The commands for opening and closing output files are explained elsewhere. However, we see in the examples on output of expressions that the standard "natural" method of printing expressions is not compatible with the input syntax. So to print the expression in an input compatible form we must inhibit this natural style by turning off the switch `NAT`. If this is done, a dollar sign will also be printed at the end of the expression.

Example:

The following sequence of commands

```
off nat; out "out"; x := (y+z)^2; write "end";
shut "out"; on nat;
```

will generate a file `out` that contains

```
X := Y**2 + 2*Y*Z + Z**2$
END$
```

8.3.8 Displaying Expression Structure

In those cases where the final result has a complicated form, it is often convenient to display the skeletal structure of the answer. The operator `STRUCTR`, that takes a single expression as argument, will do this for you. Its syntax is:

```
STRUCTR(EXPRN:algebraic[,ID1:identifier[,ID2:identifier]]);
```

The structure is printed effectively as a tree, in which the subparts are laid out with auxiliary names. If the optional `ID1` is absent, the auxiliary names are prefixed by the root `ANS`. This root may be changed by the operator `VARNAME`. If the optional `ID1` is present, and is an array name, the subparts are named as elements of that array, otherwise `ID1` is used as the root prefix. (The second optional argument `ID2` is explained later.)

The `EXPRN` can be either a scalar or a matrix expression. Use of any other will result in an error.

Example:

Let us suppose that the workspace contains `((A+B)^2+C)^3+D`. Then the input `STRUCTR WS;` will (with `EXP` off) result in the output:

```
ANS3

   where

                      3
      ANS3 := ANS2  + D

                      2
      ANS2 := ANS1  + C

      ANS1 := A + B
```

The workspace remains unchanged after this operation, since STRUCTR in the default situation returns no value (if STRUCTR is used as a sub-expression, its value is taken to be 0). In addition, the sub-expressions are normally only displayed and not retained. If you wish to access the sub-expressions with their displayed names, the switch SAVESTRUCTR should be turned on. In this case, STRUCTR returns a list whose first element is a representation for the expression, and subsequent elements are the sub-expressions relations. Thus, with SAVESTRUCTR on, STRUCTR WS in the above example would return

```
                   3               2
{ANS3,ANS3=ANS2  + D,ANS2=ANS1  + C,ANS1=A + B}
```

The PART operator can be used to retrieve the required parts of the expression. For example, to get the value of ANS2 in the above, one could say:

```
part(ws,3,2);
```

If FORT is on, then the results are printed in the reverse order; the algorithm in fact guaranteeing that no sub-expression will be referenced before it is defined. The second optional argument ID2 may also be used in this case to name the actual expression (or expressions in the case of a matrix argument).

Example:

Let us suppose that M, a 2 by 1 matrix, contains the elements `((a+b)^2 + c)^3 + d` and `(a + b)*(c + d)` respectively, and that V has been declared to be an array. With EXP off and FORT on, the statement `structr(2*m,v,k);` will result in the output

```
V(1)=A+B
V(2)=V(1)**2+C
V(3)=V(2)**3+D
```

```
V(4)=C+D
K(1,1)=2.*V(3)
K(2,1)=2.*V(1)*V(4)
```

8.4 Changing the Internal Order of Variables

The internal ordering of variables (more specifically kernels) can have a significant effect on the space and time associated with a calculation. In its default state, REDUCE uses a specific order for this which may vary between sessions. However, it is possible for the user to change this internal order by means of the declaration `KORDER`. The syntax for this is:

```
korder v1,...,vn;
```

where the `Vi` are kernels. With this declaration, the `Vi` are ordered internally ahead of any other kernels in the system. `V1` has the highest order, `V2` the next highest, and so on. A further call of `KORDER` replaces a previous one. `KORDER NIL;` resets the internal order to the system default.

Unlike the `ORDER` declaration, that has a purely cosmetic effect on the way results are printed, the use of `KORDER` can have a significant effect on computation time. In critical cases then, the user can experiment with the ordering of the variables used to determine the optimum set for a given problem.

8.5 Obtaining Parts of Algebraic Expressions

There are many occasions where it is desirable to obtain a specific part of an expression, or even change such a part to another expression. A number of operators are available in REDUCE for this purpose, and will be described in this section. In addition, operators for obtaining specific parts of polynomials and rational functions (such as a denominator) are described in another section.

8.5.1 COEFF Operator

Syntax:

```
COEFF(EXPRN:polynomial,VAR:kernel)
```

COEFF is an operator that partitions EXPRN into its various coefficients with respect to VAR and returns them as a list, with the coefficient independent of VAR first.

Under normal circumstances, an error results if EXPRN is not a polynomial in VAR, although the coefficients themselves can be rational as long as they do not depend on VAR. However, if the switch RATARG is on, denominators are not checked for dependence on VAR, and are taken to be part of the coefficients.

Example:

```
coeff((y^2+z)^3/z,y);
```

returns the result

```
   2
{Z  ,0,3*Z,0,3,0,1/Z}.
```

whereas

```
coeff((y^2+z)^3/y,y);
```

gives an error if RATARG is off, and the result

```
  3        2
{Z /Y,0,3*Z /Y,0,3*Z/Y,0,1/Y}
```

if RATARG is on.

The length of the result of COEFF is the highest power of VAR encountered plus 1. In the above examples it is 7. In addition, the variable HIGH_POW is set to the highest non-zero power found in EXPRN during the evaluation, and LOW_POW to the lowest non-zero power, or zero if there is a constant term. If EXPRN is a constant, then HIGH_POW and LOW_POW are both set to zero.

8.5.2 COEFFN Operator

The COEFFN operator is designed to give the user a particular coefficient of a variable in a polynomial, as opposed to COEFF that returns all coefficients. COEFFN is used with the syntax

```
COEFFN(EXPRN:polynomial,VAR:kernel,N:integer)
```

It returns the n^{th} coefficient of `VAR` in the polynomial `EXPRN`.

8.5.3 PART Operator

Syntax:

```
PART(EXPRN:algebraic[,INTEXP:integer])
```

This operator works on the form of the expression as printed *or as it would have been printed at that point in the calculation* bearing in mind all the relevant switch settings at that point. The reader therefore needs some familiarity with the way that expressions are represented in prefix form in REDUCE to use these operators effectively. Furthermore, it is assumed that `PRI` is `ON` at that point in the calculation. The reason for this is that with `PRI` off, an expression is printed by walking the tree representing the expression internally. To save space, it is never actually transformed into the equivalent prefix expression as occurs when `PRI` is on. However, the operations on polynomials described elsewhere can be equally well used in this case to obtain the relevant parts.

The evaluation proceeds recursively down the integer expression list. In other words,

```
PART(<expression>,<integer1>,<integer2>)
    ->  PART(PART(<expression>,<integer1>),<integer2>)
```

and so on, and

```
PART(<expression>) ->  <expression>.
```

`INTEXP` can be any expression that evaluates to an integer. If the integer is positive, then that term of the expression is found. If the integer is 0, the operator is returned. Finally, if the integer is negative, the counting is from the tail of the expression rather than the head.

For example, if the expression `a+b` is printed as `A+B` (i.e., the ordering of the variables is alphabetical), then

```
part(a+b,2)  ->   B
part(a+b,-1) ->   B
```

and

```
part(a+b,0)  ->  PLUS
```

An operator `ARGLENGTH` is available to determine the number of arguments of the top level operator in an expression. If the expression does not contain a top level operator, then -1 is returned. For example,

```
arglength(a+b+c) ->  3
arglength(f())   ->  0
arglength(a)     ->  -1
```

8.5.4 Substituting for Parts of Expressions

`PART` may also be used to substitute for a given part of an expression. In this case, the `PART` construct appears on the left-hand side of an assignment statement, and the expression to replace the given part on the right-hand side.

For example, with the normal settings of the REDUCE switches:

```
xx := a+b;
part(xx,2) := c;    ->  A+C
part(c+d,0) := -;    -> C-D
```

Note that `xx` in the above is not changed by this substitution. In addition, unlike expressions such as array and matrix elements that have an *instant evaluation* property, the values of `part(xx,2)` and `part(c+d,0)` are also not changed.

Chapter 9

Polynomials and Rationals

Many operations in computer algebra are concerned with polynomials and rational functions. In this section, we review some of the switches and operators available for this purpose. These are in addition to those that work on general expressions (such as `DF` and `INT`) described elsewhere. In the case of operators, the arguments are first simplified before the operations are applied. In addition, they operate only on arguments of prescribed types, and produce a type mismatch error if given arguments which cannot be interpreted in the required mode with the current switch settings. For example, if an argument is required to be a kernel and `a/2` is used (with no other rules for `A`), an error

```
A/2 invalid as kernel
```

will result.

With the exception of those that select various parts of a polynomial or rational function, these operations have potentially significant effects on the space and time associated with a given calculation. The user should therefore experiment with their use in a given calculation in order to determine the optimum set for a given problem.

One such operation provided by the system is an operator `LENGTH` which returns the number of top level terms in the numerator of its argument. For example,

```
length ((a+b+c)^3/(c+d));
```

has the value 10. To get the number of terms in the denominator, one would first select the denominator by the operator DEN and then call LENGTH, as in

```
length den ((a+b+c)^3/(c+d));
```

Other operations currently supported, the relevant switches and operators, and the required argument and value modes of the latter, follow.

9.1 Controlling the Expansion of Expressions

The switch EXP controls the expansion of expressions. If it is off, no expansion of powers or products of expressions occurs. Users should note however that in this case results come out in a normal but not necessarily canonical form. This means that zero expressions simplify to zero, but that two equivalent expressions need not necessarily simplify to the same form.

Example: With EXP on, the two expressions

```
(a+b)*(a+2*b)
```

and

```
a^2+3*a*b+2*b^2
```

will both simplify to the latter form. With EXP off, they would remain unchanged, unless the complete factoring (ALLFAC) option were in force. EXP is normally on.

Several operators that expect a polynomial as an argument behave differently when EXP is off, since there is often only one term at the top level. For example, with EXP off

```
length((a+b+c)^3/(c+d));
```

returns the value 1.

9.2 Factorization of Polynomials

REDUCE is capable of factorizing univariate and multivariate polynomials that have integer coefficients, finding all factors that also have integer coefficients. The package for doing this was written by Dr. Arthur C. Norman and Ms. P. Mary Ann Moore at The University of Cambridge. It is described in P. M. A. Moore and A. C. Norman, "Implementing a Polynomial Factorization and GCD Package", Proc. SYMSAC '81, ACM (New York) (1981), 109-116.

The easiest way to use this facility is to turn on the switch `FACTOR`, which causes all expressions to be output in a factored form. For example, with `FACTOR` on, the expression `A^2-B^2` is returned as `(A+B)*(A-B)`.

It is also possible to factorize a given expression explicitly. The operator `FACTORIZE` that invokes this facility is used with the syntax

```
FACTORIZE(EXPRN:polynomial[,INTEXP:prime integer]):list,
```

the optional argument of which will be described later. Thus to find and display all factors of the cyclotomic polynomial $x^{105} - 1$, one could write:

```
factorize(x^105-1);
```

The result is a list of factor,exponent pairs. In the above example, there is no overall numerical factor in the result, so the results will consist only of polynomials in x. The number of such polynomials can be found by using the operator `LENGTH`. If there is a numerical factor, as in factorizing $12x^2 - 12$, that factor will appear as the first member of the result. It will however not be factored further. Prime factors of such numbers can be found, using a probabilistic algorithm, by turning on the switch `IFACTOR`. For example,

```
on ifactor; factorize(12x^2-12);
```

would result in the output

```
{{2,2},{3,1},{X + 1,1},{X - 1,1}}.
```

If the first argument of `FACTORIZE` is an integer, it will be decomposed into its prime components, whether or not `IFACTOR` is on.

Note that the `IFACTOR` switch only affects the result of `FACTORIZE`. It has no effect if the `FACTOR` switch is also on.

The order in which the factors occur in the result (with the exception of a possible overall numerical coefficient which comes first) can be system dependent and should not be relied on. Similarly it should be noted that any pair of individual factors can be negated without altering their product, and that REDUCE may sometimes do that.

The factorizer works by first reducing multivariate problems to univariate ones and then solving the univariate ones modulo small primes. It normally selects both evaluation points and primes using a random number generator that should lead to different detailed behavior each time any particular problem is tackled. If, for some reason, it is known that a certain (probably univariate) factorization can be performed effectively with a known prime, `P` say, this value of `P` can be handed to `FACTORIZE` as a second argument. An error will occur if a non-prime is provided to `FACTORIZE` in this manner. It is also an error to specify a prime that divides the discriminant of the polynomial being factored, but users should note that this condition is not checked by the program, so this capability should be used with care.

Factorization can be performed over a number of polynomial coefficient domains in addition to integers. The particular description of the relevant domain should be consulted to see if factorization is supported. For example, the following statements will factorize $x^4 + 1$ modulo 7:

```
setmod 7;
on modular;
factorize(x^4+1);
```

The factorization module is provided with a trace facility that may be useful as a way of monitoring progress on large problems, and of satisfying curiosity about the internal workings of the package. The most simple use of this is enabled by issuing the REDUCE command `on trfac;` . Following this, all calls to the factorizer will generate informative messages reporting on such things as the reduction of multivariate to univariate cases, the choice of a prime and the reconstruction of full factors from their images. Further levels of detail in the trace are intended mainly for system tuners and for the investigation of suspected bugs. For example, `TRALLFAC` gives tracing information at all levels of detail. The switch that can be set by `on timings;` makes it possible for one who is familiar with the algorithms used to determine what part of the factorization code is consuming the most resources. `on overview`; reduces the amount of detail presented in other forms of trace. Other forms of trace output are enabled by directives of the form

```
symbolic set!-trace!-factor(<number>,<filename>);
```

where useful numbers are 1, 2, 3 and 100, 101, ... This facility is intended to make it possible to discover in fairly great detail what just some small part of the code has been doing — the numbers refer mainly to depths of recursion when the factorizer calls itself, and to the split between its work forming and factorizing images and reconstructing full factors from these. If `NIL` is used in place of a filename the trace output requested is directed to the standard output stream. After use of this trace facility the generated trace files should be closed by calling

```
symbolic close!-trace!-files();
```

NOTE: Using the factorizer with `MCD` off will result in an error.

9.3 Cancellation of Common Factors

Facilities are available in REDUCE for cancelling common factors in the numerators and denominators of expressions, at the option of the user. The system will perform this greatest common divisor computation if the switch `GCD` is on. (`GCD` is normally off.)

A check is automatically made, however, for common variable and numerical products in the numerators and denominators of expressions, and the appropriate cancellations made.

When `GCD` is on, and `EXP` is off, a check is made for square free factors in an expression. This includes separating out and independently checking the content of a given polynomial where appropriate. (For an explanation of these terms, see Anthony C. Hearn, "Non-Modular Computation of Polynomial GCDs Using Trial Division", Proc. EUROSAM 79, published as Lecture Notes on Comp. Science, Springer-Verlag, Berlin, No 72 (1979) 227-239.)

Example: With `EXP` off and `GCD` on, the polynomial `a*c+a*d+b*c+b*d` would be returned as `(A+B)*(C+D)`.

Under normal circumstances, GCDs are computed using an algorithm described in the above paper. It is also possible in REDUCE to compute GCDs using an alternative algorithm, called the EZGCD Algorithm, which uses modular arithmetic. The switch `EZGCD`, if on in addition to `GCD`, makes this happen.

In non-trivial cases, the EZGCD algorithm is almost always better than the basic algorithm, often by orders of magnitude. We therefore *strongly* advise users to use the `EZGCD` switch where they have the resources available for supporting the package.

For a description of the EZGCD algorithm, see J. Moses and D.Y.Y. Yun, "The EZ GCD Algorithm", Proc. ACM 1973, ACM, New York (1973) 159-

166.

NOTE: This package shares code with the factorizer, so a certain amount of trace information can be produced using the factorizer trace switches.

9.3.1 Determining the GCD of Two Polynomials

This operator, used with the syntax

```
GCD(EXPRN1:polynomial,EXPRN2:polynomial):polynomial,
```

returns the greatest common divisor of the two polynomials `EXPRN1` and `EXPRN2`.

Examples:

```
gcd(x^2+2*x+1,x^2+3*x+2) ->  X+1
gcd(2*x^2-2*y^2,4*x+4*y) ->  2*X+2*Y
gcd(x^2+y^2,x-y)         ->  1.
```

9.4 Working with Least Common Multiples

Greatest common divisor calculations can often become expensive if extensive work with large rational expressions is required. However, in many cases, the only significant cancellations arise from the fact that there are often common factors in the various denominators which are combined when two rationals are added. Since these denominators tend to be smaller and more regular in structure than the numerators, considerable savings in both time and space can occur if a full GCD check is made when the denominators are combined and only a partial check when numerators are constructed. In other words, the true least common multiple of the denominators is computed at each step. The switch `LCM` is available for this purpose, and is normally on.

In addition, the operator `LCM`, used with the syntax

```
LCM(EXPRN1:polynomial,EXPRN2:polynomial):polynomial,
```

returns the least common multiple of the two polynomials `EXPRN1` and `EXPRN2`.

Examples:

```
lcm(x^2+2*x+1,x^2+3*x+2) ->  X**3 + 4*X**2 + 5*X + 2
lcm(2*x^2-2*y^2,4*x+4*y) ->  4*(X**2 - Y**2)
```

9.5 Controlling Use of Common Denominators

When two rational functions are added, REDUCE normally produces an expression over a common denominator. However, if the user does not want denominators combined, he or she can turn off the switch `MCD` which controls this process. The latter switch is particularly useful if no greatest common divisor calculations are desired, or excessive differentiation of rational functions is required.

CAUTION: With `MCD` off, results are not guaranteed to come out in either normal or canonical form. In other words, an expression equivalent to zero may in fact not be simplified to zero. This option is therefore most useful for avoiding expression swell during intermediate parts of a calculation.

`MCD` is normally on.

9.6 REMAINDER Operator

This operator is used with the syntax

```
REMAINDER(EXPRN1:polynomial,EXPRN2:polynomial):polynomial.
```

It returns the remainder when `EXPRN1` is divided by `EXPRN2`. This is the true remainder based on the internal ordering of the variables, and not the pseudo-remainder. The pseudo-remainder and in general pseudo-division of polynomials can be calculated after loading the `polydiv` package. Please refer to the documentation of this package for details.

Examples:

```
remainder((x+y)*(x+2*y),x+3*y) ->  2*Y**2
remainder(2*x+y,2)             ->  Y.
```

CAUTION: In the default case, remainders are calculated over the integers. If you need the remainder with respect to another domain, it must be declared explicitly.

Example:

```
remainder(x^2-2,x+sqrt(2)); -> X^2 - 2
load_package arnum;
defpoly sqrt2**2-2;
remainder(x^2-2,x+sqrt2); -> 0
```

9.7 RESULTANT Operator

This is used with the syntax

```
RESULTANT(EXPRN1:polynomial,EXPRN2:polynomial,VAR:kernel):
   polynomial.
```

It computes the resultant of the two given polynomials with respect to the given variable, the coefficients of the polynomials can be taken from any domain. The result can be identified as the determinant of a Sylvester matrix, but can often also be thought of informally as the result obtained when the given variable is eliminated between the two input polynomials. If the two input polynomials have a non-trivial GCD their resultant vanishes.

The switch `Bezout` controls the computation of the resultants. It is off by default. In this case a subresultant algorithm is used. If the switch Bezout is turned on, the resultant is computed via the Bezout Matrix. However, in the latter case, only polynomial coefficients are permitted.

The sign conventions used by the resultant function follow those in R. Loos, "Computing in Algebraic Extensions" in "Computer Algebra — Symbolic and Algebraic Computation", Second Ed., Edited by B. Buchberger, G.E. Collins and R. Loos, Springer-Verlag, 1983. Namely, with A and B not dependent on X:

```
                              deg(p)*deg(q)
resultant(p(x),q(x),x)= (-1)              *resultant(q,p,x)

                           deg(p)
resultant(a,p(x),x)   = a

resultant(a,b,x)      = 1
```

Examples:

```
                                   2
resultant(x/r*u+y,u*y,u)   ->  - y
```

calculation in an algebraic extension:

```
load arnum;
defpoly sqrt2**2 - 2;

resultant(x + sqrt2,sqrt2 * x +1,x) -> -1
```

or in a modular domain:

```
setmod 17;
on modular;

resultant(2x+1,3x+4,x)    -> 5
```

9.8 DECOMPOSE Operator

The DECOMPOSE operator takes a multivariate polynomial as argument, and returns an expression and a list of equations from which the original polynomial can be found by composition. Its syntax is:

```
DECOMPOSE(EXPRN:polynomial):list.
```

For example:

```
decompose(x^8-88*x^7+2924*x^6-43912*x^5+263431*x^4-
               218900*x^3+65690*x^2-7700*x+234)
             2                  2            2
        -> {U  + 35*U + 234, U=V  + 10*V, V=X  - 22*X}
                               2
decompose(u^2+v^2+2u*v+1)  -> {W  + 1, W=U + V}
```

Users should note however that, unlike factorization, this decomposition is not unique.

9.9 INTERPOL operator

Syntax:

```
INTERPOL(<values>,<variable>,<points>);
```

where `<values>` and `<points>` are lists of equal length and `<variable>` is an algebraic expression (preferably a kernel).

`INTERPOL` generates an interpolation polynomial f in the given variable of degree length(`<values>`)-1. The unique polynomial f is defined by the property that for corresponding elements v of `<values>` and p of `<points>` the relation $f(p) = v$ holds.

The Aitken-Neville interpolation algorithm is used which guarantees a stable result even with rounded numbers and an ill-conditioned problem.

9.10 Obtaining Parts of Polynomials and Rationals

These operators select various parts of a polynomial or rational function structure. Except for the cost of rearrangement of the structure, these operations take very little time to perform.

For those operators in this section that take a kernel `VAR` as their second argument, an error results if the first expression is not a polynomial in `VAR`, although the coefficients themselves can be rational as long as they do not depend on `VAR`. However, if the switch `RATARG` is on, denominators are not checked for dependence on VAR, and are taken to be part of the coefficients.

9.10.1 DEG Operator

This operator is used with the syntax

```
DEG(EXPRN:polynomial,VAR:kernel):integer.
```

It returns the leading degree of the polynomial EXPRN in the variable VAR. If VAR does not occur as a variable in EXPRN, 0 is returned.

Examples:

```
deg((a+b)*(c+2*d)^2,a) ->  1
deg((a+b)*(c+2*d)^2,d) ->  2
deg((a+b)*(c+2*d)^2,e) ->  0.
```

Note also that if RATARG is on,

```
deg((a+b)^3/a,a)       ->  3
```

since in this case, the denominator A is considered part of the coefficients of the numerator in A. With RATARG off, however, an error would result in this case.

9.10.2 DEN Operator

This is used with the syntax:

```
DEN(EXPRN:rational):polynomial.
```

It returns the denominator of the rational expression EXPRN. If EXPRN is a polynomial, 1 is returned.

Examples:

```
den(x/y^2)   ->  Y**2
den(100/6)   ->  3
        [since 100/6 is first simplified to 50/3]
den(a/4+b/6) ->  12
den(a+b)     ->  1
```

9.10.3 LCOF Operator

LCOF is used with the syntax

```
LCOF(EXPRN:polynomial,VAR:kernel):polynomial.
```

It returns the leading coefficient of the polynomial `EXPRN` in the variable `VAR`. If `VAR` does not occur as a variable in `EXPRN`, `EXPRN` is returned.

Examples:

```
lcof((a+b)*(c+2*d)^2,a) ->  C**2+4*C*D+4*D**2
lcof((a+b)*(c+2*d)^2,d) ->  4*(A+B)
lcof((a+b)*(c+2*d),e)   ->  A*C+2*A*D+B*C+2*B*D
```

9.10.4 LPOWER Operator

Syntax:

```
LPOWER(EXPRN:polynomial,VAR:kernel):polynomial.
```

LPOWER returns the leading power of `EXPRN` with respect to `VAR`. If `EXPRN` does not depend on `VAR`, 1 is returned.

Examples:

```
lpower((a+b)*(c+2*d)^2,a) ->  A
lpower((a+b)*(c+2*d)^2,d) ->  D**2
lpower((a+b)*(c+2*d),e)   ->  1
```

9.10.5 LTERM Operator

Syntax:

```
LTERM(EXPRN:polynomial,VAR:kernel):polynomial.
```

LTERM returns the leading term of `EXPRN` with respect to `VAR`. If `EXPRN` does not depend on `VAR`, `EXPRN` is returned.

Examples:

```
lterm((a+b)*(c+2*d)^2,a) ->  A*(C**2+4*C*D+4*D**2)
lterm((a+b)*(c+2*d)^2,d) ->  4*D**2*(A+B)
lterm((a+b)*(c+2*d),e)   ->  A*C+2*A*D+B*C+2*B*D
```

Compatibility Note: In some earlier versions of REDUCE, `LTERM` returned `0` if the `EXPRN` did not depend on `VAR`. In the present version, `EXPRN` is always equal to `LTERM(EXPRN,VAR)` + `REDUCT(EXPRN,VAR)`.

9.10.6 MAINVAR Operator

Syntax:

```
MAINVAR(EXPRN:polynomial):expression.
```

Returns the main variable (based on the internal polynomial representation) of EXPRN. If EXPRN is a domain element, 0 is returned.

Examples:

Assuming A has higher kernel order than B, C, or D:

```
mainvar((a+b)*(c+2*d)^2) ->  A
mainvar(2)               ->  0
```

9.10.7 NUM Operator

Syntax:

```
NUM(EXPRN:rational):polynomial.
```

Returns the numerator of the rational expression EXPRN. If EXPRN is a polynomial, that polynomial is returned.

Examples:

```
num(x/y^2)  ->  X
num(100/6)   ->  50
num(a/4+b/6) ->  3*A+2*B
num(a+b)     ->  A+B
```

9.10.8 REDUCT Operator

Syntax:

```
REDUCT(EXPRN:polynomial,VAR:kernel):polynomial.
```

Returns the reductum of EXPRN with respect to VAR (i.e., the part of EXPRN left after the leading term is removed). If EXPRN does not depend on the variable VAR, 0 is returned.

Examples:

```
reduct((a+b)*(c+2*d),a) ->  B*(C + 2*D)
reduct((a+b)*(c+2*d),d) ->  C*(A + B)
reduct((a+b)*(c+2*d),e) ->  0
```

Compatibility Note: In some earlier versions of REDUCE, `REDUCT` returned `EXPRN` if it did not depend on `VAR`. In the present version, `EXPRN` is always equal to `LTERM(EXPRN,VAR)` + `REDUCT(EXPRN,VAR)`.

9.11 Polynomial Coefficient Arithmetic

REDUCE allows for a variety of numerical domains for the numerical coefficients of polynomials used in calculations. The default mode is integer arithmetic, although the possibility of using real coefficients has been discussed elsewhere. Rational coefficients have also been available by using integer coefficients in both the numerator and denominator of an expression, using the `ON DIV` option to print the coefficients as rationals. However, REDUCE includes several other coefficient options in its basic version which we shall describe in this section. All such coefficient modes are supported in a table-driven manner so that it is straightforward to extend the range of possibilities. A description of how to do this is given in R.J. Bradford, A.C. Hearn, J.A. Padget and E. Schrüfer, "Enlarging the REDUCE Domain of Computation," Proc. of SYMSAC '86, ACM, New York (1986), 100–106.

9.11.1 Rational Coefficients in Polynomials

Instead of treating rational numbers as the numerator and denominator of a rational expression, it is also possible to use them as polynomial coefficients directly. This is accomplished by turning on the switch `RATIONAL`.

Example: With `RATIONAL` off, the input expression `a/2` would be converted into a rational expression, whose numerator was `A` and denominator 2. With `RATIONAL` on, the same input would become a rational expression with numerator `1/2*A` and denominator `1`. Thus the latter can be used in operations that require polynomial input whereas the former could not.

9.11.2 Real Coefficients in Polynomials

The switch `ROUNDED` permits the use of arbitrary sized real coefficients in polynomial expressions. The actual precision of these coefficients can be set by the operator `PRECISION`. For example, `precision 50;` sets the precision to fifty decimal digits. The default precision is system dependent and can be found by `precision 0;`. In this mode, denominators are automatically made monic, and an appropriate adjustment is made to the numerator.

Example: With `ROUNDED` on, the input expression `a/2` would be converted into a rational expression whose numerator is `0.5*A` and denominator `1`.

Internally, REDUCE uses floating point numbers up to the precision supported by the underlying machine hardware, and so-called *bigfloats* for higher precision or whenever necessary to represent numbers whose value cannot be represented in floating point. The internal precision is two decimal digits greater than the external precision to guard against roundoff inaccuracies. Bigfloats represent the fraction and exponent parts of a floating-point number by means of (arbitrary precision) integers, which is a more precise representation in many cases than the machine floating point arithmetic, but not as efficient. If a case arises where use of the machine arithmetic leads to problems, a user can force REDUCE to use the bigfloat representation at all precisions by turning on the switch `ROUNDBF`. In rare cases, this switch is turned on by the system, and the user informed by the message

```
ROUNDBF turned on to increase accuracy
```

Rounded numbers are normally printed to the specified precision. However, if the user wishes to print such numbers with less precision, the printing precision can be set by the command `PRINT_PRECISION`. For example, `print_precision 5;` will cause such numbers to be printed with five digits maximum.

Under normal circumstances when `ROUNDED` is on, REDUCE converts the number 1.0 to the integer 1. If this is not desired, the switch `NOCONVERT` can be turned on.

Numbers that are stored internally as bigfloats are normally printed with a space between every five digits to improve readability. If this feature is not required, it can be suppressed by turning off the switch `BFSPACE`.

Further information on the bigfloat arithmetic may be found in T. Sasaki, "Manual for Arbitrary Precision Real Arithmetic System in REDUCE", Department of Computer Science, University of Utah, Technical Note No.

TR-8 (1979).

When a real number is input, it is normally truncated to the precision in effect at the time the number is read. If it is desired to keep the full precision of all numbers input, the switch ADJPREC (for *adjust precision*) can be turned on. While on, ADJPREC will automatically increase the precision, when necessary, to match that of any integer or real input, and a message printed to inform the user of the precision increase.

When ROUNDED is on, rational numbers are normally converted to rounded representation. However, if a user wishes to keep such numbers in a rational form until used in an operation that returns a real number, the switch ROUNDALL can be turned off. This switch is normally on.

Results from rounded calculations are returned in rounded form with two exceptions: if the result is recognized as 0 or 1 to the current precision, the integer result is returned.

9.11.3 Modular Number Coefficients in Polynomials

REDUCE includes facilities for manipulating polynomials whose coefficients are computed modulo a given base. To use this option, two commands must be used; SETMOD <integer>, to set the prime modulus, and ON MODULAR to cause the actual modular calculations to occur. For example, with setmod 3; and on modular;, the polynomial (a+2*b)^3 would become A^3+2*B^3.

The argument of SETMOD is evaluated algebraically, except that non-modular (integer) arithmetic is used. Thus the sequence

```
setmod 3; on modular; setmod 7;
```

will correctly set the modulus to 7.

Modular numbers are by default represented by integers in the interval [0,p-1] where p is the current modulus. Sometimes it is more convenient to use an equivalent symmetric representation in the interval [-p/2+1,p/2], or more precisely [-floor((p-1)/2), ceiling((p-1)/2)], especially if the modular numbers map objects that include negative quantities. The switch BALANCED_MOD allows you to select the symmetric representation for output.

Users should note that the modular calculations are on the polynomial coefficients only. It is not currently possible to reduce the exponents since no check for a prime modulus is made (which would allow x^{p-1} to be reduced to 1 mod p). Note also that any division by a number not co-prime with the modulus will result in the error "Invalid modular division".

9.11.4 Complex Number Coefficients in Polynomials

Although REDUCE routinely treats the square of the variable i as equivalent to -1, this is not sufficient to reduce expressions involving i to lowest terms, or to factor such expressions over the complex numbers. For example, in the default case,

```
factorize(a^2+1);
```

gives the result

```
{{A**2+1,1}}
```

and

```
(a^2+b^2)/(a+i*b)
```

is not reduced further. However, if the switch `COMPLEX` is turned on, full complex arithmetic is then carried out. In other words, the above factorization will give the result

```
{{A + I,1},{A - I,1}}
```

and the quotient will be reduced to `A-I*B`.

The switch `COMPLEX` may be combined with `ROUNDED` to give complex real numbers; the appropriate arithmetic is performed in this case.

Complex conjugation is used to remove complex numbers from denominators of expressions. To do this if `COMPLEX` is off, you must turn the switch `RATIONALIZE` on.

Chapter 10

Substitution Commands

An important class of commands in REDUCE define substitutions for variables and expressions to be made during the evaluation of expressions. Such substitutions use the prefix operator SUB, various forms of the command LET, and rule sets.

10.1 SUB Operator

Syntax:

```
SUB(<substitution_list>,EXPRN1:algebraic):algebraic
```

where <substitution_list> is a list of one or more equations of the form

```
VAR:kernel=EXPRN:algebraic
```

or a kernel that evaluates to such a list.

The SUB operator gives the algebraic result of replacing every occurrence of the variable VAR in the expression EXPRN1 by the expression EXPRN. Specifically, EXPRN1 is first evaluated using all available rules. Next the substitutions are made, and finally the substituted expression is reevaluated. When more than one variable occurs in the substitution list, the substitution is performed by recursively walking down the tree representing EXPRN1, and replacing every VAR found by the appropriate EXPRN. The EXPRN are not themselves searched for any occurrences of the various VARs. The trivial case SUB(EXPRN1) returns the algebraic value of EXPRN1.

Examples:

```
                                      2               2
sub({x=a+y,y=y+1},x^2+y^2) -> A   + 2*A*Y + 2*Y   + 2*Y + 1
```

and with s := {x=a+y,y=y+1},

```
                                      2               2
sub(s,x^2+y^2)             -> A   + 2*A*Y + 2*Y   + 2*Y + 1
```

Note that the global assignments x:=a+y, etc., do not take place.

EXPRN1 can be any valid algebraic expression whose type is such that a substitution process is defined for it (e.g., scalar expressions, lists and matrices). An error will occur if an expression of an invalid type for substitution occurs either in EXPRN or EXPRN1.

The braces around the substitution list may also be omitted, as in:

```
                                      2               2
sub(x=a+y,y=y+1,x^2+y^2)   -> A   + 2*A*Y + 2*Y   + 2*Y + 1
```

10.2 LET Rules

Unlike substitutions introduced via SUB, LET rules are global in scope and stay in effect until replaced or CLEARed.

The simplest use of the LET statement is in the form

```
LET <substitution list>
```

where <substitution list> is a list of rules separated by commas, each of the form:

```
<variable> = <expression>
```

or

```
<prefix operator>(<argument>,...,<argument>) = <expression>
```

or

```
<argument> <infix operator>,..., <argument> = <expression>
```

For example,

```
let {x = y^2,
     h(u,v) = u - v,
     cos(pi/3) = 1/2,
     a*b = c,
     l+m = n,
     w^3 = 2*z - 3,
     z^10 = 0}
```

The list brackets can be left out if preferred. The above rules could also have been entered as seven separate `LET` statements.

After such `LET` rules have been input, `X` will always be evaluated as the square of `Y`, and so on. This is so even if at the time the `LET` rule was input, the variable `Y` had a value other than `Y`. (In contrast, the assignment `x:=y^2` will set `X` equal to the square of the current value of `Y`, which could be quite different.)

The rule `let a*b=c` means that whenever `A` and `B` are both factors in an expression their product will be replaced by `C`. For example, `a^5*b^7*w` would be replaced by `c^5*b^2*w`.

The rule for `l+m` will not only replace all occurrences of `l+m` by `N`, but will also normally replace `L` by `n-m`, but not `M` by `n-l`. A more complete description of this case is given in Section 10.2.5.

The rule pertaining to `w^3` will apply to any power of `W` greater than or equal to the third.

Note especially the last example, `let z^10=0`. This declaration means, in effect: ignore the tenth or any higher power of `Z`. Such declarations, when appropriate, often speed up a computation to a considerable degree. (See Section 10.4 for more details.)

Any new operators occurring in such `LET` rules will be automatically declared `OPERATOR` by the system, if the rules are being read from a file. If they are being entered interactively, the system will ask `DECLARE ... OPERATOR?` . Answer `Y` or `N` and hit $\boxed{\text{Return}}$.

In each of these examples, substitutions are only made for the explicit expressions given; i.e., none of the variables may be considered arbitrary in any sense. For example, the command

```
let h(u,v) = u - v;
```

will cause `h(u,v)` to evaluate to `U - V`, but will not affect `h(u,z)` or `H` with any arguments other than precisely the symbols `U,V`.

These simple `LET` rules are on the same logical level as assignments made with the := operator. An assignment `x := p+q` cancels a rule `let x = y^2` made earlier, and vice versa.

CAUTION: A recursive rule such as

```
let x = x + 1;
```

is erroneous, since any subsequent evaluation of `X` would lead to a non-terminating chain of substitutions:

```
x -> x + 1 -> (x + 1) + 1 -> ((x + 1) + 1) + 1 -> ...
```

Similarly, coupled substitutions such as

```
let l = m + n, n = l + r;
```

would lead to the same error. As a result, if you try to evaluate an `X`, `L` or `N` defined as above, you will get an error such as

```
X improperly defined in terms of itself
```

Array and matrix elements can appear on the left-hand side of a `LET` statement. However, because of their *instant evaluation* property, it is the value of the element that is substituted for, rather than the element itself. E.g.,

```
array a(5);
a(2) := b;
let a(2) = c;
```

results in `B` being substituted by `C`; the assignment for `a(2)` does not change.

Finally, if an error occurs in any equation in a `LET` statement (including generalized statements involving `FOR ALL` and `SUCH THAT)`, the remaining rules are not evaluated.

10.2.1 FOR ALL ... LET

If a substitution for all possible values of a given argument of an operator is required, the declaration FOR ALL may be used. The syntax of such a command is

```
FOR ALL <variable>,...,<variable>
        <LET statement> <terminator>
```

e.g.,

```
for all x,y let h(x,y) = x-y;
for all x let k(x,y) = x^y;
```

The first of these declarations would cause h(a,b) to be evaluated as A-B, h(u+v,u+w) to be V-W, etc. If the operator symbol H is used with more or fewer argument places, not two, the LET would have no effect, and no error would result.

The second declaration would cause k(a,y) to be evaluated as a^y, but would have no effect on k(a,z) since the rule didn't say FOR ALL Y ... .

Where we used X and Y in the examples, any variables could have been used. This use of a variable doesn't affect the value it may have outside the LET statement. However, you should remember what variables you actually used. If you want to delete the rule subsequently, you must use the same variables in the CLEAR command.

It is possible to use more complicated expressions as a template for a LET statement, as explained in the section on substitutions for general expressions. In nearly all cases, the rule will be accepted, and a consistent application made by the system. However, if there is a sole constant or a sole free variable on the left-hand side of a rule (e.g., let 2=3 or for all x let x=2), then the system is unable to handle the rule, and the error message

```
Substitution for ... not allowed
```

will be issued. Any variable listed in the FOR ALL part will have its symbol preceded by an equal sign: X in the above example will appear as =X. An error will also occur if a variable in the FOR ALL part is not properly matched on both sides of the LET equation.

10.2.2 FOR ALL . . . SUCH THAT . . . LET

If a substitution is desired for more than a single value of a variable in an operator or other expression, but not all values, a conditional form of the `FOR ALL ...LET` declaration can be used.

Example:

```
for all x such that numberp x and x<0 let h(x)=0;
```

will cause `h(-5)` to be evaluated as 0, but `H` of a positive integer, or of an argument that is not an integer at all, would not be affected. Any boolean expression can follow the `SUCH THAT` keywords.

10.2.3 Removing Assignments and Substitution Rules

The user may remove all assignments and substitution rules from any expression by the command `CLEAR`, in the form

```
CLEAR <expression>,...,<expression><terminator>
```

e.g.

```
clear x, h(x,y);
```

Because of their *instant evaluation* property, array and matrix elements cannot be cleared with `CLEAR`. For example, if `A` is an array, you must say

```
a(3) := 0;
```

rather than

```
clear a(3);
```

to "clear" element `a(3)`.

On the other hand, a whole array (or matrix) `A` can be cleared by the command `clear a`; This means much more than resetting to 0 all the elements of `A`. The fact that `A` is an array, and what its dimensions are, are forgotten, so `A` can be redefined as another type of object, for example an operator.

The more general types of `LET` declarations can also be deleted by using `CLEAR`. Simply repeat the `LET` rule to be deleted, using `CLEAR` in place of

LET, and omitting the equal sign and right-hand part. The same dummy variables must be used in the FOR ALL part, and the boolean expression in the SUCH THAT part must be written the same way. (The placing of blanks doesn't have to be identical.)

Example: The LET rule

```
for all x such that numberp x and x<0 let h(x)=0;
```

can be erased by the command

```
for all x such that numberp x and x<0 clear h(x);
```

10.2.4 Overlapping LET Rules

CLEAR is not the only way to delete a LET rule. A new LET rule identical to the first, but with a different expression after the equal sign, replaces the first. Replacements are also made in other cases where the existing rule would be in conflict with the new rule. For example, a rule for x^4 would replace a rule for x^5. The user should however be cautioned against having several LET rules in effect that relate to the same expression. No guarantee can be given as to which rules will be applied by REDUCE or in what order. It is best to CLEAR an old rule before entering a new related LET rule.

10.2.5 Substitutions for General Expressions

The examples of substitutions discussed in other sections have involved very simple rules. However, the substitution mechanism used in REDUCE is very general, and can handle arbitrarily complicated rules without difficulty.

The general substitution mechanism used in REDUCE is discussed in Hearn, A. C., "REDUCE, A User-Oriented Interactive System for Algebraic Simplification," Interactive Systems for Experimental Applied Mathematics, (edited by M. Klerer and J. Reinfelds), Academic Press, New York (1968), 79-90, and Hearn. A. C., "The Problem of Substitution," Proc. 1968 Summer Institute on Symbolic Mathematical Computation, IBM Programming Laboratory Report FSC 69-0312 (1969). For the reasons given in these references, REDUCE does not attempt to implement a general pattern matching algorithm. However, the present system uses far more sophisticated techniques than those discussed in the above papers. It is now possible for the rules appearing in arguments of LET to have the form

```
<substitution expression> = <expression>
```

where any rule to which a sensible meaning can be assigned is permitted. However, this meaning can vary according to the form of `<substitution expression>`. The semantic rules associated with the application of the substitution are completely consistent, but somewhat complicated by the pragmatic need to perform such substitutions as efficiently as possible. The following rules explain how the majority of the cases are handled.

To begin with, the `<substitution expression>` is first partly simplified by collecting like terms and putting identifiers (and kernels) in the system order. However, no substitutions are performed on any part of the expression with the exception of expressions with the *instant evaluation* property, such as array and matrix elements, whose actual values are used. It should also be noted that the system order used is not changeable by the user, even with the `KORDER` command. Specific cases are then handled as follows:

1. If the resulting simplified rule has a left-hand side that is an identifier, an expression with a top-level algebraic operator or a power, then the rule is added without further change to the appropriate table.

2. If the operator * appears at the top level of the simplified left-hand side, then any constant arguments in that expression are moved to the right-hand side of the rule. The remaining left-hand side is then added to the appropriate table. For example,

   ```
   let 2*x*y=3
   ```

   becomes

   ```
   let x*y=3/2
   ```

   so that `x*y` is added to the product substitution table, and when this rule is applied, the expression `x*y` becomes 3/2, but `X` or `Y` by themselves are not replaced.

3. If the operators `+`, `-` or `/` appear at the top level of the simplified left-hand side, all but the first term is moved to the right-hand side of the rule. Thus the rules

   ```
   let l+m=n, x/2=y, a-b=c
   ```

   become

```
let l=n-m, x=2*y, a=c+b.
```

One problem that can occur in this case is that if a quantified expression is moved to the right-hand side, a given free variable might no longer appear on the left-hand side, resulting in an error because of the unmatched free variable. E.g.,

```
for all x,y let f(x)+f(y)=x*y
```

would become

```
for all x,y let f(x)=x*y-f(y)
```

which no longer has `Y` on both sides.

The fact that array and matrix elements are evaluated in the left-hand side of rules can lead to confusion at times. Consider for example the statements

```
array a(5); let x+a(2)=3; let a(3)=4;
```

The left-hand side of the first rule will become `X`, and the second 0. Thus the first rule will be instantiated as a substitution for `X`, and the second will result in an error.

The order in which a list of rules is applied is not easily understandable without a detailed knowledge of the system simplification protocol. It is also possible for this order to change from release to release, as improved substitution techniques are implemented. Users should therefore assume that the order of application of rules is arbitrary, and program accordingly.

After a substitution has been made, the expression being evaluated is reexamined in case a new allowed substitution has been generated. This process is continued until no more substitutions can be made.

As mentioned elsewhere, when a substitution expression appears in a product, the substitution is made if that expression divides the product. For example, the rule

```
let a^2*c = 3*z;
```

would cause `a^2*c*x` to be replaced by `3*Z*X` and `a^2*c^2` by `3*Z*C`. If the substitution is desired only when the substitution expression appears in a product with the explicit powers supplied in the rule, the command `MATCH` should be used instead.

For example,

```
match a^2*c = 3*z;
```

would cause `a^2*c*x` to be replaced by `3*Z*X`, but `a^2*c^2` would not be replaced. `MATCH` can also be used with the `FOR ALL` constructions described above.

To remove substitution rules of the type discussed in this section, the `CLEAR` command can be used, combined, if necessary, with the same `FOR ALL` clause with which the rule was defined, for example:

```
for all x clear log(e^x),e^log(x),cos(w*t+theta(x));
```

Note, however, that the arbitrary variable names in this case *must* be the same as those used in defining the substitution.

10.3 Rule Lists

Rule lists offer an alternative approach to defining substitutions that is different from either `SUB` or `LET`. In fact, they provide the best features of both, since they have all the capabilities of `LET`, but the rules can also be applied locally as is possible with `SUB`. In time, they will be used more and more in REDUCE. However, since they are relatively new, much of the REDUCE code you see uses the older constructs.

A rule list is a list of *rules* that have the syntax

```
<expression> => <expression> (WHEN <boolean expression>)
```

For example,

```
{cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
 cos(~n*pi)      => (-1)^n when remainder(n,2)=0}
```

The tilde preceding a variable marks that variable as *free* for that rule, much as a variable in a `FOR ALL` clause in a `LET` statement. The first occurrence of that variable in each relevant rule must be so marked on input, otherwise inconsistent results can occur. For example, the rule list

```
{cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
 cos(x)^2        => (1+cos(2x))/2}
```

designed to replace products of cosines, would not be correct, since the second rule would only apply to the explicit argument `X`. Later occurrences in the same rule may also be marked, but this is optional (internally, all such rules are stored with each relevant variable explicitly marked). The optional `WHEN` clause allows constraints to be placed on the application of the rule, much as the `SUCH THAT` clause in a `LET` statement.

A rule list may be named, for example

```
trig1 := {cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
          cos(~x)*sin(~y) => (sin(x+y)-sin(x-y))/2,
          sin(~x)*sin(~y) => (cos(x-y)-cos(x+y))/2,
          cos(~x)^2       => (1+cos(2*x))/2,
          sin(~x)^2       => (1-cos(2*x))/2};
```

Such named rule lists may be inspected as needed. E.g., the command `trig1;` would cause the above list to be printed.

Rule lists may be used in two ways. They can be globally instantiated by means of the command `LET`. For example,

```
let trig1;
```

would cause the above list of rules to be globally active from then on until cancelled by the command `CLEARRULES`, as in

```
clearrules trig1;
```

`CLEARRULES` has the syntax

```
CLEARRULES <rule list>|<name of rule list>(,...) .
```

The second way to use rule lists is to invoke them locally by means of a `WHERE` clause. For example

```
cos(a)*cos(b+c)
   where {cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2};
```

or

```
cos(a)*sin(b) where trigrules;
```

The syntax of an expression with a `WHERE` clause is:

```
<expression>
    WHERE <rule>|<rule list>(,<rule>|<rule list> ...)
```

so the first example above could also be written

```
cos(a)*cos(b+c)
   where cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2;
```

The effect of this construct is that the rule list(s) in the WHERE clause only apply to the expression on the left of WHERE. They have no effect outside the expression. In particular, they do not affect previously defined WHERE clauses or LET statements. For example, the sequence

```
let a=2;
a where a=>4;
a;
```

would result in the output

```
4

2
```

Although WHERE has a precedence less than any other infix operator, it still binds higher than keywords such as ELSE, THEN, DO, REPEAT and so on. Thus the expression

```
if a=2 then 3 else a+2 where a=3
```

will parse as

```
if a=2 then 3 else (a+2 where a=3)
```

WHERE may be used to introduce auxiliary variables in symbolic mode expressions, as described in Section 16.4. However, the symbolic mode use has different semantics, so expressions do not carry from one mode to the other.

Compatibility Note: In order to provide compatibility with older versions of rule lists released through the Network Library, it is currently possible to use an equal sign interchangeably with the replacement sign => in rules and LET statements. However, since this will change in future versions, the replacement sign is preferable in rules and the equal sign in non-rule-based LET statements.

Advanced Use of Rule Lists

Some advanced features of the rule list mechanism make it possible to write more complicated rules than those discussed so far, and in many cases to write more compact rule lists. These features are:

- Free operators
- Double slash operator
- Double tilde variables.

A **free operator** in the left hand side of a pattern will match any operator with the same number of arguments. The free operator is written in the same style as a variable. For example, the implementation of the product rule of differentiation can be written as:

```
operator diff, !~f, !~g;

prule := {diff(~f(~x) * ~g(~x),x) =>
             diff(f(x),x) * g(x) + diff(g(x),x) * f(x)};

let prule;

diff(sin(z)*cos(z),z);

         cos(z)*diff(sin(z),z) + diff(cos(z),z)*sin(z)
```

The **double slash operator** may be used as an alternative to a single slash (quotient) in order to match quotients properly. E.g., in the example of the Gamma function above, one can use:

```
gammarule :=
   {gamma(~z)//(~c*gamma(~zz))  => gamma(z)/(c*gamma(zz-1)*zz)
                  when fixp(zz -z) and (zz -z) >0,
    gamma(~z)//gamma(~zz) => gamma(z)/(gamma(zz-1)*zz)
                  when fixp(zz -z) and (zz -z) >0};

let gammarule;

gamma(z)/gamma(z+3);

          1
----------------------
```

```
 3      2
z  + 6*z  + 11*z + 6
```

The above example suffers from the fact that two rules had to be written in order to perform the required operation. This can be simplified by the use of **double tilde variables**. E.g. the rule list

```
GGrule :=  {
   gamma(~z)//(~~c*gamma(~zz))  => gamma(z)/(c*gamma(zz-1)*zz)
    when fixp(zz -z) and (zz -z) >0};
```

will implement the same operation in a much more compact way. In general, double tilde variables are bound to the neutral element with respect to the operation in which they are used.

Pattern given	Argument used	Binding
~z + ~~y	x	z=x; y=0
~z + ~~y	x+3	z=x; y=3 or z=3; y=x
~z * ~~y	x	z=x; y=1
~z * ~~y	x*3	z=x; y=3 or z=3; y=x
~z / ~~y	x	z=x; y=1
~z / ~~y	x/3	z=x; y=3

Remarks: A double tilde variable as the numerator of a pattern is not allowed. Also, using double tilde variables may lead to recursion errors when the zero case is not handled properly.

```
let f(~~a * ~x,x)  => a * f(x,x) when freeof (a,x);

f(z,z);

***** f(z,z) improperly defined in terms of itself

% BUT:

let ff(~~a * ~x,x)
       => a * ff(x,x) when freeof (a,x) and a neq 1;

ff(z,z);
                 ff(z,z)
```

```
ff(3*z,z);
                 3*ff(z,z)
```

Displaying Rules Associated with an Operator

The operator `SHOWRULES` takes a single identifier as argument, and returns in rule-list form the operator rules associated with that argument. For example:

```
showrules log;

{LOG(E) => 1,

 LOG(1) => 0,

      ~X
 LOG(E  ) => ~X,

                     1
 DF(LOG(~X),~X) => ----}
                     ~X
```

Such rules can then be manipulated further as with any list. For example `rhs first ws;` has the value `1`. Note that an operator may have other properties that cannot be displayed in such a form, such as the fact it is an odd function, or has a definition defined as a procedure.

Order of Application of Rules

If rules have overlapping domains, their order of application is important. In general, it is very difficult to specify this order precisely, so that it is best to assume that the order is arbitrary. However, if only one operator is involved, the order of application of the rules for this operator can be determined from the following:

1. Rules containing at least one free variable apply before all rules without free variables.

2. Rules activated in the most recent `LET` command are applied first.

3. LET with several entries generate the same order of application as a corresponding sequence of commands with one rule or rule set each.

4. Within a rule set, the rules containing at least one free variable are applied in their given order. In other words, the first member of the list is applied first.

5. Consistent with the first item, any rule in a rule list that contains no free variables is applied after all rules containing free variables.

Example: The following rule set enables the computation of exact values of the Gamma function:

```
operator gamma,gamma_error;
gamma_rules :=
{gamma(~x)=>sqrt(pi)/2 when x=1/2,
 gamma(~n)=>factorial(n-1) when fixp n and n>0,
 gamma(~n)=>gamma_error(n) when fixp n,
 gamma(~x)=>(x-1)*gamma(x-1) when fixp(2*x) and x>1,
 gamma(~x)=>gamma(x+1)/x when fixp(2*x)};
```

Here, rule by rule, cases of known or definitely uncomputable values are sorted out; e.g. the rule leading to the error expression will be applied for negative integers only, since the positive integers are caught by the preceding rule, and the last rule will apply for negative odd multiples of $1/2$ only. Alternatively the first rule could have been written as

```
gamma(1/2) => sqrt(pi)/2,
```

but then the case $x = 1/2$ should be excluded in the WHEN part of the last rule explicitly because a rule without free variables cannot take precedence over the other rules.

10.4 Asymptotic Commands

In expansions of polynomials involving variables that are known to be small, it is often desirable to throw away all powers of these variables beyond a certain point to avoid unnecessary computation. The command LET may be used to do this. For example, if only powers of X up to x^7 are needed, the command

```
let x^8 = 0;
```

will cause the system to delete all powers of `X` higher than 7.

CAUTION: This particular simplification works differently from most substitution mechanisms in REDUCE in that it is applied during polynomial manipulation rather than to the whole evaluated expression. Thus, with the above rule in effect, `x^10/x^5` would give the result zero, since the numerator would simplify to zero. Similarly `x^20/x^10` would give a `Zero divisor` error message, since both numerator and denominator would first simplify to zero.

The method just described is not adequate when expressions involve several variables having different degrees of smallness. In this case, it is necessary to supply an asymptotic weight to each variable and count up the total weight of each product in an expanded expression before deciding whether to keep the term or not. There are two associated commands in the system to permit this type of asymptotic constraint. The command `WEIGHT` takes a list of equations of the form

```
<kernel form> = <number>
```

where `<number>` must be a positive integer (not just evaluate to a positive integer). This command assigns the weight `<number>` to the relevant kernel form. A check is then made in all algebraic evaluations to see if the total weight of the term is greater than the weight level assigned to the calculation. If it is, the term is deleted. To compute the total weight of a product, the individual weights of each kernel form are multiplied by their corresponding powers and then added.

The weight level of the system is initially set to 1. The user may change this setting by the command

```
wtlevel <number>;
```

which sets `<number>` as the new weight level of the system. `<number>` must evaluate to a positive integer. WTLEVEL will also allow NIL as an argument, in which case the current weight level is returned.

Chapter 11

File Handling Commands

In many applications, it is desirable to load previously prepared REDUCE files into the system, or to write output on other files. REDUCE offers four commands for this purpose, namely, `IN`, `OUT`, `SHUT`, `LOAD`, and `LOAD_PACKAGE`. The first three operators are described here; `LOAD` and `LOAD_PACKAGE` are discussed in Section 18.2.

11.1 IN Command

This command takes a list of file names as argument and directs the system to input each file (that should contain REDUCE statements and commands) into the system. File names can either be an identifier or a string. The explicit format of these will be system dependent and, in many cases, site dependent. The explicit instructions for the implementation being used should therefore be consulted for further details. For example:

```
in f1,"ggg.rr.s";
```

will first load file `F1`, then `ggg.rr.s`. When a semicolon is used as the terminator of the IN statement, the statements in the file are echoed on the terminal or written on the current output file. If $ is used as the terminator, the input is not shown. Echoing of all or part of the input file can be prevented, even if a semicolon was used, by placing an `off echo;` command in the input file.

Files to be read using `IN` should end with `;END;`. Note the two semicolons!

First of all, this is protection against obscure difficulties the user will have if there are, by mistake, more BEGINs than ENDs on the file. Secondly, it triggers some file control book-keeping which may improve system efficiency. If END is omitted, an error message "End-of-file read" will occur.

11.2 OUT Command

This command takes a single file name as argument, and directs output to that file from then on, until another OUT changes the output file, or SHUT closes it. Output can go to only one file at a time, although many can be open. If the file has previously been used for output during the current job, and not SHUT, the new output is appended to the end of the file. Any existing file is erased before its first use for output in a job, or if it had been SHUT before the new OUT.

To output on the terminal without closing the output file, the reserved file name T (for terminal) may be used. For example, out ofile; will direct output to the file OFILE and out t; will direct output to the user's terminal.

The output sent to the file will be in the same form that it would have on the terminal. In particular x^2 would appear on two lines, an X on the lower line and a 2 on the line above. If the purpose of the output file is to save results to be read in later, this is not an appropriate form. We first must turn off the NAT switch that specifies that output should be in standard mathematical notation.

Example: To create a file ABCD from which it will be possible to read – using IN – the value of the expression XYZ:

```
off echo$      % needed if your input is from a file.
off nat$       % output in IN-readable form. Each expression
               % printed will end with a $ .
out abcd$      % output to new file
linelength 72$ % for systems with fixed input line length.
xyz:=xyz;      % will output "XYZ := " followed by the value
               % of XYZ
write ";end"$  % standard for ending files for IN
shut abcd$     % save ABCD, return to terminal output
on nat$                % restore usual output form
```

11.3 SHUT Command

This command takes a list of names of files that have been previously opened via an `OUT` statement and closes them. Most systems require this action by the user before he ends the REDUCE job (if not sooner), otherwise the output may be lost. If a file is shut and a further `OUT` command issued for the same file, the file is erased before the new output is written.

If it is the current output file that is shut, output will switch to the terminal. Attempts to shut files that have not been opened by `OUT`, or an input file, will lead to errors.

Chapter 12

Commands for Interactive Use

REDUCE is designed as an interactive system, but naturally it can also operate in a batch processing or background mode by taking its input command by command from the relevant input stream. There is a basic difference, however, between interactive and batch use of the system. In the former case, whenever the system discovers an ambiguity at some point in a calculation, such as a forgotten type assignment for instance, it asks the user for the correct interpretation. In batch operation, it is not practical to terminate the calculation at such points and require resubmission of the job, so the system makes the most obvious guess of the user's intentions and continues the calculation.

There is also a difference in the handling of errors. In the former case, the computation can continue since the user has the opportunity to correct the mistake. In batch mode, the error may lead to consequent erroneous (and possibly time consuming) computations. So in the default case, no further evaluation occurs, although the remainder of the input is checked for syntax errors. A message `"Continuing with parsing only"` informs the user that this is happening. On the other hand, the switch `ERRCONT`, if on, will cause the system to continue evaluating expressions after such errors occur.

When a syntactical error occurs, the place where the system detected the error is marked with three dollar signs ($$$). In interactive mode, the user can then use `ED` to correct the error, or retype the command. When a non-syntactical error occurs in interactive mode, the command being evaluated at the time the last error occurred is saved, and may later be reevaluated by the command `RETRY`.

12.1 Referencing Previous Results

It is often useful to be able to reference results of previous computations during a REDUCE session. For this purpose, REDUCE maintains a history of all interactive inputs and the results of all interactive computations during a given session. These results are referenced by the command number that REDUCE prints automatically in interactive mode. To use an input expression in a new computation, one writes `input(`n`)`, where n is the command number. To use an output expression, one writes `WS(`n`)`. `WS` references the previous command. E.g., if command number 1 was `INT(X-1,X)`; and the result of command number 7 was `X-1`, then

```
2*input(1)-ws(7)^2;
```

would give the result `-1`, whereas

```
2*ws(1)-ws(7)^2;
```

would yield the same result, but *without* a recomputation of the integral.

The operator `DISPLAY` is available to display previous inputs. If its argument is a positive integer, n say, then the previous n inputs are displayed. If its argument is `ALL` (or in fact any non-numerical expression), then all previous inputs are displayed.

12.2 Interactive Editing

It is possible when working interactively to edit any REDUCE input that comes from the user's terminal, and also some user-defined procedure definitions. At the top level, one can access any previous command string by the command `ed(`n`)`, where n is the desired command number as prompted by the system in interactive mode. `ED`; (i.e. no argument) accesses the previous command.

After `ED` has been called, you can now edit the displayed string using a string editor with the following commands:

`B`	move pointer to beginning
`C<character>`	replace next character by *character*
`D`	delete next character
`E`	end editing and reread text
`F<character>`	move pointer to next occurrence of *character*
`I<string><escape>`	insert *string* in front of pointer
`K<character>`	delete all characters until *character*
`P`	print string from current pointer
`Q`	give up with error exit
`S<string><escape>`	search for first occurrence of *string*, positioning pointer just before it
`space` or `X`	move pointer right one character.

The above table can be displayed online by typing a question mark followed by a carriage return to the editor. The editor prompts with an angle bracket. Commands can be combined on a single line, and all command sequences must be followed by a carriage return to become effective.

Thus, to change the command `x := a+1;` to `x := a+2`; and cause it to be executed, the following edit command sequence could be used:

```
f1c2e<return>.
```

The interactive editor may also be used to edit a user-defined procedure that has not been compiled. To do this, one says:

```
editdef <id>;
```

where `<id>` is the name of the procedure. The procedure definition will then be displayed in editing mode, and may then be edited and redefined on exiting from the editor.

Some versions of REDUCE now include input editing that uses the capabilities of modern window systems. Please consult your system dependent documentation to see if this is possible. Such editing techniques are usually much easier to use then `ED` or `EDITDEF`.

12.3 Interactive File Control

If input is coming from an external file, the system treats it as a batch processed calculation. If the user desires interactive response in this case, he can include the command `on int`; in the file. Likewise, he can issue the command `off int`; in the main program if he does not desire continual questioning from the system. Regardless of the setting of `INT`, input commands from a file are not kept in the system, and so cannot be edited using `ED`. However, many implementations of REDUCE provide a link to an external system editor that can be used for such editing. The specific instructions for the particular implementation should be consulted for information on this.

Two commands are available in REDUCE for interactive use of files. `PAUSE`; may be inserted at any point in an input file. When this command is encountered on input, the system prints the message `CONT?` on the user's terminal and halts. If the user responds `Y` (for yes), the calculation continues from that point in the file. If the user responds `N` (for no), control is returned to the terminal, and the user can input further statements and commands. Later on he can use the command `cont;` to transfer control back to the point in the file following the last `PAUSE` encountered. A top-level `pause;` from the user's terminal has no effect.

Chapter 13

Matrix Calculations

A very powerful feature of REDUCE is the ease with which matrix calculations can be performed. To extend our syntax to this class of calculations we need to add another prefix operator, `MAT`, and a further variable and expression type as follows:

13.1 MAT Operator

This prefix operator is used to represent $n \times m$ matrices. `MAT` has n arguments interpreted as rows of the matrix, each of which is a list of m expressions representing elements in that row. For example, the matrix

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

would be written as `mat((a,b,c),(d,e,f))`.

Note that the single column matrix

$$\begin{pmatrix} x \\ y \end{pmatrix}$$

becomes `mat((x),(y))`. The inside parentheses are required to distinguish it from the single row matrix

$$\begin{pmatrix} x & y \end{pmatrix}$$

that would be written as `mat((x,y))`.

13.2 Matrix Variables

An identifier may be declared a matrix variable by the declaration `MATRIX`. The size of the matrix may be declared explicitly in the matrix declaration, or by default in assigning such a variable to a matrix expression. For example,

```
matrix x(2,1),y(3,4),z;
```

declares `X` to be a 2 x 1 (column) matrix, `Y` to be a 3 x 4 matrix and `Z` a matrix whose size is to be declared later.

Matrix declarations can appear anywhere in a program. Once a symbol is declared to name a matrix, it can not also be used to name an array, operator or a procedure, or used as an ordinary variable. It can however be redeclared to be a matrix, and its size may be changed at that time. Note however that matrices once declared are *global* in scope, and so can then be referenced anywhere in the program. In other words, a declaration within a block (or a procedure) does not limit the scope of the matrix to that block, nor does the matrix go away on exiting the block (use `CLEAR` instead for this purpose). An element of a matrix is referred to in the expected manner; thus `x(1,1)` gives the first element of the matrix `X` defined above. References to elements of a matrix whose size has not yet been declared leads to an error. All elements of a matrix whose size is declared are initialized to 0. As a result, a matrix element has an *instant evaluation* property and cannot stand for itself. If this is required, then an operator should be used to name the matrix elements as in:

```
matrix m; operator x;  m := mat((x(1,1),x(1,2));
```

13.3 Matrix Expressions

These follow the normal rules of matrix algebra as defined by the following syntax:

```
<matrix expression> ::=
          MAT<matrix description>|<matrix variable>|
          <scalar expression>*<matrix expression>|
          <matrix expression>*<matrix expression>
          <matrix expression>+<matrix expression>|
          <matrix expression>^<integer>|
          <matrix expression>/<matrix expression>
```

Sums and products of matrix expressions must be of compatible size; otherwise an error will result during their evaluation. Similarly, only square matrices may be raised to a power. A negative power is computed as the inverse of the matrix raised to the corresponding positive power. `a/b` is interpreted as `a*b^(-1)`.

Examples:

Assuming `X` and `Y` have been declared as matrices, the following are matrix expressions

```
y
y^2*x-3*y^(-2)*x
y + mat((1,a),(b,c))/2
```

The computation of the quotient of two matrices normally uses a two-step elimination method due to Bareiss. An alternative method using Cramer's method is also available. This is usually less efficient than the Bareiss method unless the matrices are large and dense, although we have no solid statistics on this as yet. To use Cramer's method instead, the switch `CRAMER` should be turned on.

13.4 Operators with Matrix Arguments

The operator `LENGTH` applied to a matrix returns a list of the number of rows and columns in the matrix. Other operators useful in matrix calculations are defined in the following subsections. Attention is also drawn to the LINALG (chapter 52) and NORMFORM (chapter 57) packages.

13.4.1 DET Operator

Syntax:

```
DET(EXPRN:matrix_expression):algebraic.
```

The operator `DET` is used to represent the determinant of a square matrix expression. E.g.,

```
det(y^2)
```

is a scalar expression whose value is the determinant of the square of the matrix Y, and

```
det mat((a,b,c),(d,e,f),(g,h,j));
```

is a scalar expression whose value is the determinant of the matrix

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & j \end{pmatrix}$$

Determinant expressions have the *instant evaluation* property. In other words, the statement

```
let det mat((a,b),(c,d)) = 2;
```

sets the *value* of the determinant to 2, and does not set up a rule for the determinant itself.

13.4.2 MATEIGEN Operator

Syntax:

```
MATEIGEN(EXPRN:matrix_expression,ID):list.
```

`MATEIGEN` calculates the eigenvalue equation and the corresponding eigenvectors of a matrix, using the variable `ID` to denote the eigenvalue. A square free decomposition of the characteristic polynomial is carried out. The result is a list of lists of 3 elements, where the first element is a square free factor of the characteristic polynomial, the second its multiplicity and the third the corresponding eigenvector (as an n by 1 matrix). If the square free decomposition was successful, the product of the first elements in the lists is the minimal polynomial. In the case of degeneracy, several eigenvectors can exist for the same eigenvalue, which manifests itself in the appearance of more than one arbitrary variable in the eigenvector. To extract the various parts of the result use the operations defined on lists.

Example: The command

```
mateigen(mat((2,-1,1),(0,1,1),(-1,1,1)),eta);
```

gives the output

```
{{ETA - 1,2,

  [ARBCOMPLEX(1)]
  [             ]
  [ARBCOMPLEX(1)]
  [             ]
  [      0      ]

  },

 {ETA - 2,1,

  [      0      ]
  [             ]
  [ARBCOMPLEX(2)]
  [             ]
  [ARBCOMPLEX(2)]

  }}
```

13.4.3 TP Operator

Syntax:

```
TP(EXPRN:matrix_expression):matrix.
```

This operator takes a single matrix argument and returns its transpose.

13.4.4 Trace Operator

Syntax:

```
TRACE(EXPRN:matrix_expression):algebraic.
```

The operator `TRACE` is used to represent the trace of a square matrix.

13.4.5 Matrix Cofactors

Syntax:

```
COFACTOR(EXPRN:matrix_expression,ROW:integer,COLUMN:integer):
```

```
algebraic
```

The operator COFACTOR returns the cofactor of the element in row ROW and column COLUMN of the matrix MATRIX. Errors occur if ROW or COLUMN do not simplify to integer expressions or if MATRIX is not square.

13.4.6 NULLSPACE Operator

Syntax:

```
NULLSPACE(EXPRN:matrix_expression):list
```

NULLSPACE calculates for a matrix A a list of linear independent vectors (a basis) whose linear combinations satisfy the equation $Ax = 0$. The basis is provided in a form such that as many upper components as possible are isolated.

Note that with `b := nullspace a` the expression `length b` is the *nullity* of A, and that `second length a - length b` calculates the *rank* of A. The rank of a matrix expression can also be found more directly by the RANK operator described below.

Example: The command

```
nullspace mat((1,2,3,4),(5,6,7,8));
```

gives the output

```
{
 [ 1  ]
 [    ]
 [ 0  ]
 [    ]
 [ - 3]
 [    ]
 [ 2  ]
 ,
 [ 0  ]
 [    ]
 [ 1  ]
 [    ]
 [ - 2]
 [    ]
```

```
[ 1  ]
}
```

In addition to the REDUCE matrix form, NULLSPACE accepts as input a matrix given as a list of lists, that is interpreted as a row matrix. If that form of input is chosen, the vectors in the result will be represented by lists as well. This additional input syntax facilitates the use of NULLSPACE in applications different from classical linear algebra.

13.4.7 RANK Operator

Syntax:

```
RANK(EXPRN:matrix_expression):integer
```

RANK calculates the rank of its argument, that, like NULLSPACE can either be a standard matrix expression, or a list of lists, that can be interpreted either as a row matrix or a set of equations.

Example:

```
rank mat((a,b,c),(d,e,f));
```

returns the value 2.

13.5 Matrix Assignments

Matrix expressions may appear in the right-hand side of assignment statements. If the left-hand side of the assignment, which must be a variable, has not already been declared a matrix, it is declared by default to the size of the right-hand side. The variable is then set to the value of the right-hand side.

Such an assignment may be used very conveniently to find the solution of a set of linear equations. For example, to find the solution of the following set of equations

```
a11*x(1) + a12*x(2) = y1
a21*x(1) + a22*x(2) = y2
```

we simply write

```
x := 1/mat((a11,a12),(a21,a22))*mat((y1),(y2));
```

13.6 Evaluating Matrix Elements

Once an element of a matrix has been assigned, it may be referred to in standard array element notation. Thus `y(2,1)` refers to the element in the second row and first column of the matrix `Y`.

Chapter 14

Procedures

It is often useful to name a statement for repeated use in calculations with varying parameters, or to define a complete evaluation procedure for an operator. REDUCE offers a procedural declaration for this purpose. Its general syntax is:

```
[<procedural type>] PROCEDURE <name>[<varlist>];<statement>;
```

where

```
<varlist> ::= (<variable>,...,<variable>)
```

This will be explained more fully in the following sections.

In the algebraic mode of REDUCE the `<procedure type>` can be omitted, since the default is `ALGEBRAIC`. Procedures of type `INTEGER` or `REAL` may also be used. In the former case, the system checks that the value of the procedure is an integer. At present, such checking is not done for a real procedure, although this will change in the future when a more complete type checking mechanism is installed. Users should therefore only use these types when appropriate. An empty variable list may also be omitted.

All user-defined procedures are automatically declared to be operators.

In order to allow users relatively easy access to the whole REDUCE source program, system procedures are not protected against user redefinition. If a procedure is redefined, a message

```
*** <procedure name> REDEFINED
```

is printed. If this occurs, and the user is not redefining his own procedure, he is well advised to rename it, and possibly start over (because he has *already* redefined some internal procedure whose correct functioning may be required for his job!)

All required procedures should be defined at the top level, since they have global scope throughout a program. In particular, an attempt to define a procedure within a procedure will cause an error to occur.

14.1 Procedure Heading

Each procedure has a heading consisting of the word `PROCEDURE` (optionally preceded by the word `ALGEBRAIC`), followed by the name of the procedure to be defined, and followed by its formal parameters – the symbols that will be used in the body of the definition to illustrate what is to be done. There are three cases:

1. No parameters. Simply follow the procedure name with a terminator (semicolon or dollar sign).

   ```
   procedure abc;
   ```

   When such a procedure is used in an expression or command, `abc()`, with empty parentheses, must be written.

2. One parameter. Enclose it in parentheses *or* just leave at least one space, then follow with a terminator.

   ```
   procedure abc(x);
   ```

   or

   ```
   procedure abc x;
   ```

3. More than one parameter. Enclose them in parentheses, separated by commas, then follow with a terminator.

   ```
   procedure abc(x,y,z);
   ```

Referring to the last example, if later in some expression being evaluated the symbols `abc(u,p*q,123)` appear, the operations of the procedure body will be carried out as if `X` had the same value as `U` does, `Y` the same value as `p*q` does, and `Z` the value 123. The values of `X`, `Y`, `Z`, after the procedure body operations are completed are unchanged. So, normally, are the values of `U`, `P`, `Q`, and (of course) 123. (This is technically referred to as call by value.)

The reader will have noted the word *normally* a few lines earlier. The call by value protections can be bypassed if necessary, as described elsewhere.

14.2 Procedure Body

Following the delimiter that ends the procedure heading must be a *single* statement defining the action to be performed or the value to be delivered. A terminator must follow the statement. If it is a semicolon, the name of the procedure just defined is printed. It is not printed if a dollar sign is used.

If the result wanted is given by a formula of some kind, the body is just that formula, using the variables in the procedure heading.

Simple Example:

If `f(x)` is to mean `(x+5)*(x+6)/(x+7)`, the entire procedure definition could read

```
procedure f x; (x+5)*(x+6)/(x+7);
```

Then `f(10)` would evaluate to 240/17, `f(a-6)` to `A*(A-1)/(A+1)`, and so on.

More Complicated Example:

Suppose we need a function `p(n,x)` that, for any positive integer `N`, is the Legendre polynomial of order n. We can define this operator using the textbook formula defining these functions:

$$p_n(x) = \frac{1}{n!}\,\frac{d^n}{dy^n}\,\frac{1}{(y^2-2xy+1)^{\frac{1}{2}}}\Bigg|_{y=0}$$

Put into words, the Legendre polynomial $p_n(x)$ is the result of substituting $y=0$ in the n^{th} partial derivative with respect to y of a certain fraction involving x and y, then dividing that by $n!$.

This verbal formula can easily be written in REDUCE:

```
procedure p(n,x);
   sub(y=0,df(1/(y^2-2*x*y+1)^(1/2),y,n))
       /(for i:=1:n product i);
```

Having input this definition, the expression evaluation

```
2p(2,w);
```

would result in the output

```
   2
3*W  - 1 .
```

If the desired process is best described as a series of steps, then a group or compound statement can be used.

Example:

The above Legendre polynomial example can be rewritten as a series of steps instead of a single formula as follows:

```
procedure p(n,x);
  begin scalar seed,deriv,top,fact;
       seed:=1/(y^2 - 2*x*y +1)^(1/2);
       deriv:=df(seed,y,n);
       top:=sub(y=0,deriv);
       fact:=for i:=1:n product i;
       return top/fact
  end;
```

Procedures may also be defined recursively. In other words, the procedure body can include references to the procedure name itself, or to other procedures that themselves reference the given procedure. As an example, we can define the Legendre polynomial through its standard recurrence relation:

```
procedure p(n,x);
   if n<0 then rederr "Invalid argument to P(N,X)"
    else if n=0 then 1
    else if n=1 then x
    else ((2*n-1)*x*p(n-1,x)-(n-1)*p(n-2,x))/n;
```

The operator `REDERR` in the above example provides for a simple error exit from an algebraic procedure (and also a block). It can take a string as argument.

It should be noted however that all the above definitions of `p(n,x)` are quite inefficient if extensive use is to be made of such polynomials, since each call effectively recomputes all lower order polynomials. It would be better to store these expressions in an array, and then use say the recurrence relation to compute only those polynomials that have not already been derived. We leave it as an exercise for the reader to write such a definition.

14.3 Using LET Inside Procedures

By using `LET` instead of an assignment in the procedure body it is possible to bypass the call-by-value protection. If `X` is a formal parameter or local variable of the procedure (i.e. is in the heading or in a local declaration), and `LET` is used instead of `:=` to make an assignment to `X`, e.g.

```
let x = 123;
```

then it is the variable that is the value of X that is changed. This effect also occurs with local variables defined in a block. If the value of X is not a variable, but a more general expression, then it is that expression that is used on the left-hand side of the LET statement. For example, if X had the value p*q, it is as if let p*q = 123 had been executed.

14.4 LET Rules as Procedures

The LET statement offers an alternative syntax and semantics for procedure definition.

In place of

```
procedure abc(x,y,z); <procedure body>;
```

one can write

```
for all x,y,z let abc(x,y,z) = <procedure body>;
```

There are several differences to note.

If the procedure body contains an assignment to one of the formal parameters, e.g.

```
x := 123;
```

in the PROCEDURE case it is a variable holding a copy of the first actual argument that is changed. The actual argument is not changed.

In the LET case, the actual argument is changed. Thus, if ABC is defined using LET, and abc(u,v,w) is evaluated, the value of U changes to 123. That is, the LET form of definition allows the user to bypass the protections that are enforced by the call by value conventions of standard PROCEDURE definitions.

Example: We take our earlier FACTORIAL procedure and write it as a LET statement.

```
for all n let factorial n =
            begin scalar m,s;
            m:=1; s:=n;
```

```
          l1: if s=0 then return m;
              m:=m*s;
              s:=s-1;
              go to l1
          end;
```

The reader will notice that we introduced a new local variable, `S`, and set it equal to `N`. The original form of the procedure contained the statement `n:=n-1;`. If the user asked for the value of `factorial(5)` then `N` would correspond to, not just have the value of, 5, and REDUCE would object to trying to execute the statement $5 := 5 - 1$.

If `PQR` is a procedure with no parameters,

```
procedure pqr;
   <procedure body>;
```

it can be written as a `LET` statement quite simply:

```
let pqr = <procedure body>;
```

To call *procedure* `PQR`, if defined in the latter form, the empty parentheses would not be used: use `PQR` not `PQR()` where a call on the procedure is needed.

The two notations for a procedure with no arguments can be combined. `PQR` can be defined in the standard `PROCEDURE` form. Then a `LET` statement

```
let pqr = pqr();
```

would allow a user to use `PQR` instead of `PQR()` in calling the procedure.

A feature available with `LET`-defined procedures and not with procedures defined in the standard way is the possibility of defining partial functions.

```
for all x such that numberp x let uvw(x)=<procedure body>;
```

Now `UVW` of an integer would be calculated as prescribed by the procedure body, while `UVW` of a general argument, such as `Z` or `p+q` (assuming these evaluate to themselves) would simply stay `uvw(z)` or `uvw(p+q)` as the case may be.

14.5 REMEMBER Statement

Setting the remember option for an algebraic procedure by

```
REMEMBER (PROCNAME:procedure);
```

saves all intermediate results of such procedure evaluations, including recursive calls. Subsequent calls to the procedure can then be determined from the saved results, and thus the number of evaluations (or the complexity) can be reduced. This mode of evalation costs extra memory, of course. In addition, the procedure must be free of side–effects.

The following examples show the effect of the remember statement on two well–known examples.

```
procedure H(n);      % Hofstadter's function
 if numberp n then
 << cnn := cnn +1;   % counts the calls
 if n < 3 then 1 else H(n-H(n-1))+H(n-H(n-2))>>;

remember h;

> << cnn := 0; H(100); cnn>>;

100

% H has been called 100 times only.

procedure A(m,n);    % Ackermann function

 if m=0 then n+1 else
  if n=0 then A(m-1,1) else
  A(m-1,A(m,n-1));

remember a;

A(3,3);
```

Chapter 15

User Contributed Packages

The complete REDUCE system includes a number of packages contributed by users that are provided as a service to the user community. Questions regarding these packages should be directed to their individual authors.

All such packages have been precompiled as part of the installation process. However, many must be specifically loaded before they can be used. (Those that are loaded automatically are so noted in their description.) You should also consult the user notes for your particular implementation for further information on whether this is necessary. If it is, the relevant command is `LOAD_PACKAGE`, which takes a list of one or more package names as argument, for example:

```
load_package algint;
```

although this syntax may vary from implementation to implementation.

Nearly all these packages come with separate documentation and test files (except those noted here that have no additional documentation), which is included, along with the source of the package, in the REDUCE system distribution. These items should be studied for any additional details on the use of a particular package.

Part 2 of this manual contains short documentation for the packages

- ALGINT: Integration of square roots (chapter 20);
- APPLYSYM: Infinitesimal symmetries of differential equations (chap-

ter 21);

- ARNUM: An algebraic number package (chapter 22);
- ASSIST: Useful utilities for various applications (chapter 23);
- AVECTOR: A vector algebra and calculus package (chapter 25);
- BOOLEAN: A package for boolean algebra (chapter 26);
- CALI: A package for computational commutative algebra (chapter 27);
- CAMAL: Calculations in celestial mechanics (chapter 28);
- CHANGEVR: Change of Independent Variable(s) in DEs (chapter 30);
- COMPACT: Package for compacting expressions (chapter 31);
- CONTFR: Approximation of a number by continued fractions (chapter **??**);
- CRACK: Solving overdetermined systems of PDEs or ODEs (chapter 32);
- CVIT: Fast calculation of Dirac gamma matrix traces (chapter 33);
- DEFINT: A definite integration interface for REDUCE (chapter 34);
- DESIR: Differential linear homogeneous equation solutions in the neighborhood of irregular and regular singular points (chapter 35);
- DFPART: Derivatives of generic functions (chapter 36);
- DUMMY: Canonical form of expressions with dummy variables (chapter 37);
- EXCALC: A differential geometry package (chapter 39);
- FPS: Automatic calculation of formal power series (chapter 41);
- FIDE: Finite difference method for partial differential equations (chapter 40);
- GENTRAN: A code generation package (chapter 42);
- GNUPLOT: Display of functions and surfaces (chapter 44);

- GROEBNER: A Gröbner basis package (chapter 45);
- IDEALS: Arithmetic for polynomial ideals (chapter 46);
- INEQ: Support for solving inequalities (chapter 47);
- INVBASE: A package for computing involutive bases (chapter 48);
- LAPLACE: Laplace and inverse Laplace transforms (chapter 49);
- LIE: Functions for the classification of real n-dimensional Lie algebras (chapter 50);
- LIMITS: A package for finding limits (chapter 51);
- LINALG: Linear algebra package (chapter 52);
- MODSR: Modular solve and roots (chapter 54);
- NCPOLY: Non–commutative polynomial ideals (chapter 56);
- NORMFORM: Computation of matrix normal forms (chapter 57);
- NUMERIC: Solving numerical problems (chapter 58);
- ODESOLVE: Ordinary differential equations solver (chapter 59);
- ORTHOVEC: Manipulation of scalars and vectors (chapter 60);
- PHYSOP: Operator calculus in quantum theory (chapter 61);
- PM: A REDUCE pattern matcher (chapter 62);
- RANDPOLY: A random polynomial generator (chapter 64);
- REACTEQN: Support for chemical reaction equation systems (chapter 66);
- RESET: Code to reset REDUCE to its initial state (chapter 68);
- RESIDUE: A residue package (chapter 69);
- RLFI: REDUCE LaTeX formula interface (chapter 70);
- RSOLVE: Rational/integer polynomial solvers (chapter 72);
- ROOTS: A REDUCE root finding package (chapter 71);

- SCOPE: REDUCE source code optimization package (chapter 73);
- SETS: A basic set theory package (chapter 74);
- SPDE: A package for finding symmetry groups of PDE's (chapter 76);
- SPECFN: Package for special functions (chapter 77);
- SPECFN2: Package for special special functions (chapter 78);
- SUM: A package for series summation (chapter 79);
- SYMMETRY: Operations on symmetric matrices (chapter 81);
- TAYLOR: Manipulation of Taylor series (chapter 82);
- TPS: A truncated power series package (chapter 83);
- TRI: TeX REDUCE interface (chapter 84);
- TRIGSIMP: Simplification and factorization of trigonometric and hyperbolic functions (chapter 85);
- XCOLOR: Calculation of the color factor in non-abelian gauge field theories (chapter 87);
- XIDEAL: Gröbner Bases for exterior algebra (chapter 88);
- WU: Wu algorithm for polynomial systems (chapter 86);
- ZEILBERG: A package for indefinite and definite summation (chapter 89);
- ZTRANS: Z-transform package (chapter 90);

Chapter 16

Symbolic Mode

At the system level, REDUCE is based on a version of the programming language Lisp known as *Standard Lisp* which is described in J. Marti, Hearn, A. C., Griss, M. L. and Griss, C., "Standard LISP Report" SIGPLAN Notices, ACM, New York, 14, No 10 (1979) 48-68. We shall assume in this section that the reader is familiar with the material in that paper. This also assumes implicitly that the reader has a reasonable knowledge about Lisp in general, say at the level of the LISP 1.5 Programmer's Manual (McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. and Levin, M. I., "LISP 1.5 Programmer's Manual", M.I.T. Press, 1965) or any of the books mentioned at the end of this section. Persons unfamiliar with this material will have some difficulty understanding this section.

Although REDUCE is designed primarily for algebraic calculations, its source language is general enough to allow for a full range of Lisp-like symbolic calculations. To achieve this generality, however, it is necessary to provide the user with two modes of evaluation, namely an algebraic mode and a symbolic mode. To enter symbolic mode, the user types `symbolic;` (or `lisp;`) and to return to algebraic mode one types `algebraic;`. Evaluations proceed differently in each mode so the user is advised to check what mode he is in if a puzzling error arises. He can find his mode by typing

```
eval_mode;
```

The current mode will then be printed as `ALGEBRAIC` or `SYMBOLIC`.

Expression evaluation may proceed in either mode at any level of a calculation, provided the results are passed from mode to mode in a compatible manner. One simply prefixes the relevant expression by the appropriate mode. If the mode name prefixes an expression at the top level, it will then be handled as if the global system mode had been changed for the scope of that particular calculation.

For example, if the current mode is `ALGEBRAIC`, then the commands

```
symbolic car '(a);
x+y;
```

will cause the first expression to be evaluated and printed in symbolic mode and the second in algebraic mode. Only the second evaluation will thus affect the expression workspace. On the other hand, the statement

```
x + symbolic car '(12);
```

will result in the algebraic value `X+12`.

The use of `SYMBOLIC` (and equivalently `ALGEBRAIC`) in this manner is the same as any operator. That means that parentheses could be omitted in the above examples since the meaning is obvious. In other cases, parentheses must be used, as in

```
symbolic(x := 'a);
```

Omitting the parentheses, as in

```
symbolic x := a;
```

would be wrong, since it would parse as

```
symbolic(x) := a;
```

For convenience, it is assumed that any operator whose *first* argument is quoted is being evaluated in symbolic mode, regardless of the mode in effect at that time. Thus, the first example above could be equally well written:

```
car '(a);
```

Except where explicit limitations have been made, most REDUCE algebraic constructions carry over into symbolic mode. However, there are some differences. First, expression evaluation now becomes Lisp evaluation. Secondly, assignment statements are handled differently, as we shall discuss shortly. Thirdly, local variables and array elements are initialized to `NIL` rather than `0`. (In fact, any variables not explicitly declared `INTEGER` are also initialized to `NIL` in algebraic mode, but the algebraic evaluator recognizes `NIL` as `0`.) Finally, function definitions follow the conventions of Standard Lisp.

To begin with, we mention a few extensions to our basic syntax which are designed primarily if not exclusively for symbolic mode.

16.1 Symbolic Infix Operators

There are three binary infix operators in REDUCE intended for use in symbolic mode, namely `. (CONS), EQ and MEMQ`. The precedence of these operators was given in another section.

16.2 Symbolic Expressions

These consist of scalar variables and operators and follow the normal rules of the Lisp meta language.

Examples:

```
x
car u . reverse v
simp (u+v^2)
```

16.3 Quoted Expressions

Because symbolic evaluation requires that each variable or expression has a value, it is necessary to add to REDUCE the concept of a quoted expression by analogy with the Lisp `QUOTE` function. This is provided by the single quote mark `'`. For example,

`'a`	represents the Lisp S-expression `(quote a)`
`'(a b c)`	represents the Lisp S-expression `(quote (a b c))`

Note, however, that strings are constants and therefore evaluate to themselves in symbolic mode. Thus, to print the string `"A String"`, one would write

```
prin2 "A String";
```

Within a quoted expression, identifier syntax rules are those of REDUCE. Thus `(A !. B)` is the list consisting of the three elements `A`, `.`, and `B`, whereas `(A . B)` is the dotted pair of `A` and `B`.

16.4 Lambda Expressions

LAMBDA expressions provide the means for constructing Lisp LAMBDA expressions in symbolic mode. They may not be used in algebraic mode.

Syntax:

```
<LAMBDA expression> ::=
        LAMBDA <varlist><terminator><statement>
```

where

```
<varlist> ::= (<variable>,...,<variable>)
```

e.g.,

```
lambda (x,y); car x . cdr y;
```

is equivalent to the Lisp LAMBDA expression

```
(lambda (x y) (cons (car x) (cdr y)))
```

The parentheses may be omitted in specifying the variable list if desired.

LAMBDA expressions may be used in symbolic mode in place of prefix operators, or as an argument of the reserved word FUNCTION.

In those cases where a LAMBDA expression is used to introduce local variables to avoid recomputation, a WHERE statement can also be used. For example, the expression

```
(lambda (x,y); list(car x,cdr x,car y,cdr y))
    (reverse u,reverse v)
```

can also be written

```
{car x,cdr x,car y,cdr y} where x=reverse u,y=reverse v
```

Where possible, WHERE syntax is preferred to LAMBDA syntax, since it is more natural.

16.5 Symbolic Assignment Statements

In symbolic mode, if the left side of an assignment statement is a variable, a SETQ of the right-hand side to that variable occurs. If the left-hand side is an expression, it must be of the form of an array element, otherwise an error will result. For example, x:=y translates into (SETQ X Y) whereas a(3) := 3 will be valid if A has been previously declared a single dimensioned array of at least four elements.

16.6 FOR EACH Statement

The FOR EACH form of the FOR statement, designed for iteration down a list, is more general in symbolic mode. Its syntax is:

```
FOR EACH ID:identifier {IN|ON} LST:list
    {DO|COLLECT|JOIN|PRODUCT|SUM} EXPRN:S-expr
```

As in algebraic mode, if the keyword IN is used, iteration is on each element of the list. With ON, iteration is on the whole list remaining at each point in the iteration. As a result, we have the following equivalence between each form of FOR EACH and the various mapping functions in Lisp:

	DO	COLLECT	JOIN
IN	MAPC	MAPCAR	MAPCAN
ON	MAP	MAPLIST	MAPCON

Example: To list each element of the list (a b c):

```
for each x in '(a b c) collect list x;
```

16.7 Symbolic Procedures

All the functions described in the Standard Lisp Report are available to users in symbolic mode. Additional functions may also be defined as symbolic procedures. For example, to define the Lisp function ASSOC, the following could be used:

```
symbolic procedure assoc(u,v);
```

```
if null v then nil
 else if u = caar v then car v
 else assoc(u, cdr v);
```

If the default mode were symbolic, then `SYMBOLIC` could be omitted in the above definition. `MACRO`s may be defined by prefixing the keyword `PROCEDURE` by the word `MACRO`. (In fact, ordinary functions may be defined with the keyword `EXPR` prefixing `PROCEDURE` as was used in the Standard Lisp Report.) For example, we could define a `MACRO CONSCONS` by

```
symbolic macro procedure conscons l;
   expand(cdr l,'cons);
```

Another form of macro, the `SMACRO` is also available. These are described in the Standard Lisp Report. The Report also defines a function type `FEXPR`. However, its use is discouraged since it is hard to implement efficiently, and most uses can be replaced by macros. At the present time, there are no `FEXPR`s in the core REDUCE system.

16.8 Standard Lisp Equivalent of Reduce Input

A user can obtain the Standard Lisp equivalent of his REDUCE input by turning on the switch `DEFN` (for definition). The system then prints the Lisp translation of his input but does not evaluate it. Normal operation is resumed when `DEFN` is turned off.

16.9 Communicating with Algebraic Mode

One of the principal motivations for a user of the algebraic facilities of REDUCE to learn about symbolic mode is that it gives one access to a wider range of techniques than is possible in algebraic mode alone. For example, if a user wishes to use parts of the system defined in the basic system source code, or refine their algebraic code definitions to make them more efficient, then it is necessary to understand the source language in fairly complete detail. Moreover, it is also necessary to know a little more about the way REDUCE operates internally. Basically, REDUCE considers expressions in two forms: prefix form, which follow the normal Lisp rules of function composition, and so-called canonical form, which uses a completely different syntax.

Once these details are understood, the most critical problem faced by a user is how to make expressions and procedures communicate between symbolic and algebraic mode. The purpose of this section is to teach a user the basic principles for this.

If one wants to evaluate an expression in algebraic mode, and then use that expression in symbolic mode calculations, or vice versa, the easiest way to do this is to assign a variable to that expression whose value is easily obtainable in both modes. To facilitate this, a declaration `SHARE` is available. `SHARE` takes a list of identifiers as argument, and marks these variables as having recognizable values in both modes. The declaration may be used in either mode.

E.g.,

```
share x,y;
```

says that `X` and `Y` will receive values to be used in both modes.

If a `SHARE` declaration is made for a variable with a previously assigned algebraic value, that value is also made available in symbolic mode.

16.9.1 Passing Algebraic Mode Values to Symbolic Mode

If one wishes to work with parts of an algebraic mode expression in symbolic mode, one simply makes an assignment of a shared variable to the relevant expression in algebraic mode. For example, if one wishes to work with `(a+b)^2`, one would say, in algebraic mode:

```
x := (a+b)^2;
```

assuming that `X` was declared shared as above. If we now change to symbolic mode and say

```
x;
```

its value will be printed as a prefix form with the syntax:

```
(*SQ <standard quotient> T)
```

This particular format reflects the fact that the algebraic mode processor currently likes to transfer prefix forms from command to command, but doesn't like to reconvert standard forms (which represent polynomials) and standard quotients back to a true Lisp prefix form for the expression (which would result in excessive computation). So `*SQ` is used to tell the algebraic processor that it is dealing with a prefix form which is really a standard quotient and the second argument (`T` or `NIL`) tells it whether it needs further processing (essentially, an *already simplified* flag).

So to get the true standard quotient form in symbolic mode, one needs `CADR` of the variable. E.g.,

```
z := cadr x;
```

would store in `Z` the standard quotient form for `(a+b)^2`.

Once you have this expression, you can now manipulate it as you wish. To facilitate this, a standard set of selectors and constructors are available for getting at parts of the form. Those presently defined are as follows:

REDUCE Selectors

`DENR`	denominator of standard quotient
`LC`	leading coefficient of polynomial
`LDEG`	leading degree of polynomial
`LPOW`	leading power of polynomial
`LT`	leading term of polynomial
`MVAR`	main variable of polynomial
`NUMR`	numerator (of standard quotient)
`PDEG`	degree of a power
`RED`	reductum of polynomial
`TC`	coefficient of a term
`TDEG`	degree of a term
`TPOW`	power of a term

REDUCE Constructors

`.+`	add a term to a polynomial
`./`	divide (two polynomials to get quotient)
`.*`	multiply power by coefficient to produce term
`.^`	raise a variable to a power

For example, to find the numerator of the standard quotient above, one could say:

```
numr z;
```

or to find the leading term of the numerator:

```
lt numr z;
```

Conversion between various data structures is facilitated by the use of a set of functions defined for this purpose. Those currently implemented include:

`!*A2F` convert an algebraic expression to a standard form. If result is rational, an error results;

`!*A2K` converts an algebraic expression to a kernel. If this is not possible, an error results;

`!*F2A` converts a standard form to an algebraic expression;

`!*F2Q` convert a standard form to a standard quotient;

`!*K2F` convert a kernel to a standard form;

`!*K2Q` convert a kernel to a standard quotient;

`!*P2F` convert a standard power to a standard form;

`!*P2Q` convert a standard power to a standard quotient;

`!*Q2F` convert a standard quotient to a standard form. If the quotient denominator is not 1, an error results;

`!*Q2K` convert a standard quotient to a kernel. If this is not possible, an error results;

`!*T2F` convert a standard term to a standard form

`!*T2Q` convert a standard term to a standard quotient.

16.9.2 Passing Symbolic Mode Values to Algebraic Mode

In order to pass the value of a shared variable from symbolic mode to algebraic mode, the only thing to do is make sure that the value in symbolic mode is a prefix expression. E.g., one uses `(expt (plus a b) 2)` for `(a+b)^2`, or the format `(*sq <standard quotient> t)` as described above. However, if you have been working with parts of a standard form they will probably not be in this form. In that case, you can do the following:

1. If it is a standard quotient, call `PREPSQ` on it. This takes a standard quotient as argument, and returns a prefix expression. Alternatively, you can call `MK!*SQ` on it, which returns a prefix form like `(*SQ <standard quotient> T)` and avoids translation of the expression into a true prefix form.

2. If it is a standard form, call `PREPF` on it. This takes a standard form as argument, and returns the equivalent prefix expression. Alternatively,

you can convert it to a standard quotient and then call MK!*SQ.

3. If it is a part of a standard form, you must usually first build up a standard form out of it, and then go to step 2. The conversion functions described earlier may be used for this purpose. For example,
   (a) If Z is an expression which is a term, !*T2F Z is a standard form.
   (b) If Z is a standard power, !*P2F Z is a standard form.
   (c) If Z is a variable, you can pass it direct to algebraic mode.

For example, to pass the leading term of (a+b)^2 back to algebraic mode, one could say:

```
y:= mk!*sq !*t2q lt numr z;
```

where Y has been declared shared as above. If you now go back to algebraic mode, you can work with Y in the usual way.

16.9.3 Complete Example

The following is the complete code for doing the above steps. The end result will be that the square of the leading term of $(a+b)^2$ is calculated.

```
share x,y;                  % declare X and Y as shared
x := (a+b)^2;               % store (a+b)^2 in X
symbolic;                   % transfer to symbolic mode
z := cadr x;                % store a true standard quotient in Z
lt numr z;                  % print the leading term of the
                            % numerator of Z
y := mk!*sq !*t2q numr z;   % store the prefix form of this
                            % leading term in Y
algebraic;                  % return to algebraic mode
y^2;                        % evaluate square of the leading term
                            % of (a+b)^2
```

16.9.4 Defining Procedures for Intermode Communication

If one wishes to define a procedure in symbolic mode for use as an operator in algebraic mode, it is necessary to declare this fact to the system by using the declaration OPERATOR in symbolic mode. Thus

```
symbolic operator leadterm;
```

would declare the procedure `LEADTERM` as an algebraic operator. This declaration *must* be made in symbolic mode as the effect in algebraic mode is different. The value of such a procedure must be a prefix form.

The algebraic processor will pass arguments to such procedures in prefix form. Therefore if you want to work with the arguments as standard quotients you must first convert them to that form by using the function `SIMP!*`. This function takes a prefix form as argument and returns the evaluated standard quotient.

For example, if you want to define a procedure `LEADTERM` which gives the leading term of an algebraic expression, one could do this as follows:

```
symbolic operator leadterm; % Declare LEADTERM as a symbolic
                            % mode procedure to be used in
                            % algebraic mode.

symbolic procedure leadterm u; % Define LEADTERM.
   mk!*sq !*t2q lt numr simp!* u;
```

Note that this operator has a different effect than the operator `LTERM` . In the latter case, the calculation is done with respect to the second argument of the operator. In the example here, we simply extract the leading term with respect to the system's choice of main variable.

Finally, if you wish to use the algebraic evaluator on an argument in a symbolic mode definition, the function `REVAL` can be used. The one argument of `REVAL` must be the prefix form of an expression. `REVAL` returns the evaluated expression as a true Lisp prefix form.

16.10 Rlisp '88

Rlisp '88 is a superset of the Rlisp that has been traditionally used for the support of REDUCE. It is fully documented in the book Marti, J.B., "RLISP '88: An Evolutionary Approach to Program Design and Reuse", World Scientific, Singapore (1993). Rlisp '88 adds to the traditional Rlisp the following facilities:

1. more general versions of the looping constructs `for`, `repeat` and `while`;

2. support for a backquote construct;
3. support for active comments;
4. support for vectors of the form name[index];
5. support for simple structures;
6. support for records.

In addition, "–" is a letter in Rlisp '88. In other words, `A-B` is an identifier, not the difference of the identifiers `A` and `B`. If the latter construct is required, it is necessary to put spaces around the - character. For compatibility between the two versions of Rlisp, we recommend this convention be used in all symbolic mode programs.

To use Rlisp '88, type `on rlisp88;`. This switches to symbolic mode with the Rlisp '88 syntax and extensions. While in this environment, it is impossible to switch to algebraic mode, or prefix expressions by "algebraic". However, symbolic mode programs written in Rlisp '88 may be run in algebraic mode provided the rlisp88 package has been loaded. We also expect that many of the extensions defined in Rlisp '88 will migrate to the basic Rlisp over time. To return to traditional Rlisp or to switch to algebraic mode, say "off rlisp88".

16.11 References

There are a number of useful books which can give you further information about LISP. Here is a selection:

Allen, J.R., "The Anatomy of LISP", McGraw Hill, New York, 1978.

McCarthy J., P.W. Abrahams, J. Edwards, T.P. Hart and M.I. Levin, "LISP 1.5 Programmer's Manual", M.I.T. Press, 1965.

Touretzky, D.S, "LISP: A Gentle Introduction to Symbolic Computation", Harper & Row, New York, 1984.

Winston, P.H. and Horn, B.K.P., "LISP", Addison-Wesley, 1981.

Chapter 17

Calculations in High Energy Physics

A set of REDUCE commands is provided for users interested in symbolic calculations in high energy physics. Several extensions to our basic syntax are necessary, however, to allow for the different data structures encountered.

17.1 High Energy Physics Operators

We begin by introducing three new operators required in these calculations.

17.1.1 . (Cons) Operator

Syntax:

```
        (EXPRN1:vector_expression)
                . (EXPRN2:vector_expression):algebraic.
```

The binary . operator, which is normally used to denote the addition of an element to the front of a list, can also be used in algebraic mode to denote the scalar product of two Lorentz four-vectors. For this to happen, the second argument must be recognizable as a vector expression at the time of evaluation. With this meaning, this operator is often referred to as the *dot* operator. In the present system, the index handling routines all assume that Lorentz four-vectors are used, but these routines could be rewritten to handle other cases.

Components of vectors can be represented by including representations of unit vectors in the system. Thus if `E0` represents the unit vector `(1,0,0,0)`, `(p.eo)` represents the zeroth component of the four-vector P. Our metric and notation follows Bjorken and Drell "Relativistic Quantum Mechanics" (McGraw-Hill, New York, 1965). Similarly, an arbitrary component `P` may be represented by `(p.u)`. If contraction over components of vectors is required, then the declaration `INDEX` must be used. Thus

```
index u;
```

declares `U` as an index, and the simplification of

```
p.u * q.u
```

would result in

```
P.Q
```

The metric tensor $g^{\mu\nu}$ may be represented by `(u.v)`. If contraction over `U` and `V` is required, then they should be declared as indices.

Errors occur if indices are not properly matched in expressions.

If a user later wishes to remove the index property from specific vectors, he can do it with the declaration `REMIND`. Thus `remind v1...vn;` removes the index flags from the variables `V1` through `Vn`. However, these variables remain vectors in the system.

17.1.2 G Operator for Gamma Matrices

Syntax:

```
G(ID:identifier[,EXPRN:vector_expression])
        :gamma_matrix_expression.
```

`G` is an n-ary operator used to denote a product of γ matrices contracted with Lorentz four-vectors. Gamma matrices are associated with fermion lines in a Feynman diagram. If more than one such line occurs, then a different set of γ matrices (operating in independent spin spaces) is required to represent each line. To facilitate this, the first argument of `G` is a line identification identifier (not a number) used to distinguish different lines.

Thus

```
g(l1,p) * g(l2,q)
```

denotes the product of γ`.p` associated with a fermion line identified as `L1`, and γ`.q` associated with another line identified as `L2` and where `p` and `q` are Lorentz four-vectors. A product of γ matrices associated with the same line may be written in a contracted form.

Thus

```
g(l1,p1,p2,...,p3) = g(l1,p1)*g(l1,p2)*...*g(l1,p3) .
```

The vector `A` is reserved in arguments of G to denote the special γ matrix γ^5. Thus

`g(l,a)`	$=$	γ^5	associated with the line `L`
`g(l,p,a)`	$=$	$\gamma.\mathrm{p}\ \times\gamma^5$	associated with the line `L`.

γ^μ (associated with the line `L`) may be written as `g(l,u)`, with `U` flagged as an index if contraction over `U` is required.

The notation of Bjorken and Drell is assumed in all operations involving γ matrices.

17.1.3 EPS Operator

Syntax:

```
EPS(EXPRN1:vector_expression,...,EXPRN4:vector_exp)
   :vector_exp.
```

The operator `EPS` has four arguments, and is used only to denote the completely antisymmetric tensor of order 4 and its contraction with Lorentz four-vectors. Thus

$$\epsilon_{ijkl} = \begin{cases} +1 & \text{if } i,j,k,l \text{ is an even permutation of 0,1,2,3} \\ -1 & \text{if an odd permutation} \\ 0 & \text{otherwise} \end{cases}$$

A contraction of the form $\epsilon_{ij\mu\nu}p_\mu q_\nu$ may be written as `eps(i,j,p,q)`, with `I` and `J` flagged as indices, and so on.

17.2 Vector Variables

Apart from the line identification identifier in the `G` operator, all other arguments of the operators in this section are vectors. Variables used as such must be declared so by the type declaration `VECTOR`, for example:

```
vector  p1,p2;
```

declares `P1` and `P2` to be vectors. Variables declared as indices or given a mass are automatically declared vector by these declarations.

17.3 Additional Expression Types

Two additional expression types are necessary for high energy calculations, namely

17.3.1 Vector Expressions

These follow the normal rules of vector combination. Thus the product of a scalar or numerical expression and a vector expression is a vector, as are the sum and difference of vector expressions. If these rules are not followed, error messages are printed. Furthermore, if the system finds an undeclared variable where it expects a vector variable, it will ask the user in interactive mode whether to make that variable a vector or not. In batch mode, the declaration will be made automatically and the user informed of this by a message.

`Examples:`

Assuming `P` and `Q` have been declared vectors, the following are vector expressions

```
p
2*q/3
2*x*y*p - p.q*q/(3*q.q)
```

whereas `p*q` and `p/q` are not.

17.3.2 Dirac Expressions

These denote those expressions which involve γ matrices. A γ matrix is implicitly a 4×4 matrix, and so the product, sum and difference of such expressions, or the product of a scalar and Dirac expression is again a Dirac expression. There are no Dirac variables in the system, so whenever a scalar variable appears in a Dirac expression without an associated γ matrix expression, an implicit unit 4 by 4 matrix is assumed. For example, `g(l,p) + m` denotes `g(l,p) + m*<unit 4 by 4 matrix>`. Multiplication of Dirac expressions, as for matrix expressions, is of course non-commutative.

17.4 Trace Calculations

When a Dirac expression is evaluated, the system computes one quarter of the trace of each γ matrix product in the expansion of the expression. One quarter of each trace is taken in order to avoid confusion between the trace of the scalar `M`, say, and `M` representing `M * <unit 4 by 4 matrix>`. Contraction over indices occurring in such expressions is also performed. If an unmatched index is found in such an expression, an error occurs.

The algorithms used for trace calculations are the best available at the time this system was produced. For example, in addition to the algorithm developed by Chisholm for contracting indices in products of traces, REDUCE uses the elegant algorithm of Kahane for contracting indices in γ matrix products. These algorithms are described in Chisholm, J. S. R., Il Nuovo Cimento X, 30, 426 (1963) and Kahane, J., Journal Math. Phys. 9, 1732 (1968).

It is possible to prevent the trace calculation over any line identifier by the declaration `NOSPUR`. For example,

```
nospur l1,l2;
```

will mean that no traces are taken of γ matrix terms involving the line numbers `L1` and `L2`. However, in some calculations involving more than one line, a catastrophic error

```
This NOSPUR option not implemented
```

can occur (for the reason stated!) If you encounter this error, please let us know!

A trace of a γ matrix expression involving a line identifier which has been declared `NOSPUR` may be later taken by making the declaration `SPUR`.

See also the CVIT package for an alternative mechanism (chapter 33).

17.5 Mass Declarations

It is often necessary to put a particle "on the mass shell" in a calculation. This can, of course, be accomplished with a `LET` command such as

```
let p.p= m^2;
```

but an alternative method is provided by two commands `MASS` and `MSHELL`. `MASS` takes a list of equations of the form:

```
<vector variable> = <scalar variable>
```

for example,

```
mass p1=m, q1=mu;
```

The only effect of this command is to associate the relevant scalar variable as a mass with the corresponding vector. If we now say

```
mshell <vector variable>,...,<vector variable>;
```

and a mass has been associated with these arguments, a substitution of the form

```
<vector variable>.<vector variable> = <mass>^2
```

is set up. An error results if the variable has no preassigned mass.

17.6 Example

We give here as an example of a simple calculation in high energy physics the computation of the Compton scattering cross-section as given in Bjorken and Drell Eqs. (7.72) through (7.74). We wish to compute the trace of

$$\frac{\alpha^2}{2}\left(\frac{k'}{k}\right)^2\left(\frac{\gamma.p_f+m}{2m}\right)\left(\frac{\gamma.e'\gamma.e\gamma.k_i}{2k.p_i}+\frac{\gamma.e\gamma.e'\gamma.k_f}{2k'.p_i}\right)\left(\frac{\gamma.p_i+m}{2m}\right)$$
$$\left(\frac{\gamma.k_i\gamma.e\gamma.e'}{2k.p_i}+\frac{\gamma.k_f\gamma.e'\gamma.e}{2k'.p_i}\right)$$

where k_i and k_f are the four-momenta of incoming and outgoing photons (with polarization vectors e and e' and laboratory energies k and k' respectively) and p_i, p_f are incident and final electron four-momenta.

Omitting therefore an overall factor $\frac{\alpha^2}{2m^2}\left(\frac{k'}{k}\right)^2$ we need to find one quarter of the trace of

$$(\gamma.p_f+m)\left(\frac{\gamma.e'\gamma.e\gamma.k_i}{2k.p_i}+\frac{\gamma.e\gamma.e'\gamma.k_f}{2k'.p_i}\right)(\gamma.p_i+m)$$
$$\left(\frac{\gamma.k_i\gamma.e\gamma.e'}{2k.p_i}+\frac{\gamma.k_f\gamma.e'\gamma.e}{2k'.p_i}\right)$$

A straightforward REDUCE program for this, with appropriate substitutions (using `P1` for p_i, `PF` for p_f, `KI` for k_i and `KF` for k_f) is

```
on div; % this gives output in same form as Bjorken and Drell.
mass ki= 0, kf= 0, p1= m, pf= m; vector e,ep;
% if e is used as a vector, it loses its scalar identity as
%      the base of natural logarithms.
mshell ki,kf,p1,pf;
let p1.e= 0, p1.ep= 0, p1.pf= m^2+ki.kf, p1.ki= m*k,p1.kf=
    m*kp, pf.e= -kf.e, pf.ep= ki.ep, pf.ki= m*kp, pf.kf=
    m*k, ki.e= 0, ki.kf= m*(k-kp), kf.ep= 0, e.e= -1,
    ep.ep=-1;
for all p let gp(p)= g(l,p)+m;
comment this is just to save us a lot of writing;
gp(pf)*(g(l,ep,e,ki)/(2*ki.p1) + g(l,e,ep,kf)/(2*kf.p1))
  * gp(p1)*(g(l,ki,e,ep)/(2*ki.p1) + g(l,kf,ep,e)/
    (2*kf.p1))$
write "The Compton cxn is",ws;
```

(We use `P1` instead of `PI` in the above to avoid confusion with the reserved variable `PI`).

This program will print the following result

```
                             (-1)         (-1)              2
The Compton cxn is 1/2*K*KP     + 1/2*K    *KP + 2*E.EP  - 1
```

17.7 Extensions to More Than Four Dimensions

In our discussion so far, we have assumed that we are working in the normal four dimensions of QED calculations. However, in most cases, the programs will also work in an arbitrary number of dimensions. The command

```
vecdim <expression>;
```

sets the appropriate dimension. The dimension can be symbolic as well as numerical. Users should note however, that the `EPS` operator and the γ_5 symbol (`A`) are not properly defined in other than four dimensions and will lead to an error if used.

Chapter 18

REDUCE and Rlisp Utilities

REDUCE and its associated support language system Rlisp include a number of utilities which have proved useful for program development over the years. The following are supported in most of the implementations of REDUCE currently available.

18.1 The Standard Lisp Compiler

Many versions of REDUCE include a Standard Lisp compiler that is automatically loaded on demand. You should check your system specific user guide to make sure you have such a compiler. To make the compiler active, the switch `COMP` should be turned on. Any further definitions input after this will be compiled automatically. If the compiler used is a derivative version of the original Griss-Hearn compiler, (M. L. Griss and A. C. Hearn, "A Portable LISP Compiler", SOFTWARE — Practice and Experience 11 (1981) 541-605), there are other switches that might also be used in this regard. However, these additional switches are not supported in all compilers. They are as follows:

`PLAP` If ON, causes the printing of the portable macros produced by the compiler;

`PGWD` If ON, causes the printing of the actual assembly language instructions generated from the macros;

`PWRDS` If ON, causes a statistic message of the form
`<function> COMPILED, <words> WORDS, <words> LEFT`
to be printed. The first number is the number of words of binary program space the compiled function took, and the second number the number of words left unused in binary program space.

18.2 Fast Loading Code Generation Program

In most versions of REDUCE, it is possible to take any set of Lisp, Rlisp or REDUCE commands and build a fast loading version of them. In Rlisp or REDUCE, one does the following:

```
faslout <filename>;
<commands or IN statements>
faslend;
```

To load such a file, one uses the command `LOAD`, e.g. `load foo;` or `load foo,bah;`

This process produces a fast-loading version of the original file. In some implementations, this means another file is created with the same name but a different extension. For example, in PSL-based systems, the extension is `b` (for binary). In CSL-based systems, however, this process adds the fast-loading code to a single file in which all such code is stored. Particular functions are provided by CSL for managing this file, and described in the CSL user documentation.

In doing this build, as with the production of a Standard Lisp form of such statements, it is important to remember that some of the commands must be instantiated during the building process. For example, macros must be expanded, and some property list operations must happen. The REDUCE sources should be consulted for further details on this.

To avoid excessive printout, input statements should be followed by a $ instead of the semicolon. With `LOAD` however, the input doesn't print out regardless of which terminator is used with the command.

If you subsequently change the source files used in producing a fast loading file, don't forget to repeat the above process in order to update the fast loading file correspondingly. Remember also that the text which is read in during the creation of the fast load file, in the compiling process described above, is *not* stored in your REDUCE environment, but only translated and output. If you want to use the file just created, you must then use `LOAD` to load the output of the fast-loading file generation program.

When the file to be loaded contains a complete package for a given application, `LOAD_PACKAGE` rather than `LOAD` should be used. The syntax is the same. However, `LOAD_PACKAGE` does some additional bookkeeping such as recording that this package has now been loaded, that is required for the correct operation of the system.

18.3 The Standard Lisp Cross Reference Program

`CREF` is a Standard Lisp program for processing a set of Standard LISP function definitions to produce:

1. A "summary" showing:
    (a) A list of files processed;
    (b) A list of "entry points" (functions which are not called or are only called by themselves);
    (c) A list of undefined functions (functions called but not defined in this set of functions);
    (d) A list of variables that were used non-locally but not declared `GLOBAL` or `FLUID` before their use;
    (e) A list of variables that were declared `GLOBAL` but not used as `FLUID`s, i.e., bound in a function;
    (f) A list of `FLUID` variables that were not bound in a function so that one might consider declaring them `GLOBAL`s;
    (g) A list of all `GLOBAL` variables present;
    (h) A list of all `FLUID` variables present;

   (i) A list of all functions present.

2. A "global variable usage" table, showing for each non-local variable:
   (a) Functions in which it is used as a declared `FLUID` or `GLOBAL`;
   (b) Functions in which it is used but not declared;
   (c) Functions in which it is bound;
   (d) Functions in which it is changed by `SETQ`.
3. A "function usage" table showing for each function:
   (a) Where it is defined;
   (b) Functions which call this function;
   (c) Functions called by it;
   (d) Non-local variables used.

The program will also check that functions are called with the correct number of arguments, and print a diagnostic message otherwise.

The output is alphabetized on the first seven characters of each function name.

18.3.1 Restrictions

Algebraic procedures in REDUCE are treated as if they were symbolic, so that algebraic constructs will actually appear as calls to symbolic functions, such as `AEVAL`.

18.3.2 Usage

To invoke the cross reference program, the switch `CREF` is used. `on cref` causes the cref program to load and the cross-referencing process to begin. After all the required definitions are loaded, `off cref` will cause the cross-reference listing to be produced. For example, if you wish to cross-reference all functions in the file `tst.red`, and produce the cross-reference listing in the file `tst.crf`, the following sequence can be used:

```
out "tst.crf";
on cref;
in "tst.red"$
```

```
off cref;
shut "tst.crf";
```

To process more than one file, more `IN` statements may be added before the call of `off cref`, or the `IN` statement changed to include a list of files.

18.3.3 Options

Functions with the flag `NOLIST` will not be examined or output. Initially, all Standard Lisp functions are so flagged. (In fact, they are kept on a list `NOLIST!*`, so if you wish to see references to *all* functions, then `CREF` should be first loaded with the command `load cref`, and this variable then set to `NIL`).

It should also be remembered that any macros with the property list flag `EXPAND`, or, if the switch `FORCE` is on, without the property list flag `NOEXPAND`, will be expanded before the definition is seen by the cross-reference program, so this flag can also be used to select those macros you require expanded and those you do not.

18.4 Prettyprinting Reduce Expressions

REDUCE includes a module for printing REDUCE syntax in a standard format. This module is activated by the switch `PRET`, which is normally off.

Since the system converts algebraic input into an equivalent symbolic form, the printing program tries to interpret this as an algebraic expression before printing it. In most cases, this can be done successfully. However, there will be occasional instances where results are printed in symbolic mode form that bears little resemblance to the original input, even though it is formally equivalent.

If you want to prettyprint a whole file, say `off output,msg;` and (hopefully) only clean output will result. Unlike `DEFN`, input is also evaluated with `PRET` on.

18.5 Prettyprinting Standard Lisp S-Expressions

REDUCE includes a module for printing S-expressions in a standard format. The Standard Lisp function for this purpose is `PRETTYPRINT` which takes a Lisp expression and prints the formatted equivalent.

Users can also have their REDUCE input printed in this form by use of the switch `DEFN`. This is in fact a convenient way to convert REDUCE (or Rlisp) syntax into Lisp. `off msg;` will prevent warning messages from being printed.

NOTE: When `DEFN` is on, input is not evaluated.

Chapter 19

Maintaining REDUCE

REDUCE continues to evolve both in terms of the number of facilities available, and the power of the individual facilities. Corrections are made as bugs are discovered, and awkward features simplified. In order to provide users with easy access to such enhancements, a *REDUCE network library* has been established from which material can be extracted by anyone with electronic mail access to the Internet computer network.

In addition to miscellaneous documents, source and utility files, the library includes a bibliography of papers referencing REDUCE which contains over 800 entries. Instructions on using this library are sent to all registered REDUCE users who provide a network address. If you would like a more complete list of the contents of the library, send to *reduce-netlib@rand.org* the single line message *send index* or *help*. The current REDUCE information package can be obtained from the network library by including on a separate line *send info-package* and a demonstration file by including the line *send demonstration*. If you prefer, hard copies of the information package and the bibliography are available from the REDUCE secretary at RAND, 1700 Main Street, P.O. Box 2138, Santa Monica, CA 90407-2138 (*reduce@rand.org*). Copies of the network library are also maintained at other addresses. At the time of writing, *reduce-netlib@can.nl* and *reduce-netlib@pi.cc.u-tokyo.ac.jp* may also be used instead of *reduce-netlib@rand.org*.

A World Wide Web REDUCE server with URL

```
http://www.rrz.uni-koeln.de/REDUCE/
```

is also supported. In addition to general information about REDUCE, this server has pointers to the network library, the demonstration versions, examples of REDUCE programming, a set of manuals, and the REDUCE online help system.

Finally, there is a REDUCE electronic forum accessible from the same networks. This enables REDUCE users to raise questions and discuss ideas concerning the use and development of REDUCE with other users. Additions and changes to the network library and new releases of REDUCE are also announced in this forum. Any user with appropriate electronic mail access is encouraged to register for membership in this forum. To do so, send a message requesting inclusion to
reduce-forum-request@rand.org.

Part II

Additional REDUCE Documentation

The documentation in this section was written using to a large part the LaTeX files provided by the authors, and distributed with REDUCE. There has been extensive editing and much rewriting, so the responsibility for this part of the manual rests with the editor, John Fitch. It is hoped that this version of the documentation contains sufficient information about the facilities available that a user may be able to progress. It deliberately avoids discussions of algorithms or advanced use; for these the package author's own documentation should be consulted. In general the package documentation will contain more examples and in some cases additional facilities such as tracing.

Chapter 20

ALGINT: Integration of square roots

James Davenport
School of Mathematical Sciences
University of Bath
Bath BA2 7AY
England
e–mail: jhd@maths.bath.ac.uk

The package supplies no new functions, but extends the INT operator for indefinite integration so it can handle a wider range of expressions involving square roots. When it is loaded the controlling switch ALGINT is turned on. If it is desired to revert to the standard integrator, then it may be turned off. The normal integrator can deal with some square roots but in an unsystematic fashion.

```
1: load_package algint;

2: int(sqrt(sqrt(a^2+x^2)+x)/x,x);


                                   2    2
sqrt(a)*atan((sqrt(a)*sqrt(sqrt(a  + x ) + x)

                        2    2
              *sqrt(a  + x )

                                      2    2
```

```
 - sqrt(a)*sqrt(sqrt(a  + x ) + x)*a

                         2     2
 - sqrt(a)*sqrt(sqrt(a  + x ) + x)*x)/(2
```

```
               2                  2    2
              *a )) + 2*sqrt(sqrt(a  + x ) + x)

                          2    2
 + sqrt(a)*log(sqrt(sqrt(a  + x ) + x) - sqrt(a))

                          2    2
 - sqrt(a)*log(sqrt(sqrt(a  + x ) + x) + sqrt(a))

3: off algint;

4: int(sqrt(sqrt(a^2+x^2)+x)/x,x);

                2    2
     sqrt(sqrt(a  + x ) + x)
int(-------------------------,x)
                x
```

There is also a switch TRA, which may be set on to provide detailed tracing of the algorithm used. This is not recommended for casual use.

Chapter 21

APPLYSYM: Infinitesimal symmetries of differential equations

Thomas Wolf
School of Mathematical Sciences, Queen Mary and Westfield College
University of London
London E1 4NS, England
e–mail: T.Wolf@maths.qmw.ac.uk

The investigation of infinitesimal symmetries of differential equations (DEs) with computer algebra programs attracted considerable attention over the last years. The package `APPLYSYM` concentrates on the implementation of applying symmetries for calculating similarity variables to perform a point transformation which lowers the order of an ODE or effectively reduces the number of explicitly occuring independent variables of a PDE(-system) and for generalising given special solutions of ODEs/PDEs with new constant parameters.

A prerequisite for applying symmetries is the solution of first order quasilinear PDEs. The corresponding program `QUASILINPDE` can as well be used without `APPLYSYM` for solving first order PDEs which are linear in their first order derivative and otherwise at most rationally non-linear. The following two PDEs are equations (2.40) and (3.12) taken from E. Kamke, "Loesungsmethoden und Loesungen von Differential- gleichungen, Partielle Differentialgleichungen erster Ordnung", B.G. Teubner, Stuttgart (1979).

```
------------------------ Equation 2.40 ------------------------

                                     2               3      4
The quasilinear PDE:  0 = df(z,x)*x*y  + 2*df(z,y)*y  - 2*x

      2          2  2
 + 4*x *y*z - 2*y *z .
The equivalent characteristic system:

              3    4      2        2  2
0=2*(df(z,y)*y  - x  + 2*x *y*z - y *z )

   2
0=y *(2*df(x,y)*y - x)

for the functions: x(y)  z(y)  .
The general solution of the PDE is given through

                4           2        2
        log(y)*x  - log(y)*x *y*z - y *z   sqrt(y)*x
0 = ff(----------------------------------,-----------)
                   4    2                      y
                  x  - x *y*z

with arbitrary function ff(..).

------------------------ Equation 3.12 ------------------------

The quasilinear PDE:  0 = df(w,x)*x + df(w,y)*a*x + df(w,y)*b*y

 + df(w,z)*c*x + df(w,z)*d*y + df(w,z)*f*z.
The equivalent characteristic system:

0=df(w,x)*x


0=df(z,x)*x - c*x - d*y - f*z


0=df(y,x)*x - a*x - b*y

for the functions: z(x)  y(x)  w(x)  .
The general solution of the PDE is given through

        a*x + b*y - y
```

```
0 = ff(---------------,( - a*d*x + b*c*x + b*f*z - b*z - c*f*x
         b      b
        x *b - x

                        2           f       f     f  2    f
         - d*f*y + d*y - f *z + f*z)/(x *b*f - x *b - x *f  + x *f)

      ,w)

with arbitrary function ff(..).
```

The program `DETRAFO` can be used to perform point transformations of ODEs/PDEs (and -systems).

For detailed explanations the user is referred to the paper *Programs for Applying Symmetries of PDEs* by Thomas Wolf, supplied as part of the Reduce documentation as `applysym.tex` and published in the Proceedings of ISSAC'95 - 7/95 Montreal, Canada, ACM Press (1995).

Chapter 22

ARNUM: An algebraic number package

Eberhard Schrüfer
Institute SCAI.Alg
German National Research Center for Information Technology (GMD)
Schloss Birlinghoven
D-53754 Sankt Augustin, Germany
e–mail: schruefer@gmd.de

Algebraic numbers are the solutions of an irreducible polynomial over some ground domain. The algebraic number i (imaginary unit), for example, would be defined by the polynomial $i^2 + 1$. The arithmetic of algebraic number s can be viewed as a polynomial arithmetic modulo the defining polynomial.

The `ARNUM` package provides a mechanism to define other algebraic numbers, and compute with them.

22.1 DEFPOLY

`DEFPOLY` takes as its argument the defining polynomial for an algebraic number, or a number of defining polynomials for different algebraic numbers, and arranges that arithmetic with the new symbol(s) is performed relative to these polynomials.

```
load_package arnum;

defpoly sqrt2**2-2;

1/(sqrt2+1);

SQRT2 - 1

(x**2+2*sqrt2*x+2)/(x+sqrt2);

X + SQRT2

on gcd;

(x**3+(sqrt2-2)*x**2-(2*sqrt2+3)*x-3*sqrt2)/(x**2-2);

  2
 X  - 2*X - 3
-------------
  X - SQRT2

off gcd;

sqrt(x**2-2*sqrt2*x*y+2*y**2);

ABS(X - SQRT2*Y)
```

The following example introduces both $\sqrt{2}$ and $5^{\frac{1}{3}}$:

```
defpoly sqrt2**2-2,cbrt5**3-5;

*** defining polynomial for primitive element:

  6       4        3        2
A1  - 6*A1  - 10*A1  + 12*A1  - 60*A1 + 17

sqrt2;

         5              4               3               2
48/1187*A1  + 45/1187*A1  - 320/1187*A1  - 780/1187*A1  +


735/1187*A1 - 1820/1187
```

```
sqrt2**2;

2
```

22.2 SPLIT_FIELD

The function `SPLIT_FIELD` calculates a primitive element of minimal degree for which a given polynomial splits into linear factors.

```
split_field(x**3-3*x+7);

*** Splitting field is generated by:

  6        4        2
A5  - 18*A5  + 81*A5  + 1215



         4         2
{1/126*A5  - 5/42*A5  - 1/2*A5 + 2/7,


           4         2
 - (1/63*A5  - 5/21*A5  + 4/7),


        4         2
1/126*A5  - 5/42*A5  + 1/2*A5 + 2/7}


for each j in ws product (x-j);

 3
X  - 3*X + 7
```

Chapter 23

ASSIST: Various Useful Utilities

Hubert Caprasse
Département d'Astronomie et d'Astrophysique
Institut de Physique, B–5, Sart Tilman
B–4000 LIEGE 1, Belgium
e–mail: caprasse@vm1.ulg.ac.be

The ASSIST package provides a number of general purpose functions which adapt REDUCE to various calculational strategies. All the examples in this section require the ASSIST package to be loaded.

23.1 Control of Switches

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

SWITCHES displays the current status of the most often used switches when manipulating rational functions; EXP, DIV, MCD, GCD, ALLFAC, INTSTR, RAT, RATIONAL, FACTOR. The switch DISTRIBUTE which controls the handling of distributed polynomials is included as well (see section 23.8).

SWITCHORG resets (almost) *all* switches in the status they have when **entering** into REDUCE. (See also RESET, chapter 68). The new switch DISTRIBUTE facilitates changing polynomials to a distributed form.

23.2 Manipulation of the List Structure

Functions for list manipulation are provided and are generalised to deal with the new structure BAG.

i. Generation of a list of length n with its elements initialised to 0 and also to append to a list l sufficient zeros to make it of length n:

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

ii. Generation of a list of sublists of length n containing p elements equal to 0 and $n - p$ elements equal to 1.

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

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

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

There are four functions to delete elements from lists. The DELETE function deletes the first occurrence of its first argument from the second, while REMOVE removes a numbered element. DELETE_ALL eliminates from a list *all* elements equal to its first argument. DELPAIR acts on list of pairs and eliminates from it the *first* pair whose first element is equal to its first argument:

```
DELETE(x,{a,b,x,f,x}); ==> {a,b,f,x}
REMOVE({a,b,x,f,x},3); ==> {a,b,f,x}
DELETE_ALL(x,{a,b,x,f,x}); ==> {a,b,f}
DELPAIR(a,{{a,1},{b,2},{c,3}}; ==> {{b,2},{c,3}}
```

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

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

v. The function INSERT inserts a given object into a list at the wanted position. The functions INSERT_KEEP_ORDER and MERGE_LIST keep a given ordering when inserting one element inside a list or when merging two lists. Both have 3 arguments. The last one is the name of a binary boolean ordering function:

```
ll:={1,2,3}$
INSERT(x,ll,3); ==> {1,2,x,3}
INSERT_KEEP_ORDER(5,ll,lessp); ==> {1,2,3,5}
MERGE_LIST(ll,ll,lessp); ==> {1,1,2,2,3,3}
```

vi. Algebraic lists can be read from right to left or left to right. They *look* symmetrical. It is sometimes convenient to have functions which reflect this. So, as well as `FIRST` and `REST` this package provides the functions `LAST` and `BELAST`. `LAST` gives the last element of the list while `BELAST` gives the list *without* its last element.
Various additional functions are provided. They are: `CONS`, `(.)`, `POSITION`, `DEPTH`, `PAIR`, `APPENDN`, `REPFIRST`, `REPLAST` The token "dot" needs a special comment. It corresponds to several different operations.

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

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

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

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

3. If one applies it on 4–dimensional vectors, it acts as in the HEPHYS package (chapter 17.1

`POSITION` returns the position of the first occurrence of x in a list or a message if x is not present in it. `DEPTH` returns an *integer* equal to the number of levels where a list is found if and only if this number is the *same* for each element of the list otherwise it returns a message telling the user that list is of *unequal depth.* `PAIR` has two arguments which must be lists. It returns a list whose elements are *lists of two elements.* The n^{th} sublist contains the n^{th} element of the first list and the n^{th} element of the second list. These types of lists are called *association lists* or ALISTS in the following. `APPENDN` has *any* number of lists as arguments, and appends them all. `REPFIRST` has two arguments. The first one is any object, the second one is a list. It replaces the first element of the list by the object. `REPREST` has also two arguments. It replaces the rest of the list by its first argument and returns the new list without destroying the original list.

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

```
0 . ll;                       ==> {0,{a,b}}
DEPTH ll;                     ==> 2
PAIR(lll,lll);                ==> {{a,a},{b,b}}
REPFIRST{new,ll);             ==> {new}
ll3:=APPENDN(lll,lll,lll);    ==> {a,b,a,b,a,b}
POSITION(b,ll3);              ==> 2
REPREST(new,ll3);             ==> {a,new}
```

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

```
lp:={{a,1},{b,2},{c,3}}$
ASFIRST(a,lp);                ==> {a,1}
ASLAST(1,lp);                 ==> {a,1}
ASREST({1},lp);               ==> {a,1}
RESTASLIST({a,b},lp);         ==> {{1},{2}}
lpp:=APPEND(lp,lp)$
ASFLIST(a,lpp);               ==> {{a,1},{a,1}}
ASSLIST(1,lpp);               ==> {{a,1},{a,1}}
```

23.3 The Bag Structure and its Associated Functions

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

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

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

Definition:

A bag is an object endowed with the following properties:

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

Available Functions:

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

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

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

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

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

ii. The boolean function BAGP detects the bag property.

```
aa:=bag(x,y,z)$
if BAGP aa then "ok";      ==> ok
```

iii. Most functions defined above for lists do also work for bags. Moreover functions subsequently defined for SETS (see section 23.4) also work. However, because of the conservation of the envelope, they act somewhat differently.

```
PUTBAG op;               ==> T
aa:=op(x,y,z)$
FIRST op(x,y,z);         ==> op(x)
REST op(x,y,z);          ==> op(y,z)
BELAST op(x,y,z);        ==> op(x,y)
APPEND(aa,aa);           ==> op(x,y,z,x,y,z)
LENGTH aa;               ==> 3
DEPTH aa;                ==> 1
```

When "appending" two bags with *different* envelopes, the resulting bag gets the name of the one bound to the first parameter of APPEND. The function LENGTH gives the actual number of variables on which the operator (or the function) depends. The NAME of the ENVELOPE is kept by the functions FIRST, SECOND, LAST and BELAST.

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

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

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

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

23.4 Sets and their Manipulation Functions

The ASSIST package makes the Standard LISP set functions available in algebraic mode and also *generalises* them so that they can be applied on bag–like objects as well.

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

```
MKSET({1,a,a1});      ==> {1,a}
MKSET bag(1,a,a1);    ==> bag(1,a)
```

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

ii. The standard functions are UNION, INTERSECT, DIFFSET and SYMDIFF. They have two arguments which must be sets; otherwise an error message is issued.

23.5 General Purpose Utility Functions

i. The functions MKIDNEW, DELLASTDIGIT, DETIDNUM, LIST_TO_IDS handle identifiers. MKIDNEW is a variant of MKID.

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

```
MKIDNEW(); ==> g0001
MKIDNEW(a); ==> ag0002
```

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

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

DETIDNUM, determines the trailing integer from an identifier. It is convenient when one wants to make a do loop starting from a set of indices $a_1, \ldots, a_n$.

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

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

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

The function `ODDP` detects odd integers.

The function `FOLLOWLINE` is convenient when using the function `PRIN2` for controlling layout.

```
        <<prin2 2; prin2 5>>$
25
        <<prin2 2; followline(3); prin2 5>>$
2
   5
```

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

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

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

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

The output is the array dimension.

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

`PERMUTATIONS(n)` gives the list of permutations on n objects, each permutation being represented as a list. `CYCLICPERMLIST` gives the list of *cyclic* permutations. For both functions, the argument may also be a `bag`.

```
        PERMUTATIONS {1,2} ==> {{1,2},{2,1}}
        CYCLICPERMLIST {1,2,3} ==>
                {{1,2,3},{2,3,1},{3,1,2}}
```

`COMBINATIONS` gives a list of combinations on n objects taken p at a time. The first argument is a list (or a bag) and the second is the integer p.

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

`REMSYM` is a command that erases the REDUCE commands `symmetric` or `antisymmetric`.

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

```
ll:={a,b,c}$
SYMMETRIZE(ll,op,cyclicpermlist); ==>
        OP(A,B,C) + OP(B,C,A) + OP(C,A,B)
SYMMETRIZE(list ll,op,cyclicpermlist); ==>
        OP({A,B,C}) + OP({B,C,A}) + OP({C,A,B})
```

Notice that taking for the first argument a list of lists gives rise to an expression where each kernel has a *list as argument*. Another peculiarity of this function is that, unless a pattern matching is made on the operator `OP`, it needs to be reevaluated. Here is an illustration:

```
op(a,b,c):=a*b*c$
SYMMETRIZE(ll,op,cyclicpermlist); ==>
         OP(A,B,C) + OP(B,C,A) + OP(C,A,B)
for all x let op(x,a,b)=sin(x*a*b);
SYMMETRIZE(ll,op,cyclicpermlist); ==>
          OP(B,C,A) + SIN(A*B*C) + OP(A,B,C)
```

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

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

`SORTLIST` is a generalisation of the above function. It sorts the list according to any well defined ordering. Its first argument is the list and its second argument is the ordering function. The content of the list is not necessary numbers but must be such that the ordering function has a meaning.

```
l:={1,3,4,0}$  SORTNUMLIST l;      ==> {0,1,3,4}
ll:={1,a,tt,z}$ SORTLIST(ll,ordp); ==> {a,z,tt,1}
```

Note: using these functions for kernels or bags may be dangerous since they are destructive. If it is needed, it is recommended first to apply `KERNLIST` on them.

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

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

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

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

`DEPATOM` has an **atom** as argument. It returns its argument if it is a number or if no dependency has previously been declared. Otherwise, it returns the list of variables on which in depends as declared in various `DEPEND` declarations.

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

The functions `EXPLICIT` and `IMPLICIT` make explicit or implicit the dependencies.

```
depend a,x; depend x,y,z;
EXPLICIT a;       ==> a(x(y,z))
IMPLICIT ws;      ==> a
```

These are useful when one does not know the names of the variables and (or) the nature of the dependencies.

`KORDERLIST` is a zero argument function which display the actual ordering.

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

iv. A function SIMPLIFY which takes an arbitrary expression is available which *forces* down-to-the-bottom simplification of an expression. It is useful with SYMMETRIZE. It has also proved useful to simplify some output expressions of the package EXCALC (chapter 39).

```
l:=op(x,y,z)$
op(x,y,z):=x*y*z$
SYMMETRIZE(l,op,cyclicpermlist); ==>
                        op(x,y,z)+op(y,z,x)+op(z,x,y)
SIMPLIFY ws;       ==> op(y,z,x)+op(z,x,y)+x*y*z
```

v. Filtering functions for lists.

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

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

```
l:={1,a,b,"st"}$
EXTRACTLIST(l,fixp);    ==> {1}
EXTRACTLIST(l,stringp); ==> {st}
```

23.6 Properties and Flags

It may be useful to provide analogous functions in algebraic mode to the properties and flags of LISP. Just using the symbolic mode functions to alter property lists of objects may easily destroy the integrity of the system. The functions which are here described **do ignore** the property list and flags already defined by the system itself. They generate and track the *additional properties and flags* that the user issues using them. They offer the possibility of working on property lists in an algebraic context.

i. Flags To a given identifier, one may associates another one linked to it "in the background". The three functions PUTFLAG, DISPLAYFLAG and CLEARFLAG handle them.

PUTFLAG has 3 arguments. The first is the identifier or a list of identifiers, the second is the name of the flag, the third is T (true) or 0 (zero). When the third argument is T, it creates the flag, when it is 0 it destroys it.

```
PUTFLAG(z1,flag_name,t);       ==> flag_name
PUTFLAG({z1,z2},flag1_name,t); ==> t
PUTFLAG(z2,flag1_name,0);      ==>
```

DISPLAYFLAG allows to extract flags. Continuing the example:

```
DISPLAYFLAG z1;    ==> {flag_name,flag1_name}
DISPLAYFLAG z2;    ==> {}
```

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

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

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

In general, one enter

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

If the last argument is 0 then the property is removed. To display a specific property, one uses DISPLAYPROP which takes two arguments. The first is the name of the identifier, the second is the indicator of the property.

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

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

23.7 Control Functions

The ASSIST package also provides additional functions which improve the user control of the environment.

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

```
ALATOMP x;    x is anything.
ALKERNP x;    x is anything.
DEPVARP(x,v); x is anything.
              (v is an atom or a kernel)
```

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

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

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

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

`NORDP` is essentially equivalent to `notORDP` when inside a conditional statement. Otherwise, it can be used while `notORDP` cannot.

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

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

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

```
SHOW all;         ==> scalars are: NIL
                      arrays are: NIL
                      lists are: NIL
                      matrices are: NIL
                      vectors are: NIL
                      forms are: NIL
```

It is a convenient way to remember the various options. Starting from a fresh environment

```
a:=b:=1$
SHOW scalars;     ==>  scalars are: (A B)
SUPPRESS scalars; ==> t
SHOW scalars;     ==>  scalars are: NIL
```

iii. The CLEAR function of the system does not do a complete cleaning of OPERATORS and FUNCTIONS. The following two functions do a more complete cleaning and, also automatically takes into account the *user* flag and properties that the functions PUTFLAG and PUTPROP may have introduced.

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

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

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

23.8 Handling of Polynomials

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

i. Two functions ALG_TO_SYMB and SYMB_TO_ALG allow the changing of an expression which is in the algebraic standard quotient form into a prefix lisp form and vice-versa. This is made in such a way that the symbol list which appears in the algebraic mode disappear in the symbolic form (there it becomes a parenthesis "()") and it is reintroduced in the translation from a symbolic prefix lisp expression to an algebraic one. The following example shows how the well-known lisp function FLATTENS can be trivially transportd into algebraic mode:

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

 symbolic procedure flattens1 x;
```

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

gives, for instance,

```
ll:={a,{b,{c},d,e},{{{z}}}}$
ECRASE ll; ==> {A, B, C, D, E, Z}
```

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

```
pol:=x+y+z$
LEADTERM pol; ==> x
korder y,x,z;
LEADTERM pol; ==> y
REDEXPR pol; ==> x + z
```

By default, the representation of multivariate polynomials is recursive. With such a representation, the function `LEADTERM` does not necessarily extract a true monom. It extracts a monom in the leading indeterminate multiplied by a polynomial in the other indeterminates. However, very often one needs to handle true monoms separately. In that case, one needs a polynomial in *distributive* form. Such a form is provided by the package GROEBNER (chapter 45). The facility there may be too involved and the need to load an additional package can be a problem. So, a new switch is created to handle *distributed* polynomials. It is called `DISTRIBUTE` and a new function `DISTRIBUTE` puts a polynomial in distributive form. With the switch **on**, `LEADTERM` gives **true** monoms.

`MONOM` transforms a polynomial into a list of monoms. It works whatever the setting of the switch `DISTRIBUTE`.

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

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

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

23.9 Handling of Transcendental Functions

The functions TRIGREDUCE and TRIGEXPAND and the equivalent ones for hyperbolic functions HYPREDUCE and HYPEXPAND make the transformations to multiple arguments and from multiple arguments to elementary arguments.

```
aa:=sin(x+y)$
TRIGEXPAND aa; ==> SIN(X)*COS(Y) + SIN(Y)*COS(X)
TRIGREDUCE ws; ==> SIN(Y + X)
```

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

```
aa:=(sin(x)^2+cos(x)^2)^3$
TRIGREDUCE aa; ==> 1
bb:=1+sin(x)^3$
TRIGREDUCE bb; ==>
        - SIN(3*X) + 3*SIN(X) + 4
       ---------------------------
                   4

 TRIGEXPAND ws; ==>
        3                  2
  SIN(X)  - 3*SIN(X)*COS(X)  + 3*SIN(X) + 4
  -------------------------------------------
                     4
```

See also the TRIGSIMP package (chapter 85).

23.10 Coercion from lists to arrays and converse

Sometimes when a list is very long and especially if frequent access to its elements are needed it is advantageous (temporarily) to transform it into an array. LIST_TO_ARRAY has three arguments. The first is the list. The second is an integer which indicates the array dimension required. The third is the name of an identifier which will play the role of the array name generated by it. If the chosen dimension is not compatible with the list depth and structure an error message is issued. ARRAY_TO_LIST does the opposite coercion. It takes the array name as its sole argument.

23.11 Handling of n–dimensional Vectors

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

```
l:={1,2,3}$
ll:=list(a,b,c)$
SUMVECT(l,ll);    ==> {A + 1,B + 2,C + 3}
MINVECT(l,ll);    ==> { - A + 1, - B + 2, - C + 3}
SCALVECT(l,ll);   ==> A + 2*B + 3*C
CROSSVECT(l,ll);  ==> { - 3*B + 2*C,3*A - C, - 2*A + B}
MPVECT(l,ll,l);   ==> 0
```

23.12 Handling of Grassmann Operators

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

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

`GRASSP` is a boolean function which detects Grassmann kernels.

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

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

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

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

```
PUTGRASS eta;
if GRASSP eta(1) then "Grassmann kernel"; ==>
                                              Grassmann kernel
aa:=eta(1)*eta(2)-eta(2)*eta(1);       ==>
                      AA :=  - ETA(2)*ETA(1) + ETA(1)*ETA(2)
GRASSPARITY eta(1);                    ==> 1
GRASSPARITY (eta(1)*eta(2));           ==> 0
GHOSTFACTOR(eta(1),eta(2));            ==> -1
grasskernel:=
  {eta(~x)*eta(~y) => -eta y * eta x when nordp(x,y),
  (~x)*(~x) => 0 when grassp x}$
exp:=eta(1)^2$
exp where grasskernel;                 ==> 0
aa where grasskernel;                  ==>  - 2*ETA(2)*ETA(1)
```

23.13 Handling of Matrices

There are additional facilities for matrices.

i. Often one needs to construct some `UNIT` matrix of a given dimension. This construction is performed by the function `UNITMAT`. It is a nary function. The command is

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

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

`MKIDM` is a generalisation of `MKID`. It allows the indexing of matrix names. If `u` and `u1` are two matrices, one can go from one to the other:

```
matrix u(2,2);$  unitmat u1(2)$
u1; ==>
        [1  0]
        [    ]
        [0  1]

mkidm(u,1); ==>
        [1  0]
        [    ]
        [0  1]
```

Note: MKIDM(V,1) will fail even if the matrix V1 exists, unless V is also a matrix.

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

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

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

`COERCEMAT` transforms the matrix first argument into a list of lists.

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

When `id` is `list` the matrix is transformed into a list of lists. Otherwise it transforms it into a bag of bags whose envelope is equal to `id`.

`BAGLMAT` does the inverse. The **first** argument is the bag-like or list-like object while the second argument is the matrix identifier.

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

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

ii. The functions `SUBMAT`, `MATEXTR`, `MATEXTC` take parts of a given matrix. `SUBMAT` has three arguments.

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

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

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

```
MATEXTR(U,VN,nr)
MATEXTC(U,VN,nc)
```

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

iii. Functions which manipulate matrices: `MATSUBR`, `MATSUBC`, `HCONCMAT`, `VCONCMAT`, `TPMAT`, `HERMAT`.

`MATSUBR` and `MATSUBC` substitute rows and columns. They have three arguments.

```
MATSUBR(U,bgl,nr)
MATSUBC(U,bgl,nc)
```

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

`HCONCMAT` and `VCONCMAT` concatenate two matrices.

```
HCONCMAT(U,V)
VCONCMAT(U,V)
```

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

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

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

`HERMAT` takes the hermitian conjugate of a matrix

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

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

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

Chapter 24

ATENSOR: Package for Tensor Simplification

V. A. Ilyin and A. P. Kryukov

Tensors are classical examples for Objects often used in mathematics and physics. Indexed objects can have very complicated and intricated properties. For example the Riemann tensor has symmetry properties with respect to permutation of indices. Moreover it satisfies the cyclic identity. There are a number of linear identities with many terms in the case of Riemann-Cartan geometry with torsion. From the user's point of view, there are three groups of tensor properties:

- **S** - symmetry with respect to index permutation;
- **I** - linear identities;
- **D** - invariance with respect to renamings of dummy indices;

The problem under investigation can be formulated as whether two tensor expressions are equal or not by taking into account S-I-D properties.

24.1 Basic tensors and tensor expressions

Under basic tensors we understand the object with finite number of indices which can have such properties as *symmetry* and *multiterm linear identities*

(including the *symmetry relations*).
Under tensor expression we understand any expression which can be obtained from basic tensors by summation with integer coefficients and multiplication (commutative) of basic tensors.
It is assumed that all terms in the tensor expression have the same number of indices. Some pairs of them are marked as dummy ones. The set of non-dummy names have to be the same for each term in the tensor expression. The names of dummies can be arbitrary.

24.2 Operators for tensors

Use `TENSOR` to declare tensors and `TCLEAR` to remove them. The command `TSYM` defines symmetry relations of basic tensors and `KBASIS` determines the **K**-Basis, which is the general name for a "triangle" set of linear independent vectors for a basic tensor considered as a separate tensor expression. It is possible to build the sum, the difference and the multiplication for tensors. It is assumed that indices with identical names means the summation over their values.

Example:

```
1: load atensor;

2: tensor s2,a3;

3: tsym s2(i,j) - s2(j,i),       % Symmetric
3:      a3(i,j,k) + a3(j,i,k),   % Antisymm.
3:      a3(i,j,k) - a3(j,k,i);

4: kbasis s2,a3;

        s2(j,i) + (-1)*s2(i,j)
        1
        a3(k,i,j) + a3(j,i,k)
        a3(k,j,i) + (-1)*a3(j,i,k)
        a3(i,k,j) + (-1)*a3(j,i,k)
        a3(i,j,k) + a3(j,i,k)
        a3(j,k,i) + a3(j,i,k)
        5
```

24.3 Switches

There are two switches defined. The switch `DUMMYPRI` prints dummy indices with internal names and numbers. It's default value is `OFF`. The other switch called `SHORTEST` prints tensor expressions in shortest form that was produced during evaluation. The default value is `OFF`.

For further information refer to the documentation which comes with this package.

Chapter 25

AVECTOR: A vector algebra and calculus package

David Harper
Astronomy Unit, Queen Mary and Westfield College
University of London
Mile End Road
London E1 4NS, England
e–mail: adh@star.qmw.ac.uk

This package provides REDUCE with the ability to perform vector algebra using the same notation as scalar algebra. The basic algebraic operations are supported, as are differentiation and integration of vectors with respect to scalar variables, cross product and dot product, component manipulation and application of scalar functions (*e.g.* cosine) to a vector to yield a vector result.

25.1 Vector declaration and initialisation

To declare a list of names to be vectors use the VEC command:

```
VEC A,B,C;
```

declares the variables `A`, `B` and `C` to be vectors. If they have already been assigned (scalar) values, these will be lost.

When a vector is declared using the `VEC` command, it does not have an initial value.

If a vector value is assigned to a scalar variable, then that variable will automatically be declared as a vector and the user will be notified that this has happened.

A vector may be initialised using the `AVEC` function which takes three scalar arguments and returns a vector made up from those scalars. For example

```
A := AVEC(A1, A2, A3);
```

sets the components of the vector `A` to `A1`, `A2` and `A3`.

25.2 Vector algebra

(In the examples which follow, `V`, `V1`, `V2` *etc* are assumed to be vectors while `S`, `S1`, `S2` etc are scalars.)

The scalar algebra operators +,-,* and / may be used with vector operands according to the rules of vector algebra. Thus multiplication and division of a vector by a scalar are both allowed, but it is an error to multiply or divide one vector by another.

`V := V1 + V2 - V3;`	Addition and subtraction
`V := S1*3*V1;`	Scalar multiplication
`V := V1/S;`	Scalar division
`V := -V1;`	Negation

Vector multiplication is carried out using the infix operators `DOT` and `CROSS`. These are defined to have higher precedence than scalar multiplication and division.

`V := V1 CROSS V2;`	Cross product
`S := V1 DOT V2;`	Dot product
`V := V1 CROSS V2 + V3;`	
`V := (V1 CROSS V2) + V3;`	

The last two expressions are equivalent due to the precedence of the `CROSS` operator.

The modulus of a vector may be calculated using the `VMOD` operator.

```
S := VMOD V;
```

A unit vector may be generated from any vector using the VMOD operator.

```
V1 := V/(VMOD V);
```

Components may be extracted from any vector using index notation in the same way as an array.

`V := AVEC(AX, AY, AZ);`	
`V(0);`	yields AX
`V(1);`	yields AY
`V(2);`	yields AZ

It is also possible to set values of individual components. Following from above:

```
V(1) := B;
```

The vector V now has components AX, B, AZ.

Vectors may be used as arguments in the differentiation and integration routines in place of the dependent expression.

`V := AVEC(X**2, SIN(X), Y);`	
`DF(V,X);`	yields (2*X, COS(X), 0)
`INT(V,X);`	yields (X**3/3, -COS(X), Y*X)

Vectors may be given as arguments to monomial functions such as SIN, LOG and TAN. The result is a vector obtained by applying the function component-wise to the argument vector.

`V := AVEC(A1, A2, A3);`	
`SIN(V);`	yields (SIN(A1), SIN(A2), SIN(A3))

25.3 Vector calculus

The vector calculus operators div, grad and curl are recognised. The Laplacian operator is also available and may be applied to scalar and vector arguments.

`V := GRAD S;`	Gradient of a scalar field
`S := DIV V;`	Divergence of a vector field
`V := CURL V1;`	Curl of a vector field
`S := DELSQ S1;`	Laplacian of a scalar field
`V := DELSQ V1;`	Laplacian of a vector field

These operators may be used in any orthogonal curvilinear coordinate system. The user may alter the names of the coordinates and the values of the scale factors. Initially the coordinates are X, Y and Z and the scale factors are all unity.

There are two special vectors : COORDS contains the names of the coordinates in the current system and HFACTORS contains the values of the scale factors.

The coordinate names may be changed using the COORDINATES operator.

```
COORDINATES R,THETA,PHI;
```

This command changes the coordinate names to R, THETA and PHI.

The scale factors may be altered using the SCALEFACTORS operator.

```
SCALEFACTORS(1,R,R*SIN(THETA));
```

This command changes the scale factors to 1, R and R SIN(THETA).

Note that the arguments of SCALEFACTORS must be enclosed in parentheses. This is not necessary with COORDINATES.

When vector differential operators are applied to an expression, the current set of coordinates are used as the independent variables and the scale factors are employed in the calculation.

Several coordinate systems are pre-defined and may be invoked by name. To see a list of valid names enter

```
SYMBOLIC !*CSYSTEMS;
```

and REDUCE will respond with something like

```
(CARTESIAN SPHERICAL CYLINDRICAL)
```

To choose a coordinate system by name, use the command GETCSYSTEM.

To choose the Cartesian coordinate system :

```
GETCSYSTEM 'CARTESIAN;
```

Note the quote which prefixes the name of the coordinate system. This is required because `GETCSYSTEM` (and its complement `PUTCSYSTEM`) is a `SYMBOLIC` procedure which requires a literal argument.

REDUCE responds by typing a list of the coordinate names in that coordinate system. The example above would produce the response

```
(X Y Z)
```

whilst

```
GETCSYSTEM 'SPHERICAL;
```

would produce

```
(R THETA PHI)
```

Note that any attempt to invoke a coordinate system is subject to the same restrictions as the implied calls to `COORDINATES` and `SCALEFACTORS`. In particular, `GETCSYSTEM` fails if any of the coordinate names has been assigned a value and the previous coordinate system remains in effect.

A user-defined coordinate system can be assigned a name using the command `PUTCSYSTEM`. It may then be re-invoked at a later stage using `GETCSYSTEM`.

Example 1

We define a general coordinate system with coordinate names `X`,`Y`,`Z` and scale factors `H1`,`H2`,`H3` :

```
COORDINATES X,Y,Z;
SCALEFACTORS(H1,H2,H3);
PUTCSYSTEM 'GENERAL;
```

This system may later be invoked by entering

```
GETCSYSTEM 'GENERAL;
```

25.4 Volume and Line Integration

Several functions are provided to perform volume and line integrals. These operate in any orthogonal curvilinear coordinate system and make use of the scale factors described in the previous section.

Definite integrals of scalar and vector expressions may be calculated using the DEFINT function[1].

Example 2

To calculate the definite integral of $\sin(x)^2$ between 0 and 2π we enter

```
DEFINT(SIN(X)**2,X,0,2*PI);
```

This function is a simple extension of the INT function taking two extra arguments, the lower and upper bounds of integration respectively.

Definite volume integrals may be calculated using the VOLINTEGRAL function whose syntax is as follows :

VOLINTEGRAL(integrand, vector lower-bound, vector upper-bound);

Example 3

In spherical polar coordinates we may calculate the volume of a sphere by integrating unity over the range r=0 to RR, θ=0 to PI, ϕ=0 to 2*π as follows :

`VLB := AVEC(0,0,0);`	Lower bound
`VUB := AVEC(RR,PI,2*PI);`	Upper bound in r, θ, ϕ respectively
`VOLINTORDER := (0,1,2);`	The order of integration
`VOLINTEGRAL(1,VLB,VUB);`	

Note the use of the special vector VOLINTORDER which controls the order in which the integrations are carried out. This vector should be set to contain the number 0, 1 and 2 in the required order. The first component of VOLINTORDER contains the index of the first integration variable, the second component is the index of the second integration variable and the third component is the index of the third integration variable.

Example 4

Suppose we wish to calculate the volume of a right circular cone. This is

[1]Not to be confused with the DEFINT package described in chapter 34

equivalent to integrating unity over a conical region with the bounds:

z = 0 to H	(H = the height of the cone)
r = 0 to pZ	(p = ratio of base diameter to height)
phi = 0 to 2*PI	

We evaluate the volume by integrating a series of infinitesimally thin circular disks of constant z-value. The integration is thus performed in the order : d(ϕ) from 0 to 2π, dr from 0 to p*Z, dz from 0 to H. The order of the indices is thus 2, 0, 1.

```
VOLINTORDER := AVEC(2,0,1);
VLB := AVEC(0,0,0);
VUB := AVEC(P*Z,H,2*PI);
VOLINTEGRAL(1,VLB,VUB);
```

Line integrals may be calculated using the `LINEINT` and `DEFLINEINT` functions. Their general syntax is

`LINEINT`(`vector-fnct`, `vector-curve`, `variable`);

`DEFLINENINT`(`vector-fnct`, `vector-curve`, `variable`,
`lower-bnd`, `upper-bnd`);

where

`vector-fnct` is any vector-valued expression;

`vector-curve` is a vector expression which describes the path of integration in terms of the independent variable;

`variable` is the independent variable;

`lower-bnd`

`upper-bnd` are the bounds of integration in terms of the independent variable.

Example 5

In spherical polar coordinates, we may integrate round a line of constant theta ('latitude') to find the length of such a line. The vector function is thus the tangent to the 'line of latitude', (0,0,1) and the path is `(0,LAT,PHI)` where `PHI` is the independent variable. We show how to obtain the definite integral *i.e.* from $\phi = 0$ to 2π :

```
DEFLINEINT(AVEC(0,0,1),AVEC(0,LAT,PHI),PHI,0,2*PI);
```

Chapter 26

BOOLEAN: A package for boolean algebra

Herbert Melenk
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany
e–mail: melenk@zib.de

The package **Boolean** supports the computation with boolean expressions in the propositional calculus. The data objects are composed from algebraic expressions ("atomic parts", "leafs") connected by the infix boolean operators **and**, **or**, **implies**, **equiv**, and the unary prefix operator **not**. **Boolean** allows simplification of expressions built from these operators, and to test properties like equivalence, subset property etc. Also the reduction of a boolean expression by a partial evaluation and combination of its atomic parts is supported.

26.1 Entering boolean expressions

In order to distinguish boolean data expressions from boolean expressions in the REDUCE programming language (*e.g.* in an **if** statement), each expression must be tagged explicitly by an operator **boolean**. Otherwise the boolean operators are not accepted in the REDUCE algebraic mode input. The first argument of **boolean** can be any boolean expression, which may contain references to other boolean values.

```
load_package boolean;
boolean (a and b or c);
q := boolean(a and b implies c);
boolean(q or not c);
```

Brackets are used to override the operator precedence as usual. The leafs or atoms of a boolean expression are those parts which do not contain a leading boolean operator. These are considered as constants during the boolean evaluation. There are two pre-defined values:

- **true**, **t** or **1**
- **false**, **nil** or **0**

These represent the boolean constants. In a result form they are used only as **1** and **0**.

By default, a **boolean** expression is converted to a disjunctive normal form.

On output, the operators **and** and **or** are represented as /\ and \/, respectively.

```
boolean(true and false);    ->   0
boolean(a or not(b and c)); -> boolean(not(b) \/ not(c) \/ a)
boolean(a equiv not c);     -> boolean(not(a)/\c \/ a/\not(c))
```

26.2 Normal forms

The **disjunctive** normal form is used by default. Alternatively a **conjunctive** normal form can be selected as simplification target, which is a form with leading operator **and**. To produce that form add the keyword **and** as an additional argument to a call of **boolean**.

```
boolean (a or b implies c);
                    ->
     boolean(not(a)/\not(b) \/ c)

boolean (a or b implies c, and);
                    ->
     boolean((not(a) \/ c)/\(not(b) \/ c))
```

Usually the result is a fully reduced disjunctive or conjuntive normal form, where all redundant elements have been eliminated following the rules

$a \wedge b \vee \neg a \wedge b \longleftrightarrow b$

$a \vee b \wedge \neg a \vee b \longleftrightarrow b$

Internally the full normal forms are computed as intermediate result; in these forms each term contains all leaf expressions, each one exactly once. This unreduced form is returned when the additional keyword **full** is set:

```
boolean (a or b implies c, full);
                    ->
boolean(a/\b/\c \/ a/\not(b)/\c \/ not(a)/\b/\c \/ not(a)/\not(b)/\c

         \/ not(a)/\not(b)/\not(c))
```

The keywords **full** and **and** may be combined.

26.3 Evaluation of a boolean expression

If the leafs of the boolean expression are algebraic expressions which may evaluate to logical values because the environment has changed (*e.g.* variables have been bound), one can re–investigate the expression using the operator `TESTBOOL` with the boolean expression as argument. This operator tries to evaluate all leaf expressions in REDUCE boolean style. As many terms as possible are replaced by their boolean values; the others remain unchanged. The resulting expression is contracted to a minimal form. The result **1** (= true) or **0** (=false) signals that the complete expression could be evaluated.

In the following example the leafs are built as numeric greater test. For using $>$ in the expressions the greater sign must be declared operator first. The error messages are meaningless.

```
operator >;
fm:=boolean(x>v or not (u>v));
        ->
    fm := boolean(not(u>v) \/ x>v)

v:=10$ testbool fm;

   ***** u - 10 invalid as number
   ***** x - 10 invalid as number

        ->
   boolean(not(u>10) \/ x>10)

x:=3$ testbool fm;

   ***** u - 10 invalid as number

        ->
```

```
   boolean(not(u>10))

x:=17$ testbool fm;

   ***** u - 10 invalid as number

        ->
    1
```

Chapter 27

CALI: Computational Commutative Algebra

Hans-Gert Gräbe
Institut für Informatik, Universität Leipzig
Augustusplatz 10 – 11
04109 Leipzig, Germany
e–mail: graebe@informatik.uni-leipzig.de

This package contains algorithms for computations in commutative algebra closely related to the Gröbner algorithm for ideals and modules. Its heart is a new implementation of the Gröbner algorithm that also allows for the computation of syzygies. This implementation is also applicable to submodules of free modules with generators represented as rows of a matrix. As main topics CALI contains facilities for

- defining rings, ideals and modules,
- computing Gröbner bases and local standard bases,
- computing syzygies, resolutions and (graded) Betti numbers,
- computing (now also weighted) Hilbert series, multiplicities, independent sets, and dimensions,
- computing normal forms and representations,
- computing sums, products, intersections, quotients, stable quotients, elimination ideals etc.,

- primality tests, computation of radicals, unmixed radicals, equidimensional parts, primary decompositions etc. of ideals and modules,
- advanced applications of Gröbner bases (blowup, associated graded ring, analytic spread, symmetric algebra, monomial curves etc.),
- applications of linear algebra techniques to zero dimensional ideals, as *e.g.* the FGLM change of term orders, border bases and affine and projective ideals of sets of points,
- splitting polynomial systems of equations mixing factorisation and the Gröbner algorithm, triangular systems, and different versions of the extended Gröbner factoriser.

There is more extended documentation on this package elsewhere, which includes facilities for tracing and switches to control its behaviour.

Chapter 28

CAMAL: Calculations in Celestial Mechanics

J. P. Fitch
School of Mathematical Sciences, University of Bath
BATH BA2 7AY, England
e–mail: jpff@maths.bath.ac.uk

The CAMAL package provides facilities for calculations in Fourier series similar to those in the specialist Celestial Mechanics system of the 1970s, and the Cambridge Algebra system in particular.

28.1 Operators for Fourier Series

HARMONIC

The celestial mechanics system distinguish between polynomial variables and angular variables. All angles must be declared before use with the HARMONIC function.

```
harmonic theta, phi;
```

FOURIER

The FOURIER function coerces its argument into the domain of a Fourier Series. The expression may contain *sine* and *cosine* terms of linear sums of harmonic variables.

```
fourier sin(theta)
```

Fourier series expressions may be added, subtracted multiplies and differentiated in the usual REDUCE fashion. Multiplications involve the automatic linearisation of products of angular functions.

There are three other functions which correspond to the usual restrictive harmonic differentiation and integration, and harmonic substitution.

HDIFF and HINT

Differentiate or integrate a Fourier expression with respect to an angular variable. Any secular terms in the integration are disregarded without comment.

```
load_package camal;
harmonic u;
bige := fourier (sin(u) + cos(2*u));
aa := fourier 1+hdiff(bige,u);
ff := hint(aa*aa*fourier cc,u);
```

HSUB

The operation of substituting an angle plus a Fourier expression for an angles and expanding to some degree is called harmonic substitution. The function takes 5 arguments; the basic expression, the angle being replaced, the angular part of the replacement, the fourier part of the replacement and a degree to which to expand.

```
harmonic u,v,w,x,y,z;
xx:=hsub(fourier((1-d*d)*cos(u)),u,u-v+w-x-y+z,yy,n);
```

28.2 A Short Example

The following program solves Kepler's Equation as a Fourier series to the degree n.

```
bige := fourier 0;
for k:=1:n do <<
  wtlevel k;
  bige:=fourier e * hsub(fourier(sin u), u, u, bige, k);
>>;
write "Kepler Eqn solution:", bige$
```

Chapter 29

CGB: Comprehensive Gröbner Bases

Andreas Dolzmann & Thomas Sturm
Department of Mathematics and Computer Science
University of Passau
D-94030 Passau, Germany
e-mail: dolzmann@uni-passau.de, sturm@uni-passau.de

29.1 Introduction

Consider the ideal basis $F = \{ax, x + y\}$. Treating a as a parameter, the calling sequence

```
torder({x,y},lex)$
groebner{a*x,x+y};

  {x,y}
```

yields $\{x, y\}$ as reduced Gröbner basis. This is, however, not correct under the specialization $a = 0$. The reduced Gröbner basis would then be $\{x + y\}$. Taking these results together, we obtain $C = \{x+y, ax, ay\}$, which is correct wrt. *all* specializations for a including zero specializations. We call this set C a *comprehensive Gröbner basis* (CGB).

The notion of a CGB and a corresponding algorithm has been introduced bei Weispfenning [20]. This algorithm works by performing case distinctions wrt. parametric coefficient polynomials in order to find out what the head monomials are under all possible specializations. It does thus not only determine a CGB, but even classifies the contained polynomials wrt. the specializations they are relevant for. If we keep the Gröbner bases for all cases separate and associate information on the respective specializations with them, we obtain a *Gröbner system*. For our example, the Gröbner system is the following;

$$\left[\begin{array}{c|c} a \neq 0 & \{x + y, ax, ay\} \\ a = 0 & \{x + y\} \end{array}\right].$$

A CGB is obtained as the union of the single Gröbner bases in a Gröbner system. It has also been shown that, on the other hand, a Gröbner system can easily be reconstructed from a given CGB [20].

The CGB package provides functions for computing both CGB's and Gröbner systems, and for turning Gröbner systems into CGB's.

29.2 Using the REDLOG Package

For managing the conditions occurring with the CGB computations, the CGB package uses the package REDLOG implementing first-order formulas, [6, 8], which is also part of the REDUCE distribution.

29.3 Term Ordering Mode

The CGB package uses the settings made with the function `TORDER` of the GROEBNER package. This includes in particular the choice of the main variables. All variables not mentioned in the variable list argument of `TORDER` are parameters. The only term ordering modes recognized by CGB are `LEX` and `REVGRADLEX`.

29.4 CGB: Comprehensive Gröbner Basis

The function `CGB` expects a list F of expressions. It returns a CGB of F wrt. the current `TORDER` setting.

Example:

```
torder({x,y},lex)$
cgb{a*x+y,x+b*y};

{x + b*y,a*x + y,(a*b - 1)*y}

ws;

{b*y + x,

 a*x + y,

 y*(a*b - 1)}
```

Note that the basis returned by the `CGB` call has not undergone the standard evaluation process: The returned polynomials are ordered wrt. the chosen term order. Reevaluation changes this as can be seen with the output of `WS`.

29.5 GSYS: Gröbner System

The function `GSYS` follows the same calling conventions as `CGB`. It returns the complete Gröbner system represented as a nested list

$$\{\{c_1, \{g_{11}, \ldots, g_{1n_1}\}\}, \ldots, \{c_m, \{g_{m1}, \ldots, g_{1n_m}\}\}\}.$$

The c_i are conditions in the parameters represented as quantifier-free REDLOG formulas. Each choice of parameters will obey at least one of the c_i. Whenever a choice of parameters obeys some c_i, the corresponding $\{g_{i1}, \ldots, g_{in_i}\}$ is a Gröbner basis for this choice.

Example:

```
torder({x,y},lex)$
gsys {a*x+y,x+b*y};

{{a*b - 1 <> 0 and a <> 0,

  {a*x + y,x + b*y,(a*b - 1)*y}},
```

```
{a <> 0 and a*b - 1 = 0,

 {a*x + y,x + b*y}},

{a = 0,{a*x + y,x + b*y}}}
```

As with the function CGB, the contained polynomials remain unevaluated.

Computing a Gröbner system is not harder than computing a CGB. In fact, CGB also computes a Gröbner system and then turns it into a CGB.

29.5.1 Switch CGBGEN: Only the Generic Case

If the switch CGBGEN is turned on, both GSYS and CGB will assume all parametric coefficients to be non-zero ignoring the other cases. For CGB this means that the result equals—up to auto-reduction—that of GROEBNER. A call to GSYS will return this result as a single case including the assumptions made during the computation:

Example:

```
torder({x,y},lex)$
on cgbgen;
gsys{a*x+y,x+b*y};

{{a*b - 1 <> 0 and a <> 0,

  {a*x + y,x + b*y,(a*b - 1)*y}}}

off cgbgen;
```

29.6 GSYS2CGB: Gröbner System to CGB

The call GSYS2CGB turns a given Gröbner system into a CGB by constructing the union of the Gröbner bases of the single cases.

Example:

```
torder({x,y},lex)$
```

```
gsys{a*x+y,x+b*y}$
gsys2cgb ws;

{x + b*y,a*x + y,(a*b - 1)*y}
```

29.7 Switch CGBREAL: Computing over the Real Numbers

All computations considered so far have taken place over the complex numbers, more precisely, over algebraically closed fields. Over the real numbers, certain branches of the CGB computation can become inconsitent though they are not inconsistent over the complex numbers. Consider, e.g., a condition $a^2 + 1 = 0$.

When turning on the switch CGBREAL, all simplifications of conditions are performed over the real numbers. The methods used for this are described in [7].

Example:

```
torder({x,y},lex)$
off cgbreal;
gsys {a*x+y,x-a*y};

   2
{{a  + 1 <> 0 and a <> 0,

                     2
  {a*x + y,x - a*y,(a  + 1)*y}},

              2
 {a <> 0 and a  + 1 = 0,{a*x + y,x - a*y}},

 {a = 0,{a*x + y,x - a*y}}}

on cgbreal;
gsys({a*x+y,x-a*y});

{{a <> 0,

                     2
```

```
  {a*x + y,x - a*y,(a  + 1)*y}},

 {a = 0,{a*x + y,x - a*y}}}
```

29.8 Switches

CGBREAL Compute over the real numbers. See Section 29.7 for details.

CGBGS Gröbner simplification of the condition. The switch CGBGS can be turned on for applying advanced algebraic simplification techniques to the conditions. This will, in general, slow down the computation, but lead to a simpler Gröbner system.

CGBSTAT Statistics of the CGB run. The switch CGBSTAT toggles the creation and output of statistical information on the CGB run. The statistical information is printed at the end of the run.

CGBFULLRED Full reduction. By default, the CGB functions perform full reductions in contrast to pure top reductions. By turning off the switch CGBFULLRED, reduction can be restricted to top reductions.

Chapter 30

CHANGEVR: Change of Independent Variables in DEs

G. Üçoluk
Department of Physics, Middle East Technical University
Ankara, Turkey
e–mail: ucoluk@trmetu.bitnet

The function `CHANGEVAR` has (at least) four different arguments.

- **FIRST ARGUMENT**
  is a list of the dependent variables of the differential equation. If there is only one dependent variable it can be given directly, not as a list.
- **SECOND ARGUMENT**
  is a list of the **new** independent variables, or in the case of only one, the variable.
- **THIRD ARGUMENT, FOURTH** *etc.*
  are equations is of the form

  `old variable = a function in new variables`

  The left hand side cannot be a non-kernel structure. These give the old variables in terms of the new ones.

- **LAST ARGUMENT**
  is a list of algebraic expressions which evaluates to differential equations in the usual list notation. Again it is possible to omit the list form if there is only **one** differential equation.

If the last argument is a list then the result of `CHANGEVAR` is a list too.

It is possible to display the entries of the inverse Jacobian. To do so, turn `ON` the flag `DISPJACOBIAN`.

30.1 An example: the 2-D Laplace Equation

The 2-dimensional Laplace equation in Cartesian coordinates is:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Now assume we want to obtain the polar coordinate form of Laplace equation. The change of variables is:

$$x = r\cos\theta, \qquad y = r\sin\theta$$

The solution using `CHANGEVAR` is

```
CHANGEVAR({u},{r,theta},{x=r*cos theta,y=r*sin theta},
                        {df(u(x,y),x,2)+df(u(x,y),y,2)} );
```

Here we could omit the curly braces in the first and last arguments (because those lists have only one member) and the curly braces in the third argument (because they are optional), but not in the second. So one could equivalently write

```
CHANGEVAR(u,{r,theta},x=r*cos theta,y=r*sin theta,
                      df(u(x,y),x,2)+df(u(x,y),y,2) );
```

The `u(x,y)` operator will be changed to `u(r,theta)` in the result as one would do with pencil and paper. `u(r,theta)` represents the the transformed dependent variable.

Chapter 31

COMPACT: Package for compacting expressions

Anthony C. Hearn
RAND
Santa Monica
CA 90407-2138, U.S.A.
e–mail: hearn@rand.org

COMPACT is a package of functions for the reduction of a polynomial in the presence of side relations. The package defines one operator COMPACT whose syntax is:

COMPACT(<*expression*>, <*list*>):<*expression*>

<*expression*> can be any well-formed algebraic expression, and <*list*> an expression whose value is a list of either expressions or equations. For example

```
compact(x**2+y**3*x-5y,{x+y-z,x-y-z1});
compact(sin(x)**10*cos(x)**3+sin(x)**8*cos(x)**5,
        {cos(x)**2+sin(x)**2=1});
let y = {cos(x)**2+sin(x)**2-1};
compact(sin(x)**10*cos(x)**3+sin(x)**8*cos(x)**5,y);
```

COMPACT applies the relations to the expression so that an equivalent expression results with as few terms as possible. The method used is briefly as follows:

1. Side relations are applied separately to numerator and denominator, so that the problem is reduced to the reduction of a polynomial with respect to a set of polynomial side relations.

2. Reduction is performed sequentially, so that the problem is reduced further to the reduction of a polynomial with respect to a single polynomial relation.

3. The polynomial being reduced is reordered so that the variables (kernels) occurring in the side relation have least precedence.

4. Each coefficient of the remaining kernels (which now only contain the kernels in the side relation) is reduced with respect to that side relation.

5. A polynomial quotient/remainder calculation is performed on the coefficient. The remainder is used instead of the original if it has fewer terms.

6. The remaining expression is reduced with respect to the side relation using a "nearest neighbour" approach.

Chapter 32

CRACK: Solving overdetermined systems of PDEs or ODEs

Thomas Wolf
School of Mathematical Sciences, Queen Mary and Westfield College
University of London
London E1 4NS, England
e–mail: T.Wolf@maths.qmw.ac.uk

Andreas Brand
Institut für Informatik
Friedrich Schiller Universität Jena
07740 Jena, Germany
e–mail: maa@hpux.rz.uni-jena.de

The package CRACK aims at solving or at least partially integrating single ordinary differential equations or partial differential equations (ODEs/PDEs), and systems of them, exactly and in full generality. Calculations done with input DEs include the

- integration of exact DEs and generalised exact DEs
- determination of monomial integrating factors
- direct and indirect separation of DEs
- systematic application of integrability conditions

- solution of single elementary ODEs by using the REDUCE package ODESOLVE (chapter 59).

Input DEs may be polynomially non-linear in the unknown functions and their derivatives and may depend arbitrarily on the independent variables.

Suitable applications of CRACK are the solution of

- overdetermined ODE/PDE-systems (overdetermined here just means that the number of unknown functions of all independent variables is less than the number of given equations for these functions).
- simple non-overdetermined DE-systems (such as characteristic ODE-systems of first order quasilinear PDEs).

The strategy is to have **one** universal program (CRACK) which is as effective as possible for solving overdetermined PDE-systems and many application programs (such as LIEPDE) which merely generate an overdetermined PDE-system depending on what is to be investigated (for example, symmetries or conservation laws).

Examples are:

- the investigation of infinitesimal symmetries of DEs (LIEPDE),
- the determination of an equivalent Lagrangian for second order ODEs (LAGRAN)
- the investigation of first integrals of ODEs which are polynomial in their highest derivative (FIRINT)
- the splitting of an n^{th} order ODE into a first order ODE and an $(n-1)^{th}$ order problem (DECOMP)

Other applications where non-overdetermined problems are treated are

- the application of infinitesimal symmetries (*e.g.* calculated by LIEPDE) in the package APPLYSYM (chapter 21),
- the program QUASILINPDE (also in the package APPLYSYM) for solving single first order quasilinear PDEs.

The kernel package for solving overdetermined or simple non-overdetermined DE-systems is accessible through a call to the program CRACK in the package CRACK. All the application programs (LIEPDE, LAGRAN, FIRINT, DECOMP except APPLYSYM) are contained in the package CRACKAPP. The programs APPLYSYM and QUASILINPDE are contained in the package APPLYSYM (described in chapter 21).

Details of the CRACK applications can be found in the example file.

`CRACK` is called by

$$\texttt{CRACK}(\{equ_1,\ equ_2,\ \ldots,\ equ_m\},\\ \{ineq_1,\ ineq_2,\ \ldots,\ ineq_n\},\\ \{fun_1,\ fun_2,\ \ldots,\ fun_p\},\\ \{var_1,\ var_2,\ \ldots,\ var_q\});$$

m, n, p, q are arbitrary.

- The equ_i are identically vanishing partial differential expressions, *i.e.* they represent equations $0 = equ_i$, which are to be solved for the functions fun_j as far as possible, thereby drawing only necessary conclusions and not restricting the general solution.

- The $ineq_i$ are expressions which must not vanish identically for any solution to be determined, *i.e.* only such solutions are computed for which none of the $ineq_i$ vanishes identically in all independent variables.

- The dependence of the (scalar) functions fun_j on possibly a number of variables is assumed to have been defined with DEPEND rather than declaring these functions as operators. Their arguments may themselves only be independent variables and not expressions.

- The functions fun_j and their derivatives may only occur polynomially. Other unknown functions in equ_i may be represented as operators.

- The var_k are further independent variables, which are not already arguments of any of the fun_j. If there are none then the third argument is the empty list $\{\}$.

- The dependence of the equ_i on the independent variables and on constants and functions other than fun_j is arbitrary.

The result is a list of solutions

$$\{sol_1, \ldots\}$$

where each solution is a list of 3 lists:

$$\begin{array}{l}\{\{con_1,\ con_2, \ldots,\ con_q\}, \\ \ \{fun_a = ex_a,\ \ fun_b = ex_b, \ldots,\ \ fun_p = ex_p\}, \\ \ \{fun_c,\ \ fun_d, \ldots,\ \ fun_r\} \qquad\qquad\qquad\qquad \}\end{array}$$

with integer a, b, c, d, p, q, r. If `CRACK` finds a contradiction as $0 = 1$ then there exists no solution and it returns the empty list $\{\}$. The empty list is also returned if no solution exists which does not violate the inequalities $ineq_i \neq 0$. For example, in the case of a linear system as input, there is at most one solution sol_1.

The expressions con_i (if there are any), are the remaining necessary and sufficient conditions for the functions $fun_c, \ldots, fun_r$ in the third list. Those functions can be original functions from the equations to be solved (of the second argument of the call of `CRACK`) or new functions or constants which arose from integrations. The dependence of new functions on variables is declared with `DEPEND` and to visualise this dependence the algebraic mode function `FARGS`($fun_{\mathtt{i}}$) can be used. If there are no con_i then all equations are solved and the functions in the third list are unconstrained.

The second list contains equations $fun_i = ex_i$ where each fun_i is an original function and ex_i is the computed expression for fun_i.

The exact behaviour of `CRACK` can be modified by internal variables, and there is a help system particularly associated with `CRACK`. Users are referred to the detailed documentation for more information.

Chapter 33

CVIT: Fast calculation of Dirac gamma matrix traces

V. Ilyin, A. Kryukov, A. Rodionov and A. Taranov
Institute for Nuclear Physics
Moscow State University
Moscow, 119899 Russia

The package consists of 5 sections, and provides an alternative to the REDUCE high-energy physics system. Instead of being based on Γ-matrices as a basis for a Clifford algebra, it is based on treating Γ-matrices as 3-j symbols, as described by Cvitanovic.

The functions it provides are the same as those of the standard package. It does have four switches which control its behaviour.

`CVIT`

If it is on then use Kennedy-Cvitanovic algorithm else use standard facilities.

`CVITOP`

Switches on Fierz optimisation. Default is off;

`CVITBTR`

Switches on the bubbles and triangles factorisation. The default is on.

`CVITRACE`

Controls internal tracing of the CVIT package. Default is off.

```
index j1,j2,j3,;

vecdim n$

g(l,j1,j2,j2,j1);


 2
n


g(l,j1,j2)*g(l1,j3,j1,j2,j3);


 2
n

g(l,j1,j2)*g(l1,j3,j1,j3,j2);


n*( - n + 2)
```

Chapter 34

DEFINT: Definite Integration for REDUCE

Kerry Gaskell and Winfried Neun
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraße 7
D–14195 Berlin–Dahlem, Germany
e–mail: neun@zib.de

Stanley L. Kameny
Los Angeles, U.S.A.

REDUCE's definite integration package is able to calculate the definite integrals of many functions, including several special functions. There are a number of parts of this package, including contour integration. The innovative integration process is to represent each function as a Meijer G-function, and then calculating the integral by using the following Meijer G integration formula.

$$\int_0^\infty x^{\alpha-1} G^{st}_{uv}\left(\sigma x \;\middle|\; \begin{matrix}(c_u)\\(d_v)\end{matrix}\right) G^{mn}_{pq}\left(\omega x^{l/k} \;\middle|\; \begin{matrix}(a_p)\\(b_q)\end{matrix}\right) dx = k G^{ij}_{kl}\left(\xi \;\middle|\; \begin{matrix}(g_k)\\(h_l)\end{matrix}\right) \qquad (1)$$

The resulting Meijer G-function is then retransformed, either directly or via a hypergeometric function simplification, to give the answer.

The user interface is via a four argument version of the `INT` operator, with the lower and upper limits added.

```
load_package defint;

int(sin x,x,0,pi/2);

        1
```

```
int(log(x),x,1,5);

        5*log(5) - 4

int(x*e^(-1/2x),x,0,infinity);

        4

int(x^2*cos(x)*e^(-2*x),x,0,infinity);

          4
        -----
         125

int(x^(-1)*besselj(2,sqrt(x)),x,0,infinity);

        1

int(si(x),x,0,y);

        cos(y) + si(y)*y - 1

int(besselj(2,x^(1/4)),x,0,y);

                      1/4
         4*besselj(3,y   )*y
         ---------------------
                 1/4
                y
```

The DEFINT package also defines a number of additional transforms, such as the Laplace transform[1], the Hankel transform, the Y-transform, the K-transform, the StruveH transform, the Fourier sine transform, and the Fourier cosine transform.

```
laplace_transform(cosh(a*x),x);

   - s
---------
  2    2
 a  - s
```

[1] See Chapter 49 for an alternative Laplace transform with inverse Laplace transform

```
laplace_transform(Heaviside(x-1),x);

  1
------
  s
 e *s

hankel_transform(x,x);

         n + 4
  gamma(-------)
           2
-------------------
        n - 2    2
 gamma(-------)*s
          2

fourier_sin(e^(-x),x);

   s
--------
  2
 s  + 1

fourier_cos(x,e^(-1/2*x^2),x);

                                  2
                   i*s           s /2
 sqrt( - pi)*erf(---------)*s + e    *sqrt(2)
                 sqrt(2)
----------------------------------------------
                  2
                 s /2
                e    *sqrt(2)
```

It is possible to the user to extend the pattern-matching process by which the relevant Meijer G representation for any function is found. Details can be found in the complete documentation.

Acknowledgement: This package depends greatly on the pioneering work of Victor Adamchik, to whom thanks are due.

Chapter 35

DESIR: Differential linear homogeneous equation solutions in the neighbourhood of irregular and regular singular points

C. Dicrescenzo, F. Richard–Jung, E. Tournier
Groupe de Calcul Formel de Grenoble
laboratoire TIM3
France
e–mail: dicresc@afp.imag.fr

This software enables the basis of formal solutions to be computed for an ordinary homogeneous differential equation with polynomial coefficients over Q of any order, in the neighbourhood of zero (regular or irregular singular point, or ordinary point).

This software can be used in two ways, directly via the `DELIRE` procedure, or interactively with the `DESIR` procedure. The basic procedure is the fDELIRE procedure which enables the solutions of a linear homogeneous differential equation to be computed in the neighbourhood of zero.

The `DESIR` procedure is a procedure without argument whereby `DELIRE` can be called without preliminary treatment to the data, that is to say, in an interactive autonomous way. This procedure also proposes some transformations on the initial equation. This allows one to start comfortably with an equation which has a non zero singular point, a polynomial right-hand side and parameters.

`delire(x,k,grille,lcoeff,param)`

This procedure computes formal solutions of a linear homogeneous differential equation with polynomial coefficients over Q and of any order, in the neighbourhood of zero, regular or irregular singular point. `x` is the variable, `k` is the number of desired terms (that is for each formal series in x_t appearing in polysol, $a_0 + a_1x_t + a_2x_t^2 + \ldots + a_nx_t^n + \ldots$ we compute the $k+1$ first coefficients a_0, a_1 to a_k. The coefficients of the differential operator as polynomial in x^{grille}. In general grille is 1. The argument `lcoeff` is a list of coefficients of the differential operator (in increasing order of differentiation) and `param` is a list of parameters. The procedure returns the list of general solutions.

```
lcoeff:={1,x,x,x**6};

                           6
        lcoeff := {1,x,x,x }

param:={};

        param := {}

sol:=delire(x,4,1,lcoeff,param);

                  4       3        2
                xt  - 4*xt  + 12*xt  - 24*xt + 24
sol := {{{{0,1,-----------------------------------,1},{
                               12

             }}},

                             4                 3
        {{{0,1,(6*log(xt)*xt  - 18*log(xt)*xt

                                 2
                 + 36*log(xt)*xt  - 36*log(xt)*xt

                      4       3
                 - 5*xt  + 9*xt  - 36*xt + 36)/36,0},{}
```

```
  }},

       1
{{{-------,1,
         4
    4*xt

            4        3         2
    361*xt  + 4*xt  + 12*xt  + 24*xt + 24
   ---------------------------------------,10},
                      24

  {}}}}
```

Chapter 36

DFPART: Derivatives of generic functions

Herbert Melenk
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany
e–mail: melenk@zib.de

The package `DFPART` supports computations with total and partial derivatives of formal function objects. Such computations can be useful in the context of differential equations or power series expansions.

36.1 Generic Functions

A generic function is a symbol which represents a mathematical function. The minimal information about a generic function function is the number of its arguments. In order to facilitate the programming and for a better readable output this package assumes that the arguments of a generic function have default names such as $f(x,y)$, $q(rho,phi)$. A generic function is declared by prototype form in a statement

`GENERIC_FUNCTION` $fname(arg_1, arg_2 \cdots arg_n)$;

where $fname$ is the (new) name of a function and arg_i are symbols for its formal arguments. In the following $fname$ is referred to as "generic function", $arg_1, arg_2 \cdots arg_n$ as "generic arguments" and $fname(arg_1, arg_2 \cdots arg_n)$ as "generic form".

Examples:

```
generic_function f(x,y);
generic_function g(z);
```

After this declaration REDUCE knows that

- there are formal partial derivatives $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$ $\frac{\partial g}{\partial z}$ and higher ones, while partial derivatives of f and g with respect to other variables are assumed as zero,
- expressions of the type $f()$, $g()$ are abbreviations for $f(x, y)$, $g(z)$,
- expressions of the type $f(u, v)$ are abbreviations for $sub(x = u, y = v, f(x, y))$
- a total derivative $\frac{df(u,v)}{dw}$ has to be computed as $\frac{\partial f}{\partial x}\frac{du}{dw} + \frac{\partial f}{\partial y}\frac{dv}{dw}$

36.2 Partial Derivatives

The operator `DFP` represents a partial derivative:

`DFP`$(expr, dfarg_1, dfarg_2 \cdots dfarg_n)$;

where $expr$ is a function expression and $dfarg_i$ are the differentiation variables. Examples:

```
dfp(f(),{x,y});
```

means $\frac{\partial^2 f}{\partial x \partial y}$ and

```
dfp(f(u,v),{x,y});
```

stands for $\frac{\partial^2 f}{\partial x \partial y}(u, v)$. For compatibility with the DF operator the differentiation variables need not be entered in list form; instead the syntax of `DF` can be used, where the function expression is followed by the differentiation variables, eventually with repetition numbers. Such forms are internally converted to the above form with a list as second parameter.

The expression *expr* can be a generic function with or without arguments, or an arithmetic expression built from generic functions and other algebraic parts. In the second case the standard differentiation rules are applied in order to reduce each derivative expressions to a minimal form.

When the switch `NAT` is on partial derivatives of generic functions are printed in standard index notation, that is f_{xy} for $\frac{\partial^2 f}{\partial x \partial y}$ and $f_{xy}(u,v)$ for $\frac{\partial^2 f}{\partial x \partial y}(u,v)$. Therefore single characters should be used for the arguments whenever possible. Examples:

```
generic_function f(x,y);
generic_function g(y);
dfp(f(),x,2);

F
 XX

dfp(f()*g(),x,2);

F  *G()
 XX

dfp(f()*g(),x,y);

F  *G() + F *G
 XY        X  Y
```

The difference between partial and total derivatives is illustrated by the following example:

```
generic_function h(x);
dfp(f(x,h(x))*g(h(x)),x);

F (X,H(X))*G(H(X))
 X

df(f(x,h(x))*g(h(x)),x);
```

```
F (X,H(X))*G(H(X)) + F (X,H(X))*H (X)*G(H(X))
 X                    Y          X

 + G (H(X))*H (X)*F(X,H(X))
    Y        X
```

Normally partial differentials are assumed as non-commutative

```
dfp(f(),x,y)-dfp(f(),y,x);

F   - F
 XY    YX
```

However, a generic function can be declared to have globally interchangeable partial derivatives using the declaration `DFP_COMMUTE` which takes the name of a generic function or a generic function form as argument. For such a function differentiation variables are rearranged corresponding to the sequence of the generic variables.

```
generic_function q(x,y);
dfp_commute q(x,y);
dfp(q(),{x,y,y}) + dfp(q(),{y,x,y}) + dfp(q(),{y,y,x});

3*Q
   XYY
```

If only a part of the derivatives commute, this has to be declared using the standard REDUCE rule mechanism. Please note that then the derivative variables must be written as list.

36.3 Substitutions

When a generic form or a `DFP` expression takes part in a substitution the following steps are performed:

1. The substitutions are performed for the arguments. If the argument list is empty the substitution is applied to the generic arguments of the function; if these change, the resulting forms are used as new actual arguments. If the generic function itself is not affected by the substitution, the process stops here.

2. If the function name or the generic function form occurs as a left hand side in the substitution list, it is replaced by the corresponding right hand side.
3. The new form is partially differentiated according to the list of partial derivative variables.
4. The (eventually modified) actual parameters are substituted into the form for their corresponding generic variables. This substitution is done by name.

Examples:

```
generic_function f(x,y);
sub(y=10,f());

F(X,10)

sub(y=10,dfp(f(),x,2));


F  (X,10)
 XX

sub(y=10,dfp(f(y,y),x,2));

F  (10,10)
 XX

sub(f=x**3*y**3,dfp(f(),x,2));

     3
6*X*Y

generic_function ff(y,z);
sub(f=ff,f(a,b));

FF(B,Z)
```

Chapter 37

DUMMY: Canonical form of expressions with dummy variables

Alain Dresse
Université Libre de Bruxelles
Boulevard du Triomphe, CP 210/01
B–1050 BRUXELLES, Belgium
e–mail: adresse@ulb.ac.be

An expression of the type

$$\sum_{a=1}^{n} f(a)$$

for any n is simply written as

$$f(a)$$

and a is a *dummy* index. If the previous expression is written as

$$\sum_{b=1}^{n} f(b)$$

b is also a dummy index and, obviously we should be able to get the equality

$$f(a) - f(b); \rightarrow 0$$

To declare dummy variables, two declarations are available:

i. `dummy_base <idp>;`

where `idp` is the name of any unassigned identifier.

ii. `dummy_names <d>,<dp>,<dpp> ....;`

The first declares `idp1,···, idpn` as dummy variables *i.e.* all variables of the form "`idxxx`" where `xxx` is a number will be dummy variables, such as `id1, id2, ... , id23`. The second gives special names for dummy variables. All other arguments are assumed to be `free`.
An example:

```
dummy_base dv; ==> dv

          % dummy indices are dv1, dv2, dv3, ...

dummy_names i,j,k; ==> t

           % dummy names are i,j,k.
```

When this is done, an expression like

```
op(dv1)*sin(dv2)*abs(x)*op(i)^3*op(dv2)$
```

is allowed. Notice that, dummy indices may not be repeated (it is not limited to tensor calculus) or that they be repeated many times inside the expression.

By default all operators with dummy arguments are assumed to be *commutative* and without symmetry properties. This can be varied by declarations `NONCOM`, `SYMMETRIC` and `ANTISYMMETRIC` may be used on the operators. They can also be declared anticommutative.

```
anticom ao1, ao2;
```

More complex symmetries can be handled with `SYMTREE`. The corresponding declaration for the Riemann tensor is

```
symtree (r, {!+, {!-, 1, 2}, {!-, 3, 4}});
```

The symbols !*, !+ and !- at the beginning of each list mean that the operator has no symmetry, is symmetric and is antisymmetric with respect to the indices inside the list. Notice that the indices are not designated by their names but merely by their natural order of appearance. 1 means the first written argument of `r`, 2 its second argument *etc.* In the example above r is symmetric with respect to interchange of the pairs of indices 1,2 and 3,4 respectively.

Chapter 38

EDS: Exterior differential systems

David Hartley
Physics and Mathematical Physics
University of Adelaide SA 5005, Australia
e-mail: DHartley@physics.adelaide.edu.au

38.1 Introduction

Exterior differential systems give a geometrical framework for partial differential equations and more general differential geometric problems. The geometrical formulation has several advantages stemming from its coordinate-independence, including superior treatment of nonlinear and global problems. `EDS` provides a number of tools for setting up and manipulating exterior differential systems and implements many features of the theory. Its main strengths are the ability to use anholonomic or moving frames and the care taken with nonlinear problems.

The package is loaded by typing `load eds;`

Reading the full documentation, which comes with this package, is strongly recommended. The test file eds.tst, which is also in the package, provides three inspiring examples on the subject.
EDS uses E. Schrüfer's EXCALC package for the underlying exterior calculus operations.

38.2 Data Structures and Concepts

38.2.1 EDS

A simple ⟨*EDS*⟩, or exterior differential system, is a triple `(S,Ω,M)`, where M is a *coframing*, S is a system on M, and Ω is an independence condition. Exterior differential equations without independence condition are not treated by `EDS`. Ω should be either a decomposable ⟨*p-form*⟩ or a ⟨*system*⟩ of 1-forms on M.
More generally an ⟨*EDS*⟩ is a list of simple ⟨*EDS*⟩ objects where the various coframings are all disjoint.
The solutions of *(S,Ω,M)* are integral manifolds, or immersions on which S vanishes and the rank of Ω is preserved. Solutions at a single point are described by integral elements.

38.2.2 Coframing

Within the context of `EDS`, a *coframing* means a real finite-dimensional differentiable manifold with a given global cobasis. The information about a coframing required by `EDS` is kept in a ⟨*coframing*⟩ object. The cobasis is the identifying element of an `EDS`. In addition to the cobasis, there can be given *coordinates, structure equations* and *restrictions.* In addition to the cobasis, *coordinates, structure equations* and *restrictions* can be given. The coordinates may be an incomplete or overcomplete set. The structure equations express the exterior derivative of the coordinates and cobasis elements as needed. All coordinate differentials must be expressed in terms of the given cobasis, but not all cobasis derivatives need be known. The restrictions are a set of inequalities describing point sets not in the manifold.
Please note that the ⟨*coframing*⟩ object is by no means a full description of a differentiable manifold. However, the ⟨*coframing*⟩ object carries sufficient information about the underlying manifold to allow a range of exterior systems calculations to be carried out.

38.2.3 Systems and background coframing

The label ⟨*system*⟩ refers to a list {<*p-form expr*>,...} of differential forms. If an `EDS` operator also accepts a ⟨*system*⟩ as argument, then any extra information which is required is taken from the background coframing.

It is possible to activate the rules and orderings of a `COFRAMING` operator globally, by making it the *background coframing*. All subsequent `EXCALC` operations will be governed by those rules. Operations on $\langle EDS \rangle$ objects are unaffected, since their coframings are still activated locally.

38.2.4 Integral elements

An $\langle$*integral element*$\rangle$ of an exterior system (S, Ω, M) is a subspace $P \subset T_p M$ of the tangent space at some point $p \in M$. This integral element can be represented by its annihilator $P^\perp \subset T_p^* M$, comprising those 1-forms at p which annihilate every vector in P. This can also be understood as a maximal set of 1-forms at p such that $S \simeq 0 \pmod{P^\perp}$ and the rank of Ω is preserved modulo $P^\perp$.
An $\langle$*integral element*$\rangle$ in EDS is a distribution of 1-forms on M, specified as a $\langle$*system*$\rangle$ of 1-forms.

38.2.5 Properties and normal form

For large problems, it can require a great deal of computation to establish whether, for example, a system is closed or not. In order to save recomputing such properties, an $\langle EDS \rangle$ object carries a list of $\langle$*properties*$\rangle$ of the form

$$\{\langle keyword \rangle \ \texttt{=} \ \langle value \rangle \texttt{,} \cdots\}$$

where $\langle$*keyword*$\rangle$ is one of `closed`, `quasilinear`, `pfaffian` or `involutive`, and $\langle$*value*$\rangle$ is either `0` (false) or `1` (true). These properties are suppressed when an $\langle EDS \rangle$ is printed, unless the `nat` switch is `off`. They can be examined using the `PROPERTIES` operator.
Parts of the theory of exterior differential systems apply only at points on the underlying manifold where the system is in some sense non-singular. To ensure the theory applies, EDS automatically works all exterior systems (S, Ω, M) into a *normal form.* This means that the Pfaffian component of S and the independence condition Ω are in *solved* forms, distinguished terms from the 1-forms in S have been eliminated from the rest of S and from Ω and any 1-forms in S which vanish modulo the independence condition are removed from the system and their coefficients are appended as 0-forms.

38.3 The EDS Package

In the descriptions of the various operators we define the following abbreviations for function parameters:

E, E'	⟨*EDS*⟩
S	⟨*system*⟩
M, N	⟨*coframing*⟩, or a ⟨*system*⟩ specifying a ⟨*coframing*⟩
r	⟨*integer*⟩
Ω	⟨*p-form*⟩
f	⟨*map*⟩
rsx	⟨*list of inequalities*⟩
cob	⟨*list of 1-form variables*⟩
crd, dep, ind	⟨*list of 0-form variables*⟩
drv	⟨*list of rules for exterior derivatives*⟩
pde	⟨*list of expressions or equations*⟩
X	⟨*transform*⟩
T	⟨*tableau*⟩
P	⟨*integral element*⟩

38.3.1 Constructing EDS objects

An EDS ⟨*coframing*⟩ is constructed using the `COFRAMING` operator. In one form it examines the argument for 0-form and 1-form variables. The more basic syntax takes the ⟨*cobasis*⟩ as a list of 1-forms, ⟨*coordinates*⟩ as a list of 0-forms, ⟨*restrictions*⟩ as a list of inequalities and ⟨*structure equations*⟩ as a list giving the exterior derivatives of the coordinates and cobasis elements. All arguments except the cobasis are optional.
A simple ⟨*EDS*⟩ is constructed using the `EDS` operator where the ⟨*indep. condition*⟩ can be either a decomposable ⟨*p-form*⟩ or a ⟨*system*⟩ of 1-forms. The ⟨*coframing*⟩ and the ⟨*properties*⟩ arguments can be omitted. The *EDS* is put into normal form before being returned. With `SET_COFRAMING` the background coframing is set.
The operator `PDS2EDS` encodes a PDE system into an ⟨*EDS*⟩ object.

`COFRAMING`(cob,crd,rsx,drv)	`COFRAMING`(S)	`EDS`(S,Ω,M)
`CONTACT`(r,M,N)	`PDE2EDS`(pde,dep,ind)	`SET_COFRAMING`(M)
`SET_COFRAMING`(E)	`SET_COFRAMING`()	

Example:

```
1: load eds;

2: pform {x,y,z,p,q}=0,{e(i),w(i,j)}=1;

3: indexrange {i,j,k}={1,2},{a,b,c}={3};

4: eds({d z - p*d x - q*d y, d p^d q},{d x,d y});

EDS({d z - p*d x - q*d y,d p^d q},d x^d y)

5: OMrules:=index_expand {d e(i)=>-w(i,-j)^e(j),w(i,-j)+w(j,-i)=>0}$

6: eds({e(a)},{e(i)}) where OMrules;

      3    1  2
EDS({e },{e ,e })

7: coframing ws;
            3  2    1  2         1        2  2
coframing({e ,w   ,e ,e },{},{d e  =>  - e ^w   ,
                1                              1
               2     1  2
            d e  => e ^w   },{})
                         1
```

38.3.2 Inspecting EDS objects

Using these operators you can get parts of your ⟨*EDS*⟩ object. The `PROPERTIES`(E) operator for example returns a list of properties which are normally not printed out, unless the `NAT` switch is off.

`COFRAMING`(E)	`COFRAMING`()	`COBASIS`(M)
`COBASIS`(E)	`COORDINATES`(M)	`COORDINATES`(E)
`STRUCTURE_EQUATIONS`(M)	`STRUCTURE_EQUATIONS`(E)	`RESTRICTIONS`(M)
`RESTRICTIONS`(E)	`SYSTEM`(E)	`INDEPENDENCE`(E)
`PROPERTIES`(E)	`ONE_FORMS`(E)	`ONE_FORMS`(S)
`ZERO_FORMS`(E)	`ZERO_FORMS`(S)	

Example:

```
8: depend u,x,y; depend v,x,y;

9: pde2eds({df(u,y,y)=df(v,x),df(v,y)=y*df(v,x)});

EDS({d u - u *d x - u *d y, d u  - u   *d x - u   *d y,
            x        y          x    x x        y x

     d u  - u   *d x - v *d y, d v - v *d x - v *y*d y},d x^d y)
        y    y x        x             x        x

10: dependencies;

{{u,y,x},{v,y,x}}

11: coordinates contact(3,{x},{u});

{x,u,u ,u   ,u     }
      x  x x  x x x

12: fdomain u=u(x);

13: coordinates {d u+d y};

{x,y}
```

38.3.3 Manipulating EDS objects

These operators allow you to manipulate your ⟨*EDS*⟩ objects. The `AUGMENT`(E,S) operator, see example below, appends the extra forms in the second argument to the system part of the first. The original ⟨*EDS*⟩ remains unchanged. As another example by using the `TRANSFORM` operator a change of the cobasis is made, where the argument ⟨*transform*⟩ is a list of substitutions.

`AUGMENT`(E,S)	*M* `CROSS` *N*	*E* `CROSS` *N*	`PULLBACK(E,f)`
`PULLBACK`(S,f)	`PULLBACK`(Ω,f)	`PULLBACK`(M,f)	`RESTRICT`(E,f)
`RESTRICT`(S,f)	`RESTRICT`(Ω,f)	`RESTRICT`(M,f)	`TRANSFORM`(M,X)
`TRANSFORM`(E,X)	`TRANSFORM`(S,X)	`TRANSFORM`(Ω,X)	`LIFT(E)`

Example:

```
% Non-Pfaffian system for a Monge-Ampere equation

14: PFORM {x,y,z}=0$

15: S := CONTACT(1,{x,y},{z});

s := EDS({d z - z *d x - z *d y},d x^d y)
                 x        y

16: S:= AUGMENT(S,{d z(-x)^d z(-y)});

s := EDS({d z - z *d x - z *d y,
                 x        y

          d z ^d z },d x^d y)
             x    y
```

38.3.4 Analysing and Testing exterior systems

Analysing exterior systems

This section introduces higher level operators for extracting information about exterior systems. Many of them require a ⟨*EDS*⟩ in normal form generated in positive degree as input, but some can also analyse a ⟨*system*⟩ or a single ⟨*p-form*⟩.

`CARTAN_SYSTEM`(E)	`CARTAN_SYSTEM`(S)	`CARTAN_SYSTEM`(Ω)
`CAUCHY_SYSTEM`(E)	`CAUCHY_SYSTEM`(S)	`CAUCHY_SYSTEM`(Ω)
`CHARACTERS`(E)	`CHARACTERS`(T)	`CHARACTERS`(E,P)
`CLOSURE`(E)	`DERIVED_SYSTEM`(E)	`DERIVED_SYSTEM`(S)
`DIM_GRASSMANN_VARIETY`(E)	`DIM_GRASSMANN_VARIETY`(E,P)	`DIM`(M)
`DIM`(E)	`INVOLUTION`(E)	`LINEARISE`(E,P)
`INTEGRAL_ELEMENT`(E)	`PROLONG`(E)	`TABLEAU`(E)
`TORSION`(E)	`GRASSMANN_VARIETY`(E)	

Testing exterior systems

The following operators allow various properties of an ⟨*EDS*⟩ to be checked. The result is either a **1** or a **0**, so these operators can be used in boolean expressions. Since checking these properties is very time-consuming, the result of the first test is stored on the ⟨*properties*⟩ record of an ⟨*EDS*⟩ to avoid re-checking. This memory can be cleared using the `CLEANUP` opearator.

`CLOSED`(E)	`CLOSED`(S)	`CLOSED`(Ω)	`INVOLUTIVE`(E)
`PFAFFIAN`(E)	`QUASILINEAR`(E)	`SEMILINEAR`(E)	E `EQUIV` E'

38.3.5 Switches

EDS provides several switches to govern the display of information and enhance the speed or reliability of the calculations. For example the switch `EDSVERBOSE` if `ON` will display additional information as the calculation progresses, which might generate too much output for larger problems.
All switches are `OFF` by default.

`EDSVERBOSE` `EDSDEBUG` `EDSSLOPPY` `EDSDISJOINT` `RANPOS` `GENPOS`

38.3.6 Auxilliary functions

The operators of this section are designed to ease working with exterior forms and exterior systems in REDUCE .

`COORDINATES`(S)	`INVERT`(X)	`STRUCTURE_EQUATIONS`(X)
`STRUCTURE_EQUATIONS`(X,X^{-1})	`LINEAR_DIVISORS`(Ω)	`EXFACTORS`(Ω)
`INDEX_EXPAND`(ANY)	`PDE2JET`(pde,dep,ind)	`MKDEPEND`(list)
`DISJOIN`(f,g,...)	`CLEANUP`(E)	`CLEANUP`(M)
`REORDER`(E)	`REORDER`(M)	

38.3.7 Experimental Functions

The following operators are experimental facilities since, they are either algorithmically not well-founded, or their implementation is very unstable, or they have known bugs.

`POINCARE`(Ω)	`INVARIANTS`(E,crd)	`INVARIANTS`(S,crd)
`SYMBOL_RELATIONS`(E,π)	`SYMBOL_MATRIX`(E,ξ)	`CHARACTERISTIC_VARIETY`(E,ξ)

Example:

```
17: % Riemann invariants for Euler-Poisson-Darboux equation.
17: % Set up the EDS for the equation, and examine tableau.
17: depend u,x,y; EPD :=PDE2EDS{DF(u,x,y)=-(df(u,x)+df(u,y))/(x+y)}$

19: tableau EPD;

[d u       0   ]
[   x x        ]
[              ]
[  0     d u   ]
[           y y]

20: % 1-form dx is characteristic: construct characteristic EDS.
20: xvars {}; C := cartan_system select(~f^d x=0,system closure epd)$

22: S := augment(eds(system EPD,d y),C)$

23: % Compute derived flag
23: while not equiv(S,S1 := derived_system S) do S := S1;

24: % Stabilised. Find the Riemann invariants.
24: invariants(S,reverse coordinates S);

{x,

 u *x + u *y + u,
  x      x

  - u   *x - u   *y - 2*u }
     x x      x x        x
```

Chapter 39

EXCALC: A differential geometry package

Eberhard Schrüfer
GMD, Institut I1
Postfach 1316
53757 St. Augustin, GERMANY
e–mail: schruefer@gmd.de

EXCALC is designed for easy use by all who are familiar with the calculus of Modern Differential Geometry. Its syntax is kept as close as possible to standard textbook notations. Therefore, no great experience in writing computer algebra programs is required. It is almost possible to input to the computer the same as what would have been written down for a hand-calculation. For example, the statement

```
f*x^y + u _| (y^z^x)
```

would be recognized by the program as a formula involving exterior products and an inner product. The program is currently able to handle scalar-valued exterior forms, vectors and operations between them, as well as non-scalar valued forms (indexed forms). With this, it should be an ideal tool for studying differential equations, doing calculations in general relativity and field theories, or doing such simple things as calculating the Laplacian of a tensor field for an arbitrary given frame. With the increasing popularity of this calculus, this program should have an application in almost any field of physics and mathematics.

39.1 Declarations

Geometrical objects like exterior forms or vectors are introduced to the system by declaration commands. The declarations can appear anywhere in a program, but must, of course, be made prior to the use of the object. Everything that has no declaration is treated as a constant; therefore zero-forms must also be declared.

An exterior form is introduced by

PFORM <*declaration*$_1$>, <*declaration*$_2$>, . . . ;

where

<*declaration*> ::= <*name*> | <*list of names*>=<*number*> | <*identifier*> | <*expression*>
<*name*> ::= <*identifier*> | <*identifier*>(<*arguments*>)

For example

```
pform u=k,v=4,f=0,w=dim-1;
```

declares `U` to be an exterior form of degree `K`, `V` to be a form of degree 4, `F` to be a form of degree 0 (a function), and `W` to be a form of degree `DIM`-1.

The declaration of vectors is similar. The command `TVECTOR` takes a list of names.

TVECTOR <*name*$_1$>, <*name*$_2$>, . . . ;

For example, to declare `X` as a vector and `COMM` as a vector with two indices, one would say

```
tvector x,comm(a,b);
```

The exterior degree of a symbol or a general expression can be obtained with the function

EXDEGREE <*expression*>;

Example:

```
exdegree(u + 3*chris(k,-k));

1
```

39.2 Exterior Multiplication

Exterior multiplication between exterior forms is carried out with the nary infix operator ^ (wedge). Factors are ordered according to the usual ordering in REDUCE using the commutation rule for exterior products.

```
pform u=1,v=1,w=k;

u^v;

U^V

v^u;

- U^V

u^u;

0

w^u^v;

      K
( - 1) *U^V^W

(3*u-a*w)^(w+5*v)^u;

A*(5*U^V^W - U^W^W)
```

It is possible to declare the dimension of the underlying space by

SPACEDIM <*number*> | <*identifier*>;

If an exterior product has a degree higher than the dimension of the space, it is replaced by 0:

39.3 Partial Differentiation

Partial differentiation is denoted by the operator `@`. Its capability is the same as the REDUCE `DF` operator.

Example 6

```
@(sin x,x);

COS(X)

@(f,x);

0
```

An identifier can be declared to be a function of certain variables. This is done with the command `FDOMAIN`. The following would tell the partial differentiation operator that `F` is a function of the variables `X` and `Y` and that `H` is a function of `X`.

```
fdomain f=f(x,y),h=h(x);
```

Applying `@` to `F` and `H` would result in

```
@(x*f,x);

F + X*@  F
       X

@(h,y);

0
```

The partial derivative symbol can also be an operator with a single argument. It then represents a natural base element of a tangent vector.

39.4 Exterior Differentiation

Exterior differentiation of exterior forms is carried out by the operator `d`. Products are normally differentiated out,

```
pform x=0,y=k,z=m;

d(x * y);

X*d Y + d X^Y
```

This expansion can be suppressed by the command `NOXPND D`. Expansion is performed again when the command `XPND D` is executed.

If an argument of an implicitly defined function has further dependencies the chain rule will be applied *e.g.*

```
fdomain y=y(z);

d f;

@  F*d X + @  F*@  Y*d Z
 X          Y    Z
```

Expansion into partial derivatives can be inhibited by `NOXPND @` and enabled again by `XPND @`.

39.5 Inner Product

The inner product between a vector and an exterior form is represented by the diphthong _| (underscore or-bar), which is the notation of many textbooks. If the exterior form is an exterior product, the inner product is carried through any factor.

Example 7

```
pform x=0,y=k,z=m;

tvector u,v;

u _| (x*y^z);

         K
X*(( - 1) *Y^U _| Z + U _| Y^Z)
```

39.6 Lie Derivative

The Lie derivative can be taken between a vector and an exterior form or between two vectors. It is represented by the infix operator |_ . In the case of Lie differentiating, an exterior form by a vector, the Lie derivative is expressed through inner products and exterior differentiations, *i.e.*

```
pform z=k;

tvector u;

u |_ z;

U _| d Z + d(U _| Z)
```

39.7 Hodge-* Duality Operator

The Hodge-* duality operator maps an exterior form of degree `K` to an exterior form of degree `N-K`, where `N` is the dimension of the space. The double application of the operator must lead back to the original exterior form up to a factor. The following example shows how the factor is chosen here

```
spacedim n;
pform x=k;

# # x;

         2
       (K   + K*N)
( - 1)            *X*SGN
```

The indeterminate SGN in the above example denotes the sign of the determinant of the metric. It can be assigned a value or will be automatically set if more of the metric structure is specified (via COFRAME), *i.e.* it is then set to $g/|g|$, where g is the determinant of the metric. If the Hodge-* operator appears in an exterior product of maximal degree as the leftmost factor, the Hodge-* is shifted to the right according to

```
pform {x,y}=k;
```

```
# x ^ y;

        2
      (K  + K*N)
( - 1)           *X^# Y
```

39.8 Variational Derivative

The function VARDF returns as its value the variation of a given Lagrangian n-form with respect to a specified exterior form (a field of the Lagrangian). In the shared variable BNDEQ!*, the expression is stored that has to yield zero if integrated over the boundary.

Syntax:

VARDF(<*Lagrangian n-form*>,<*exterior form*>)

Example 8

```
spacedim 4;

pform l=4,a=1,j=3;

l:=-1/2*d a ^ # d a - a^# j$  %Lagrangian of the e.m. field

vardf(l,a);

- (# J + d # d A)             %Maxwell's equations

bndeq!*;

- 'A^# d A                    %Equation at the boundary
```

For the calculation of the conserved currents induced by symmetry operators (vector fields), the function NOETHER is provided. It has the syntax:

NOETHER(<*Lagrangian n-form*>,<*field*>,<*symmetry generator*>)

Example 9

```
pform l=4,a=1,f=2;
```

```
spacedim 4;

l:= -1/2*d a^#d a;   %Free Maxwell field;

tvector x(k);        %An unspecified generator;

noether(l,a,x(-k));

( - 2*d(X _|A)^# d A - (X _|d A)^# d A + d A^(X _|# d A))/2
         K               K                      K
```

39.9 Handling of Indices

Exterior forms and vectors may have indices. On input, the indices are given as arguments of the object. A positive argument denotes a superscript and a negative argument a subscript. On output, the indexed quantity is displayed two dimensionally if `NAT` is on. Indices may be identifiers or numbers.

Example 10

```
pform om(k,l)=m,e(k)=1;
e(k)^e(-l);

 K
E ^E
    L

om(4,-2);

  4
OM
   2
```

In certain cases, one would like to inhibit the summation over specified index names, or at all. For this the command

NOSUM <*indexname*$_1$>, ...;

and the switch `NOSUM` are available. The command `NOSUM` has the effect that summation is not performed over those indices which had been listed. The command `RENOSUM` enables summation again. The switch `NOSUM`, if on, inhibits any summation.

It is possible to declare symmetry properties for an indexed quantity by the command `INDEX_SYMMETRIES`. A prototypical example is as follows

```
index_symmetries u(k,l,m,n): symmetric     in {k,l},{m,n}
                             antisymmetric in {{k,l},{m,n}},
                 g(k,l),h(k,l): symmetric;
```

It declares the object `u` symmetric in the first two and last two indices and antisymmetric with respect to commutation of the given index pairs. If an object is completely symmetric or antisymmetric, the indices need not to be given after the corresponding keyword as shown above for `g` and `h`.

39.10 Metric Structures

A metric structure is defined in **EXCALC** by specifying a set of basis one-forms (the coframe) together with the metric.

Syntax:

COFRAME *<identifier><(index$_1$)>=<expression$_1$>*,
<identifier><(index$_2$)>=<expression$_2$>,
.
.
.
<identifier><(index$_n$)>=<expression$_n$>
WITH METRIC *<name>=<expression>*;

This statement automatically sets the dimension of the space and the index range. The clause `WITH METRIC` can be omitted if the metric is Euclidean and the shorthand `WITH SIGNATURE` <***diagonal elements***> can be used in the case of a pseudo-Euclidean metric. The splitting of a metric structure in its metric tensor coefficients and basis one-forms is completely arbitrary including the extremes of an orthonormal frame and a coordinate frame.

Example 11

```
coframe e r=d r, e(ph)=r*d ph
  with metric g=e(r)*e(r)+e(ph)*e(ph);    %Polar coframe
```

The frame dual to the frame defined by the `COFRAME` command can be introduced by **FRAME** command.

FRAME <*identifier*>;

This command causes the dual property to be recognised, and the tangent vectors of the coordinate functions are replaced by the frame basis vectors.

Example 12

```
coframe b r=d r,b ph=r*d ph,e z=d z; %Cylindrical coframe;

frame x; on nero;

x(-k) _| b(l);

    R
NS    := 1
  R

     PH
NS      := 1
  PH

    Z
NS    := 1
  Z

x(-k) |_ x(-l);       %The commutator of the dual frame;


NS     := X  /R
  PH R     PH


NS     := ( - X  )/R  %i.e. it is not a coordinate base;
  R PH         PH
```

As a convenience, the frames can be displayed at any point in a program by the command `DISPLAYFRAME;`.

The Hodge-* duality operator returns the explicitly constructed dual element if applied to coframe base elements. The metric is properly taken into account.

The total antisymmetric Levi-Cevita tensor `EPS` is also available. The value of `EPS` with an even permutation of the indices in a covariant position is taken to be +1.

39.11 Riemannian Connections

The command `RIEMANNCONX` is provided for calculating the connection 1 forms. The values are stored on the name given to `RIEMANNCONX`. This command is far more efficient than calculating the connection from the differential of the basis one-forms and using inner products.

39.12 Ordering and Structuring

The ordering of an exterior form or vector can be changed by the command `FORDER`. In an expression, the first identifier or kernel in the arguments of `FORDER` is ordered ahead of the second, and so on, and ordered ahead of all not appearing as arguments. This ordering is done on the internal level and not only on output. The execution of this statement can therefore have tremendous effects on computation time and memory requirements. `REMFORDER` brings back standard ordering for those elements that are listed as arguments.

An expression can be put in a more structured form by renaming a subexpression. This is done with the command `KEEP` which has the syntax

KEEP $<\mathit{name}_1>=<\mathit{expression}_1>,<\mathit{name}_2>=<\mathit{expression}_2>, \ldots$

The capabilities of `KEEP` are currently very limited. Only exterior products should occur as righthand sides in `KEEP`.

Note: This is just an introduction to the full power of `EXCALC`. The reader if referred to the full documentation.

Chapter 40

FIDE: Finite difference method for partial differential equations

Richard Liska
Faculty of Nuclear Science and Physical Engineering
Technical University of Prague
Brehova 7, 115 19 Prague 1, Czech Republic
e–mail: tjerl@aci.cvut.cz

The FIDE package performs automation of the process of numerical solving partial differential equations systems (PDES) by generating finite difference methods. In the process one can find several stages in which computer algebra can be used for performing routine analytical calculations, namely: transforming differential equations into different coordinate systems, discretisation of differential equations, analysis of difference schemes and generation of numerical programs. The FIDE package consists of the following modules:

EXPRES for transforming PDES into any orthogonal coordinate system.

IIMET for discretisation of PDES by integro-interpolation method.

APPROX for determining the order of approximation of difference scheme.

CHARPOL for calculation of amplification matrix and characteristic polynomial of difference scheme, which are needed in Fourier stability analysis.

HURWP for polynomial roots locating necessary in verifying the von Neumann stability condition.

LINBAND for generating the block of FORTRAN code, which solves a system of linear algebraic equations with band matrix appearing quite often in difference schemes.

For more details on this package are given in the FIDE documentation, and in the examples. A flavour of its capabilities can be seen from the following simple example.

```
off exp;

factor diff;

on rat,eqfu;

% Declare which indexes will be given to coordinates
coordinates x,t into j,m;

% Declares uniform grid in x coordinate
grid uniform,x;

% Declares dependencies of functions on coordinates
dependence eta(t,x),v(t,x),eps(t,x),p(t,x);

% Declares p as known function
given p;

same eta,v,p;

iim a, eta,diff(eta,t)-eta*diff(v,x)=0,
    v,diff(v,t)+eta/ro*diff(p,x)=0,
    eps,diff(eps,t)+eta*p/ro*diff(v,x)=0;


*****************************
*****      Program      *****          IIMET Ver 1.1.2
*****************************

      Partial Differential Equations
```

```
      ==============================

diff(eta,t) - diff(v,x)*eta    =    0

 diff(p,x)*eta
--------------- + diff(v,t)    =    0
      ro

                diff(v,x)*eta*p
diff(eps,t) + -----------------    =    0
                       ro


 Backtracking needed in grid optimalization
0 interpolations are needed in x coordinate
  Equation for eta variable is integrated in half grid point
  Equation for v variable is integrated in half grid point
  Equation for eps variable is integrated in half grid point
0 interpolations are needed in t coordinate
  Equation for eta variable is integrated in half grid point
  Equation for v variable is integrated in half grid point
  Equation for eps variable is integrated in half grid point

         Equations after Discretization Using IIM :
         ==========================================

(4*(eta(j,m + 1) - eta(j,m) - eta(j + 1,m)

     + eta(j + 1,m + 1))*hx - (

    (eta(j + 1,m + 1) + eta(j,m + 1))

    *(v(j + 1,m + 1) - v(j,m + 1))

     + (eta(j + 1,m) + eta(j,m))*(v(j + 1,m) - v(j,m)))

 *(ht(m + 1) + ht(m)))/(4*(ht(m + 1) + ht(m))*hx)   =   0


(4*(v(j,m + 1) - v(j,m) - v(j + 1,m) + v(j + 1,m + 1))*hx*ro

  + ((eta(j + 1,m + 1) + eta(j,m + 1))

     *(p(j + 1,m + 1) - p(j,m + 1))
```

```
      + (eta(j + 1,m) + eta(j,m))*(p(j + 1,m) - p(j,m)))

 *(ht(m + 1) + ht(m)))/(4*(ht(m + 1) + ht(m))*hx*ro)   =   0


(4*(eps(j,m + 1) - eps(j,m) - eps(j + 1,m)

     + eps(j + 1,m + 1))*hx*ro + ((

       eta(j + 1,m + 1)*p(j + 1,m + 1)

        + eta(j,m + 1)*p(j,m + 1))

    *(v(j + 1,m + 1) - v(j,m + 1)) +

    (eta(j + 1,m)*p(j + 1,m) + eta(j,m)*p(j,m))

    *(v(j + 1,m) - v(j,m)))*(ht(m + 1) + ht(m)))/(4

   *(ht(m + 1) + ht(m))*hx*ro)   =   0


clear a;

clearsame;

cleargiven;
```

Chapter 41

FPS: Automatic calculation of formal power series

Wolfram Koepf and Winfried Neun
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany
e–mail: Koepf@zib.de and Neun@zib.de

This package can expand functions of certain type into their corresponding Laurent-Puiseux series as a sum of terms of the form

$$\sum_{k=0}^{\infty} a_k (x - x_0)^{k/n+s}$$

where s is the 'shift number', n is the 'Puiseux number', and x_0 is the 'point of development'. The following types are supported:

- **functions of 'rational type'**, which are either rational or have a rational derivative of some order;
- **functions of 'hypergeometric type'** where a_{k+m}/a_k is a rational function for some integer m, the 'symmetry number';
- **functions of 'exp-like type'** which satisfy a linear homogeneous differential equation with constant coefficients.

`FPS(f,x,x0)` tries to find a formal power series expansion for `f` with respect to the variable `x` at the point of development `x0`. It also works for formal

Laurent (negative exponents) and Puiseux series (fractional exponents). If the third argument is omitted, then `x0:=0` is assumed.

Example: `FPS(asin(x)^2,x)` results in

```
         2*k  2*k             2  2
        x   *2   *factorial(k) *x
infsum(----------------------------,k,0,infinity)
        factorial(2*k + 1)*(k + 1)
```

If possible, the output is given using factorials. In some cases, the use of the Pochhammer symbol `pochhammer(a,k)`$:= a(a+1)\cdots(a+k-1)$ is necessary.

`SimpleDE(f,x)` tries to find a homogeneous linear differential equation with polynomial coefficients for f with respect to x. Make sure that y is not a used variable. The setting `factor df;` is recommended to receive a nicer output form.

Examples: `SimpleDE(asin(x)^2,x)` then results in

```
              2
df(y,x,3)*(x  - 1) + 3*df(y,x,2)*x + df(y,x)
```

The depth for the search of a differential equation for `f` is controlled by the variable `fps_search_depth`; higher values for `fps_search_depth` will increase the chance to find the solution, but increases the complexity as well. The default value for `fps_search_depth` is 5. For `FPS(sin(x^(1/3)),x)`, or `SimpleDE(sin(x^(1/3)),x)` *e.g.*, a setting `fps_search_depth:=6` is necessary.

The output of the FPS package can be influenced by the switch `tracefps`. Setting `on tracefps` causes various prints of intermediate results.

Chapter 42

GENTRAN: A code generation package

Barbara L. Gates
RAND
Santa Monica CA 90407-2138
U.S.A.

Michael C. Dewar
School of Mathematical Sciences, The University of Bath
Bath BA2 7AY, England

e–mail: mcd@maths.bath.ac.uk

GENTRAN is an automatic code GENerator and TRANslator which runs under REDUCE. It constructs complete numerical programs based on sets of algorithmic specifications and symbolic expressions. Formatted FORTRAN, RATFOR, PASCAL or C code can be generated through a series of interactive commands or under the control of a template processing routine. Large expressions can be automatically segmented into subexpressions of manageable size, and a special file-handling mechanism maintains stacks of open I/O channels to allow output to be sent to any number of files simultaneously and to facilitate recursive invocation of the whole code generation process. GENTRAN provides the flexibility necessary to handle most code generation applications. It is designed to work with the SCOPE code optimiser.

GENTRAN is a large system with a great many options. This section will only describe the FORTRAN generation facilities, and in broad outline only.

The full manual is available as part of the REDUCE documentation.

42.1 Simple Use

A substantial subset of all expressions and statements in the REDUCE programming language can be translated directly into numerical code. The **GENTRAN** command takes a REDUCE expression, statement, or procedure definition, and translates it into code in the target language.

Syntax:

GENTRAN *stmt* [**OUT** *f1,f2,... ,fn*]*;*

stmt is any REDUCE expression, statement (simple, compound, or group), or procedure definition that can be translated by GENTRAN into the target language. *stmt* may contain any number of calls to the special functions **EVAL**, **DECLARE**, and **LITERAL**. *f1,f2,... ,fn* is an optional argument list containing one or more *f*'s, where each *f* is one of:

an atom	=	an output file
T	=	the terminal
NIL	=	the current output file(s)
ALL!*	=	all files currently open for output by GENTRAN (see section 42.6)

If the optional part of the command is not given, generated code is simply written to the current output file. However, if it is given, then the current output file is temporarily overridden. Generated code is written to each file represented by *f1,f2,... ,fn* for this command only. Files which were open prior to the call to **GENTRAN** will remain open after the call, and files which did not exist prior to the call will be created, opened, written to, and closed. The output stack will be exactly the same both before and after the call.

GENTRAN returns the name(s) of the file(s) to which code was written.

```
1: GENTRANLANG!* := 'FORTRAN$

2: GENTRAN
2:     FOR I:=1:N DO
2:         V(I) := 0$
```

```
      DO 25001 I=1,N
          V(I)=0.0
25001 CONTINUE
```

42.2 Precision

By default **GENTRAN** generates constants and type declarations in single precision form. If the user requires double precision output then the switch **DOUBLE** must be set **ON**.

To ensure the correct number of floating point digits are generated it may be necessary to use either the **PRECISION** or **PRINT!-PRECISION** commands. The former alters the number of digits REDUCE calculates, the latter only the number of digits REDUCE prints. Each takes an integer argument. It is not possible to set the printed precision higher than the actual precision. Calling **PRINT!-PRECISION** with a negative argument causes the printed precision to revert to the actual precision.

42.2.1 The EVAL Function

Syntax:

> **EVAL** *exp*

Argument:

> *exp* is any REDUCE expression or statement which, after evaluation by REDUCE, results in an expression that can be translated by GENTRAN into the target language.

When **EVAL** is called on an expression which is to be translated, it tells **GENTRAN** to give the expression to REDUCE for evaluation first, and then to translate the result of that evaluation.

```
f;

   2
2*X  - 5*X + 6
```

We wish to generate an assignment statement for the quotient of F and its derivative.

```
1: GENTRAN
1:      Q := EVAL(F)/EVAL(DF(F,X))$

      Q=(2.0*X**2-(5.0*X)+6.0)/(4.0*X-5.0)
```

42.2.2 The :=: Operator

In many applications, assignments must be generated in which the left-hand side is some known variable name, but the right-hand side is an expression that must be evaluated. For this reason, a special operator is provided to indicate that the expression on the right-hand side is to be evaluated prior to translation. This special operator is **:=:** (*i.e.* the usual REDUCE assignment operator with an extra ":" on the right).

Example 13

```
1: GENTRAN
1:  DERIV :=: DF(X^4-X^3+2*x^2+1,X)$

      DERIV=4.0*X**3-(3.0*X**2)+4.0*X
```

42.2.3 The ::= Operator

When assignments to matrix or array elements must be generated, many times the indices of the element must be evaluated first. The special operator **::=** can be used within a call to **GENTRAN** to indicate that the indices of the matrix or array element on the left-hand side of the assignment are to be evaluated prior to translation. (This is the usual REDUCE assignment operator with an extra ":" on the left.)

Example 14

We wish to generate assignments which assign zeros to all elements on the main diagonal of M, an n x n matrix.

```
10: FOR j := 1 : 8 DO
10:      GENTRAN
10:          M(j,j) ::= 0$
```

```
M(1,1)=0.0
M(2,2)=0.0
:
:
M(8,8)=0.0
```

LSETQ may be used interchangeably with ::= on input.

42.2.4 The ::=: Operator

In applications in which evaluated expressions are to be assigned to array elements with evaluated subscripts, the ::=: operator can be used. It is a combination of the ::= and :=: operators described in sections 42.2.2 and 42.2.3.

Example 15

The following matrix, M, has been derived symbolically:

```
( A   0  -1   1)
(              )
( 0   B   0   0)
(              )
( -1  0   C  -1)
(              )
( 1   0  -1   D)
```

We wish to generate assignment statements for those elements on the main diagonal of the matrix.

```
10: FOR j := 1 : 4 DO
10:     GENTRAN
10:         M(j,j) ::=: M(j,j)$

      M(1,1)=A
      M(2,2)=B
      M(3,3)=C
      M(4,4)=D
```

The alternative alphanumeric identifier associated with **::=:** is **LRSETQ**.

42.3 Explicit Type Declarations

Type declarations are automatically generated each time a subprogram heading is generated. Type declarations are constructed from information stored in the GENTRAN symbol table. The user can place entries into the symbol table explicitly through calls to the special GENTRAN function **DECLARE**.

Syntax:

DECLARE *v1,v2,... ,vn* **:** *type;*

or

DECLARE
<<
v11,v12,... ,v1n **:** *type1;*
v21,v22,... ,v2n **:** *type2;*
:
:
vn1,vnn,... ,vnn **:** *typen;*
>>;

Arguments:

> Each *v1,v2,... ,vn* is a list of one or more variables (optionally subscripted to indicate array dimensions), or variable ranges (two letters separated by a "-"). *v*'s are not evaluated unless given as arguments to **EVAL**.
>
> Each *type* is a variable type in the target language. Each must be an atom, optionally preceded by the atom **IMPLICIT**. *type*'s are not evaluated unless given as arguments to **EVAL**.

The **DECLARE** statement can also be used to declare subprogram types (*i.e.* **SUBROUTINE** or **FUNCTION**) for FORTRAN and RATFOR code, and function types for all four languages.

42.4 Expression Segmentation

Symbolic derivations can easily produce formulas that can be anywhere from a few lines to several pages in length. Such formulas can be translated into numerical assignment statements, but unless they are broken into smaller pieces they may be too long for a compiler to handle. (The maximum number of continuation lines for one statement allowed by most FORTRAN compilers is only 19.) Therefore GENTRAN contains a segmentation facility which automatically *segments*, or breaks down unreasonably large expressions.

The segmentation facility generates a sequence of assignment statements, each of which assigns a subexpression to an automatically generated temporary variable. This sequence is generated in such a way that temporary variables are re-used as soon as possible, thereby keeping the number of automatically generated variables to a minimum. The facility can be turned on or off by setting the mode switch **GENTRANSEG** accordingly (*i.e.* by calling the REDUCE function **ON** or **OFF** on it). The user can control the maximum allowable expression size by setting the variable **MAXEXPPRINTLEN!*** to the maximum number of characters allowed in an expression printed in the target language (excluding spaces automatically printed by the formatter). The **GENTRANSEG** switch is on initially, and **MAXEXPPRINTLEN!*** is initialised to 800.

42.5 Template Processing

In some code generation applications pieces of the target numerical program are known in advance. A *template* file containing a program outline is supplied by the user, and formulas are derived in REDUCE, converted to numerical code, and inserted in the corresponding places in the program outline to form a complete numerical program. A template processor is provided by GENTRAN for use in these applications.

Syntax:

> **GENTRANIN** *f1,f2,... ,fm* [**OUT** *f1,f2,... ,fn*];

Arguments:

f1,f2,... ,fm is an argument list containing one or more *f*'s, where each *f* is one of:

an atom	=	a template (input) file
T	=	the terminal

f1,f2,... ,fn is an optional argument list containing one or more *f*'s, where each *f* is one of:

an atom	=	an output file
T	=	the terminal
NIL	=	the current output file(s)
ALL!*	=	all files currently open for output by GENTRAN (see section 42.6)

GENTRANIN processes each template file *f1,f2,... ,fm* sequentially.

A template file may contain any number of parts, each of which is either an active or an inactive part. All active parts start with the character sequence **;BEGIN;** and end with **;END;**. The end of the template file is indicated by an extra **;END;** character sequence.

Inactive parts of template files are assumed to contain code in the target language. All inactive parts are copied to the output.

Active parts may contain any number of REDUCE expressions, statements, and commands. They are not copied directly to the output. Instead, they are given to REDUCE for evaluation in algebraic mode. All output generated by each evaluation is sent to the output file(s). Returned values are only printed on the terminal.

Active parts will most likely contain calls to **GENTRAN** to generate code. This means that the result of processing a template file will be the original template file with all active parts replaced by generated code.

If **OUT** *f1,f2,... ,fn* is not given, generated code is simply written to the current-output file.

However, if **OUT** *f1,f2,... ,fn* is given, then the current-output file is temporarily overridden. Generated code is written to each file represented by *f1,f2,... ,fn* for this command only. Files which were open prior to the call to **GENTRANIN** will remain open after the call, and files which did not exist prior to the call will be created, opened, written to, and closed. The output-stack will be exactly the same both before and after the call.

GENTRANIN returns the names of all files written to by this command.

Example 16

Suppose we wish to generate a FORTRAN subprogram to compute the determinant of a 3 x 3 matrix. We can construct a template file with an outline of the FORTRAN subprogram and REDUCE and GENTRAN commands to fill it in:

Contents of file `det.tem`:

```
      REAL FUNCTION DET(M)
      REAL M(3,3)
;BEGIN;
      OPERATOR M$
      MATRIX MM(3,3)$
      MM := MAT( (M(1,1),M(1,2),M(1,3)),
                 (M(2,1),M(2,2),M(2,3)),
                 (M(3,1),M(3,2),M(3,3)) )$
      GENTRAN DET :=: DET(MM)$
;END;
      RETURN
      END
;END;
```

Now we can generate a FORTRAN subprogram with the following REDUCE session:

```
1: GENTRANLANG!* := 'FORTRAN$

2: GENTRANIN
2:      "det.tem"
2: OUT "det.f"$
```

Contents of file det.f:

```
      REAL FUNCTION DET(M)
      REAL M(3,3)
      DET=M(3,3)*M(2,2)*M(1,1)-(M(3,3)*M(2,1)*M(1,2))-(M(3,2)
     . *M(2,3)*M(1,1))+M(3,2)*M(2,1)*M(1,3)+M(3,1)*M(2,3)*M(1
     . ,2)-(M(3,1)*M(2,2)*M(1,3))
      RETURN
      END
```

42.6 Output Redirection

The **GENTRANOUT** and **GENTRANSHUT** commands are identical to the REDUCE **OUT** and **SHUT** commands with the following exceptions:

- **GENTRANOUT** and **GENTRANSHUT** redirect *only* code which is printed as a side effect of GENTRAN commands.
- **GENTRANOUT** allows more than one file name to be given to indicate that generated code is to be sent to two or more files. (It is particularly convenient to be able to have generated code sent to the terminal screen and one or more file simultaneously.)
- **GENTRANOUT** does not automatically erase existing files; it prints a warning message on the terminal and asks the user whether the existing file should be erased or the whole command be aborted.

Chapter 43

GEOMETRY: Mechanized (Plane) Geometry Manipulations

Hans-Gert Gräbe
Universität Leipzig, Germany
e-mail: graebe@informatik.uni-leipzig.de

43.1 Introduction

This package provides tools for formulation and mechanized proofs of geometry statements in the spirit of the "Chinese Prover" of W.-T. Wu [23] and the fundamental book [4] of S.-C. Chou who proved 512 geometry theorems with this mechanized method, see also [3], [5], [21], [22].

The general idea behind this approach is an algebraic reformulation of geometric conditions using generic coordinates. A (mathematically strong) proof of the geometry statement then may be obtained from appropriate manipulations of these algebraic expressions. A CAS as, e.g., Reduce is well suited to mechanize these manipulations.

For a more detailed introduction to the topic see the accompanying file `geometry.tex` in $REDUCEPATH/packages/geometry/.

43.2 Basic Data Types and Constructors

The basic data types in this package are `Scalar, Point, Line, Circle1 and Circle`.
The function `POINT`(a,b) creates a `Point` in the plane with the (x,y)-coordinates (a,b). A `Line` is created with the function `LINE`(a,b,c) and fulfills the equation $ax+by+c=0$. For circles there are two constructors. You can use `CIRCLE`(c_1,c_2,c_3,c_4) to create a `Circle` where the scalar variables solve the equation $c_1(x^2+y^2)+c_2x+c_3y+c_4=0$. Note that lines are a subset of the circles with $c_1=0$. The other way to create a `Circle` is the function `CIRCLE1`(M,s). The variable M here denotes a `Point` and s the squared radius. Please note that this package mostly uses the squared distances and radiuses.

There are various functions whose return type is `Scalar`. Booleans are represented as extended booleans, i.e. the procedure returns a `Scalar` that is zero iff the condition is fulfilled. For example, the function call `POINT_ON_CIRCLE(P,c)` returns zero if the `Point` P is on the circle, otherwise P is not on the circle. In some cases also a non zero result has a geometric meaning. For example, `COLLINEAR(A,B,C)` returns the signed area of the corresponding parallelogram.

43.3 Procedures

This section contains a short description of all procedures available in GEOMETRY. Per convention distances and radiuses of circles are squared.

`ANGLE_SUM(a,b:Scalar):Scalar`
Returns $\tan(\alpha+\beta)$, if $a=\tan(\alpha), b=\tan(\beta)$.

`ALTITUDE(A,B,C:Point):Line`
The altitude from A onto $g(BC)$.

`C1_CIRCLE(M:Point,sqr:Scalar):Circle`
The circle with given center and sqradius.

`CC_TANGENT(c1,c2:Circle):Scalar`
Zero iff c_1 and c_2 are tangent.

`CHOOSE_PC(M:Point,r,u):Point`

Chooses a point on the circle around M with radius r using its rational parametrization with parameter u.

`CHOOSE_PL(a:Line,u):Point`
Chooses a point on a using parameter u.

`CIRCLE(c1,c2,c3,c4:Scalar):Circle`
The `Circle` constructor.

`CIRCLE1(M:Point,sqr:Scalar):Circle1`
The `Circle1` constructor.

`CIRCLE_CENTER(c:Circle):Point`
The center of c.

`CIRCLE_SQRADIUS(c:Circle):Scalar`
The sqradius of c.

`CL_TANGENT(c:Circle,l:Line):Scalar`
Zero iff l is tangent to c.

`COLLINEAR(A,B,C:Point):Scalar`
Zero iff A, B, C are on a common line. In general the signed area of the parallelogram spanned by $\vec{AB}$ and $\vec{AC}$.

`CONCURRENT(a,b,c:Line):Scalar`
Zero iff a, b, c have a common point.

`INTERSECTION_POINT(a,b:Line):Point`
The intersection point of the lines a, b.

`L2_ANGLE(a,b:Line):Scalar`
Tangens of the angle between a and b.

`LINE(a,b,c:Scalar):Line`
The `Line` constructor.

`LOT(P:Point,a:Line):Line`
The perpendicular from P onto a.

`MEDIAN(A,B,C:Point):Line`
The median line from A to BC.

`MIDPOINT(A,B:Point):Point`
The midpoint of AB.

`MP(B,C:Point):Line`

The midpoint perpendicular of BC.

`ORTHOGONAL(a,b:Line):Scalar`
zero iff the lines a, b are orthogonal.

`OTHER_CC_POINT(P:Point,c1,c2:Circle):Point`
c_1 and c_2 intersect at P. The procedure returns the second intersection point.

`OTHER_CL_POINT(P:Point,c:Circle,l:Line):Point`
c and l intersect at P. The procedure returns the second intersection point.

`P3_ANGLE(A,B,C:Point):Scalar`
Tangens of the angle between $\vec{BA}$ and $\vec{BC}$.

`P3_CIRCLE(A,B,C:Point):Circle` or
`P3_CIRCLE1(A,B,C:Point):Circle1`
The circle through 3 given points.

`P4_CIRCLE(A,B,C,D:Point):Scalar`
Zero iff four given points are on a common circle.

`PAR(P:Point,a:Line):Line`
The line through P parallel to a.

`PARALLEL(a,b:Line):Scalar`
Zero iff the lines a, b are parallel.

`PEDALPOINT(P:Point,a:Line):Point`
The pedal point of the perpendicular from P onto a.

`POINT(a,b:Scalar):Point`
The `Point` constructor.

`POINT_ON_BISECTOR(P,A,B,C:Point):Scalar`
Zero iff P is a point on the (inner or outer) bisector of the angle $\angle ABC$.

`POINT_ON_CIRCLE(P:Point,c:Circle):Scalar` or
`POINT_ON_CIRCLE1(P:Point,c:Circle1):Scalar`
Zero iff P is on the circle c.

`POINT_ON_LINE(P:Point,a:Line):Scalar`
Zero iff P is on the line a.

`PP_LINE(A,B:Point):Line`
The line through A and B.

`SQRDIST(A,B:Point):Scalar`
    Square of the distance between A and B.

`SYMPOINT(P:Point,l:Line):Point`
    The point symmetric to P wrt. the line l.

`SYMLINE(a:Line,l:Line):Line`
    The line symmetric to a wrt. the line l.

`VARPOINT(A,B:Point,u):Point`
    The point $D = u \cdot A + (1 - u) \cdot B$.

GEOMETRY supplies as additional tools the functions

`EXTRACTMAT(polys,vars)`
    Returns the coefficient matrix of the list of equations *polys* that are linear in the variables *vars*.

`RED_HOM_COORDS(u:{Line,Circle})`
    Returns the reduced homogeneous coordinates of u, i.e., divides out the content.

43.4 Examples

Example 17

Create three points as the vertices of a generic triangle.
`A:=Point(a1,a2); B:=Point(b1,b2); C:=Point(c1,c2);`

The midpoint perpendiculars of $\Delta\,ABC$ pass through a common point since

```
concurrent(mp(A,B),mp(B,C),mp(C,A));
```

simplifies to zero.

Example 18

The intersection point of the midpoint perpendiculars

```
M:=intersection_point(mp(A,B),mp(B,C));
```

is the center of the circumscribed circle since

```
sqrdist(M,A) - sqrdist(M,B);
```

simplifies to zero.

Example 19

Euler's line:

> The center M of the circumscribed circle, the orthocenter H and the barycenter S are collinear and S divides MH with ratio 1:2.

Compute the coordinates of the corresponding points

```
M:=intersection_point(mp(a,b,c),mp(b,c,a));
H:=intersection_point(altitude(a,b,c),altitude(b,c,a));
S:=intersection_point(median(a,b,c),median(b,c,a));
```

and then prove that

```
collinear(M,H,S);
sqrdist(S,varpoint(M,H,2/3));
```

both simplify to zero.

Chapter 44

GNUPLOT: Display of functions and surfaces

Herbert Melenk
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany
e–mail: melenk@zib.de

The **gnuplot** system provides easy to use graphics output for curves or surfaces which are defined by formulas and/or data sets. The REDUCE GNUPLOT package lets one use the GNUPLOT graphical output directly from inside REDUCE, either for the interactive display of curves/surfaces or for the production of pictures on paper.

For a full understanding of use of the REDUCE GNUPLOT package it is best to be familiar with **gnuplot**.

The main command is `PLOT`. It accepts an arbitrary list of arguments which are either an expression to be plotted, a range expressions or an option.

```
load_package gnuplot;
plot(w=sin(a),a=(0 .. 10),xlabel="angle",ylabel="sine");
```

The expression can be in one or two unknowns, or a list of two functions for the x and y values. It can also be an implicit equation in 2-dimensional space.

```
plot(x**3+x*y**3-9x=0);
```

The dependent and independent variables can be limited to a range with the syntax shown in the first example. If omitted the independent variables range from -10 to 10 and the dependent variable is limited only by the precision of the IEEE floating point arithmetic.

There are a great deal of options, either as keywords or as `variable=string`. Options include:

`title`: assign a heading (default: empty)

`xlabel`: set label for the x axis

`ylabel`: set label for the y axis

`zlabel`: set label for the z axis

`terminal`: select an output device

`size`: rescale the picture

`view`: set a viewpoint

`(no)contour`: 3d: add contour lines

`(no)surface`: 3d: draw surface (default: yes)

`(no)hidden3d`: 3d: remove hidden lines (default: no)

The command `PLOTRESET` closes the current GNUPLOT windows. The next call to `PLOT` will create a new one.

GNUPLOT is controlled by a number of switches.

Normally all intermediate data sets are deleted after terminating a plot session. If the switch `PLOTKEEP` is set on, the data sets are kept for eventual post processing independent of REDUCE.

In general `PLOT` tries to generate smooth pictures by evaluating the functions at interior points until the distances are fine enough. This can require a lot of computing time if the single function evaluation is expensive. The refinement is controlled by the switch `PLOTREFINE` which is on by default. When you turn it off the functions will be evaluated only at the basic points.

The integer value of the global variable `PLOT_XMESH` defines the number of initial function evaluations in x direction for `PLOT`. For 2d graphs additional points will be used as long as `plotrefine` is on. For 3d graphs this number defines also the number of mesh lines orthogonal to the x axis. PLOT_YMESH defines for 3d plots the number of function evaluations in the y direction and the number of mesh lines orthogonal to the y axis.

The grid for localising an implicitly defined curve in PLOT consists of triangles. These are computed initially equally distributed over the x-y plane controlled by PLOT_XMESH. The grid is refined adaptively in several levels. The final grid can be visualised by setting on the switch SHOW_GRID.

Chapter 45

GROEBNER: A Gröbner basis package

Herbert Melenk & Winfried Neun
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany
e–mail: melenk@zib.de
and
H.M. Möller
Fernuniversität Hagen FB Math und Informatik
Postfach 940
D–58084 Hagen, Germany
e–mail: Michael.Moeller@fernuni-hagen.de

Gröbner bases are a valuable tool for solving problems in connection with multivariate polynomials, such as solving systems of algebraic equations and analysing polynomial ideals.

The GROEBNER package calculates Gröbner bases using the Buchberger algorithm. It can be used over a variety of different coefficient domains, and for different variable and term orderings.

45.1

45.1.1 Term Ordering

In the theory of Gröbner bases, the terms of polynomials are considered as ordered. Several order modes are available in the current package, including the basic modes:

LEX, GRADLEX, REVGRADLEX

All orderings are based on an ordering among the variables. For each pair of variables (a, b) an order relation must be defined, *e.g.* "$a \gg b$". The greater sign $\gg$ does not represent a numerical relation among the variables; it can be interpreted only in terms of formula representation: "a" will be placed in front of "b" or "a" is more complicated than "b".

The sequence of variables constitutes this order base. So the notion of

$$\{x1, x2, x3\}$$

as a list of variables at the same time means

$$x1 \gg x2 \gg x3$$

with respect to the term order.

If terms (products of powers of variables) are compared with LEX, that term is chosen which has a greater variable or a higher degree if the greatest variable is the first in both. With GRADLEX the sum of all exponents (the total degree) is compared first, and if that does not lead to a decision, the LEX method is taken for the final decision. The REVGRADLEX method also compares the total degree first, but afterward it uses the LEX method in the reverse direction; this is the method originally used by Buchberger. Note that the LEX ordering is identical to the standard REDUCE kernel ordering, when KORDER is set explicitly to the sequence of variables.

LEX is the default term order mode in the GROEBNER package.

45.2 The Basic Operators

45.2.1 Term Ordering Mode

TORDER $(vl,m,[p_1, p_2, \ldots]);$

where vl is a variable list (or the empty list if no variables are declared explicitly), m is the name of a term ordering mode LEX, GRADLEX, REVGRADLEX (or another implemented mode) and $[p_1, p_2, \ldots]$ are additional parameters for the term ordering mode (not needed for the basic modes).

TORDER sets variable set and the term ordering mode. The default mode is LEX. The previous description is returned as a list with corresponding elements. Such a list can alternatively passed as sole argument to TORDER.

If the variable list is empty or if the TORDER declaration is omitted, the automatic variable extraction is activated.

GVARS ($\{exp1,\ exp2,\ \ldots,\ expn\}$);

where $\{exp1, exp2, \ldots, expn\}$ is a list of expressions or equations.

GVARS extracts from the expressions $\{exp1, exp2, \ldots, expn\}$ the kernels, which can play the role of variables for a Gröbner calculation. This can be used *e.g.* in a TORDER declaration.

45.2.2 GROEBNER: Calculation of a Gröbner Basis

GROEBNER $\{exp1, exp2, \ldots, expm\}$;

where $\{exp1, exp2, \ldots, expm\}$ is a list of expressions or equations.

GROEBNER calculates the Gröbner basis of the given set of expressions with respect to the current TORDER setting.

The Gröbner basis $\{1\}$ means that the ideal generated by the input polynomials is the whole polynomial ring, or equivalently, that the input polynomials have no zeros in common.

As a side effect, the sequence of variables is stored as a REDUCE list in the shared variable `gvarslast`.

Example 20

```
torder({},lex)$
groebner{3*x**2*y + 2*x*y + y + 9*x**2 + 5*x - 3,
2*x**3*y - x*y - y + 6*x**3 - 2*x**2 - 3*x + 3,
x**3*y + x**2*y + 3*x**3 + 2*x**2 };

              2
```

```
{8*X - 2*Y  + 5*Y + 3,

    3      2
 2*Y  - 3*Y  - 16*Y + 21}
```

The operation of GROEBNER can be controlled by the following switches:

GROEBOPT – If set ON, the sequence of variables is optimized with respect to execution speed; note that the final list of variables is available in GVARSLAST.

An explicitly declared dependency supersedes the variable optimization. By default GROEBOPT is off, conserving the original variable sequence.

GROEBFULLREDUCTION – If set off, the reduction steps during the GROEBNER operation are limited to the pure head term reduction; subsequent terms are reduced otherwise. By default GROEBFULLREDUCTION is on.

GLTBASIS – If set on, the leading terms of the result basis are extracted. They are collected in a basis of monomials, which is available as value of the global variable with the name GLTB.

45.2.3 GZERODIM?: Test of $\dim = 0$

GZERODIM!? *bas*

where *bas* is a Gröbner basis in the current setting. The result is *NIL*, if *bas* is the basis of an ideal of polynomials with more than finitely many common zeros. If the ideal is zero dimensional, *i.e.* the polynomials of the ideal have only finitely many zeros in common, the result is an integer k which is the number of these common zeros (counted with multiplicities).

45.2.4 GDIMENSION, GINDEPENDENT_SETS

The following operators can be used to compute the dimension and the independent variable sets of an ideal which has the Gröbner basis *bas* with arbitrary term order:

Gdimension *bas*

Gindependent_sets *bas Gindependent_sets* computes the maximal left independent variable sets of the ideal, that are the variable sets which play the role of free parameters in the current ideal basis. Each set is a list which is a subset of the variable list. The result is a list of these sets. For an ideal with dimension zero the list is empty. *GDimension* computes the dimension of the ideal, which is the maximum length of the independent sets.

45.2.5 GLEXCONVERT: Conversion to a Lexical Base

$GLEXCONVERT\ (\{exp,\ldots,expm\}\,[,\{var1\ldots,varn\}]$
$[,MAXDEG=mx]\,[,NEWVARS=\{nv1,\ldots,nvk\}])$
where $\{exp1,\ldots,expm\}$ is a Gröbner basis with $\{var1,\ldots,varn\}$ as variables in the current term order mode, mx is an integer, and $\{nv1,\ldots,nvk\}$ is a subset of the basis variables. For this operator the source and target variable sets must be specified explicitly.

GLEXCONVERT converts a basis of a zero-dimensional ideal (finite number of isolated solutions) from arbitrary ordering into a basis under *lex* ordering. During the call of GLEXCONVERT the original ordering of the input basis must be still active.

NEWVARS defines the new variable sequence. If omitted, the original variable sequence is used. If only a subset of variables is specified here, the partial ideal basis is evaluated. For the calculation of a univariate polynomial, NEWVARS should be a list with one element.

MAXDEG is an upper limit for the degrees. The algorithm stops with an error message, if this limit is reached.

A warning occurs if the ideal is not zero dimensional.

GLEXCONVERT is an implementation of the FLGM algorithm. Often, the calculation of a Gröbner basis with a graded ordering and subsequent conversion to *lex* is faster than a direct *lex* calculation. Additionally, GLEXCONVERT can be used to transform a *lex* basis into one with different variable sequence, and it supports the calculation of a univariate polynomial. If the latter exists, the algorithm is even applicable in the non zero-dimensional case, if such a polynomial exists.

```
torder({{w,p,z,t,s,b},gradlex)
```

```
g  :=  groebner  { f1 := 45*p + 35*s -165*b -36,
      35*p + 40*z + 25*t - 27*s, 15*w + 25*p*s +30*z -18*t
     -165*b**2, -9*w + 15*p*t  + 20*z*s,
     w*p + 2*z*t - 11*b**3, 99*w - 11*s*b +3*b**2,
     b**2 + 33/50*b + 2673/10000};

G := {60000*W + 9500*B + 3969,

    1800*P - 3100*B - 1377,

    18000*Z + 24500*B + 10287,

    750*T - 1850*B + 81,

    200*S - 500*B - 9,
          2
    10000*B  + 6600*B + 2673}

 glexconvert(g,{w,p,z,t,s,b},maxdeg=5,newvars={w});

            2
  100000000*W  + 2780000*W + 416421

 glexconvert(g,{w,p,z,t,s,b},maxdeg=5,newvars={p});

        2
  6000*P  - 2360*P + 3051
```

45.2.6 GROEBNERF: Factorizing Gröbner Bases

If Gröbner bases are computed in order to solve systems of equations or to find the common roots of systems of polynomials, the factorizing version of the Buchberger algorithm can be used. The theoretical background is simple: if a polynomial p can be represented as a product of two (or more) polynomials, *e.g.* $h = f * g$, then h vanishes if and only if one of the factors vanishes. So if during the calculation of a Gröbner basis h of the above form is detected, the whole problem can be split into two (or more) disjoint branches. Each of the branches is simpler than the complete problem; this saves computing time and space. The result of this type of computation is a list of (partial) Gröbner bases; the solution set of the original problem is the union of the solutions of the partial problems, ignoring the multiplicity of an individual solution. If a branch results in a basis $\{1\}$, then there is no common zero, *i.e.* no additional solution for the original problem, contributed by this branch.

GROEBNERF Call

The syntax of GROEBNERF is the same as for GROEBNER.

$$GROEBNERF(\{exp1, exp2, \ldots, expm\}[,\{\},\{nz1, \ldots nzk\});$$

where $\{exp1, exp2, \ldots, expm\}$ is a given list of expressions or equations, and $\{nz1, \ldots nzk\}$ is an optional list of polynomials known to be non-zero.

GROEBNERF tries to separate polynomials into individual factors and to branch the computation in a recursive manner (factorisation tree). The result is a list of partial Gröbner bases. If no factorisation can be found or if all branches but one lead to the trivial basis $\{1\}$, the result has only one basis; nevertheless it is a list of lists of polynomials. If no solution is found, the result will be $\{\{1\}\}$. Multiplicities (one factor with a higher power, the same partial basis twice) are deleted as early as possible in order to speed up the calculation. The factorising is controlled by some switches.

As a side effect, the sequence of variables is stored as a REDUCE list in the shared variable

gvarslast .

If GLTBASIS is on, a corresponding list of leading term bases is also produced and is available in the variable GLTB.

The third parameter of GROEBNERF allows one to declare some polynomials nonzero. If any of these is found in a branch of the calculation the branch is cancelled. This can be used to save a substantial amount of computing time. The second parameter must be included as an empty list if the third parameter is to be used.

```
torder({x,y},lex)$
groebnerf { 3*x**2*y + 2*x*y + y + 9*x**2 + 5*x = 3,
            2*x**3*y - x*y - y + 6*x**3 - 2*x**2 - 3*x = -3,
             x**3*y + x**2*y + 3*x**3 + 2*x**2 };
```

```
    {{Y - 3,X},

                          2
  {2*Y + 2*X - 1,2*X   - 5*X - 5}}
```

It is obvious here that the solutions of the equations can be read off immediately.

All switches from GROEBNER are valid for GROEBNERF as well:

GROEBOPT
GLTBASIS
GROEBFULLREDUCTION
GROEBSTAT
TRGROEB
TRGROEBS
TRGROEB1

Restriction of the Solution Space

In some applications only a subset of the complete solution set of a given set of equations is relevant, *e.g.* only nonnegative values or positive definite values for the variables. A significant amount of computing time can be saved if nonrelevant computation branches can be terminated early.

Positivity: If a polynomial has no (strictly) positive zero, then every system containing it has no nonnegative or strictly positive solution. Therefore, the Buchberger algorithm tests the coefficients of the polynomials for equal sign if requested. For example, in $13 * x + 15 * y * z$ can be zero with real nonnegative values for x, y and z only if $x = 0$ and $y = 0$ or $z = 0$; this is a sort of "factorization by restriction". A polynomial $13 * x + 15 * y * z + 20$ never can vanish with nonnegative real variable values.

Zero point: If any polynomial in an ideal has an absolute term, the ideal cannot have the origin point as a common solution.

By setting the shared variable

GROEBRESTRICTION

GROEBNERF is informed of the type of restriction the user wants to impose on the solutions:

GROEBRESTRICTION:=NONEGATIVE;
only nonnegative real solutions are of interest

GROEBRESTRICTION:=POSITIVE;
only nonnegative and nonzero solutions are of interest

GROEBRESTRICTION:=ZEROPOINT;
only solution sets which contain the point $\{0, 0, \ldots, 0\}$ are or interest.

If GROEBNERF detects a polynomial which formally conflicts with the restriction, it either splits the calculation into separate branches, or, if a violation of the restriction is determined, it cancels the actual calculation branch.

45.2.7 GREDUCE, PREDUCE: Reduction of Polynomials

Background

Reduction of a polynomial "p" modulo a given sets of polynomials "B" is done by the reduction algorithm incorporated in the Buchberger algorithm.

Reduction via Gröbner Basis Calculation

$$GREDUCE(exp, \{exp1, exp2, \ldots, expm\}]);$$

where *exp* is an expression, and $\{exp1, exp2, \ldots, expm\}$ is a list of any number of expressions or equations.

GREDUCE first converts the list of expressions $\{exp1, \ldots, expn\}$ to a Gröbner basis, and then reduces the given expression modulo that basis. An error results if the list of expressions is inconsistent. The returned value is an expression representing the reduced polynomial. As a side effect, GREDUCE sets the variable *gvarslast* in the same manner as GROEBNER does.

Reduction with Respect to Arbitrary Polynomials

$$PREDUCE(exp, \{exp1, exp2, \ldots, expm\});$$

where exp is an expression, and $\{exp1, exp2, \ldots, expm\}$ is a list of any number of expressions or equations.

PREDUCE reduces the given expression modulo the set $\{exp1, \ldots, expm\}$. If this set is a Gröbner basis, the obtained reduced expression is uniquely determined. If not, then it depends on the subsequence of the single reduction steps (see 45.2.7). PREDUCE does not check whether $\{exp1, exp2, \ldots, expm\}$ is a Gröbner basis in the actual order. Therefore, if the expressions are a Gröbner basis calculated earlier with a variable sequence given explicitly or modified by optimisation, the proper variable sequence and term order must be activated first.

Example 21(PREDUCE called with a Gröbner basis):

```
torder({x,y},lex);
gb:=groebner{3*x**2*y + 2*x*y + y + 9*x**2 + 5*x - 3,
             2*x**3*y - x*y - y + 6*x**3 - 2*x**2 - 3*x + 3,
             x**3*y + x**2*y + 3*x**3 + 2*x**2}$
preduce (5*y**2 + 2*x**2*y + 5/2*x*y + 3/2*y
           + 8*x**2 + 3/2*x - 9/2, gb);


    2
   Y
```

45.3 Ideal Decomposition & Equation System Solving

Based on the elementary Gröbner operations, the GROEBNER package offers additional operators, which allow the decomposition of an ideal or of a system of equations down to the individual solutions. Details of the operators GROESOLVE, GROEBNERF and IDEALQUOTIENT can be found in the full documentation, with associated functions.

Chapter 46

IDEALS: Arithmetic for polynomial ideals

Herbert Melenk
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraße 7
D–14195 Berlin–Dahlem, Germany
e–mail: melenk@zib.de

This package implements the basic arithmetic for polynomial ideals by exploiting the Gröbner bases package of REDUCE. In order to save computing time all intermediate Gröbner bases are stored internally such that time consuming repetitions are inhibited. A uniform setting facilitates the access.

46.1 Initialization

Prior to any computation the set of variables has to be declared by calling the operator *I_setting* . For example in order to initiate computations in the polynomial ring $Q[x, y, z]$ call

```
I_setting(x,y,z);
```

A subsequent call to *I_setting* allows one to select another set of variables; at the same time the internal data structures are cleared in order to free memory resources.

46.2 Bases

An ideal is represented by a basis (set of polynomials) tagged with the symbol I, *e.g.*

```
u := I(x*z-y**2, x**3-y*z);
```

Alternatively a list of polynomials can be used as input basis; however, all arithmetic results will be presented in the above form. The operator *ideal2list* allows one to convert an ideal basis into a conventional REDUCE list.

46.2.1 Operators

Because of syntactical restrictions in REDUCE, special operators have to be used for ideal arithmetic:

```
.+            ideal sum (infix)
.*            ideal product (infix)
.:            ideal quotient (infix)
./            ideal quotient (infix)
.=            ideal equality test (infix)
subset        ideal inclusion test (infix)
intersection  ideal intersection (prefix,binary)
member        test for membership in an ideal
                (infix: polynomial and ideal)
gb            Groebner basis of an ideal (prefix, unary)
ideal2list    convert ideal basis to polynomial list
                (prefix,unary)
```

Example:

```
 I(x+y,x^2) .* I(x-z);

   2                       2    2
I(X  + X*Y - X*Z - Y*Z,X*Y  - Y *Z)
```

Note that ideal equality cannot be tested with the REDUCE equal sign:

```
I(x,y)  = I(y,x)       is false
I(x,y) .= I(y,x)       is true
```

Chapter 47

INEQ: Support for solving inequalities

Herbert Melenk
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany
e–mail: melenk@zib.de

This package supports the operator **ineq_solve** that tries to solves single inequalities and sets of coupled inequalities. The following types of systems are supported [1]:

- only numeric coefficients (no parametric system),
- a linear system of mixed equations and <= – >= inequalities, applying the method of Fourier and Motzkin,
- a univariate inequality with <=, >=, > or < operator and polynomial or rational left–hand and right–hand sides, or a system of such inequalities with only one variable.

Syntax:

```
INEQ_SOLVE(<expr> [,<vl>])
```

[1] For linear optimization problems please use the operator **simplex** of the **linalg** package (section 52.5

where <expr> is an inequality or a list of coupled inequalities and equations, and the optional argument <vl> is a single variable (kernel) or a list of variables (kernels). If not specified, they are extracted automatically from <expr>. For multivariate input an explicit variable list specifies the elimination sequence: the last member is the most specific one.

An error message occurs if the input cannot be processed by the current algorithms.

The result is a list. It is empty if the system has no feasible solution. Otherwise the result presents the admissible ranges as set of equations where each variable is equated to one expression or to an interval. The most specific variable is the first one in the result list and each form contains only preceding variables (resolved form). The interval limits can be formal **max** or **min** expressions. Algebraic numbers are encoded as rounded number approximations.

Examples:

```
ineq_solve({(2*x^2+x-1)/(x-1) >=  (x+1/2)^2, x>0});

{x=(0 .. 0.326583),x=(1 .. 2.56777)}

 reg:=
 {a + b - c>=0, a - b + c>=0, - a + b + c>=0, 0>=0, 2>=0,
  2*c - 2>=0, a - b + c>=0, a + b - c>=0, - a + b + c - 2>=0,
  2>=0, 0>=0, 2*b - 2>=0, k + 1>=0, - a - b - c + k>=0,
   - a - b - c + k + 2>=0, - 2*b + k>=0,
   - 2*c + k>=0, a + b + c - k>=0,
  2*b + 2*c - k - 2>=0, a + b + c - k>=0}$

ineq_solve (reg,{k,a,b,c});

{c=(1 .. infinity),

 b=(1 .. infinity),

 a=(max( - b + c,b - c) .. b + c - 2),

 k=a + b + c}
```

Chapter 48

INVBASE: A package for computing involutive bases

A.Yu.Zharkov, Yu.A.Blinkov
Saratov University
Astrakhanskaya 83
410071 Saratov, Russia
e–mail: postmaster@scnit.saratov.su

Involutive bases are a new tool for solving problems in connection with multivariate polynomials, such as solving systems of polynomial equations and analysing polynomial ideals. An involutive basis of polynomial ideal is a special form of a redundant Gröbner basis. The construction of involutive bases reduces the problem of solving polynomial systems to simple linear algebra.

The INVBASE package can be seen as an alternative to Buchberger's algorithm.

48.1 The Basic Operators

48.1.1 Term Ordering

The term order modes available are `REVGRADLEX`, `GRADLEX` and `LEX`. These modes have the same meaning as for the GROEBNER package.

All orderings are based on an ordering among the variables. For each pair of variables an order relation $\gg$ must be defined. The term ordering mode as well as the order of variables are set by the operator `INVTORDER` *mode*,$\{x_1, ..., x_n\}$ where *mode* is one of the term order modes listed above. The notion of $\{x_1, ..., x_n\}$ as a list of variables at the same time means $x_1 \gg \ldots \gg x_n$.

48.1.2 Computing Involutive Bases

To compute the involutive basis of ideal generated by the set of polynomials $\{p_1, ..., p_m\}$ one should type the command

`INVBASE` $\{p_1, ..., p_m\}$

where p_i are polynomials in variables listed in the `INVTORDER` operator. If some kernels in p_i were not listed previously in the `INVTORDER` operator they are considered as parameters, *i.e.* they are considered part of the coefficients of polynomials. If `INVTORDER` was omitted, all the kernels in p_i are considered as variables with the default REDUCE kernel order.

The coefficients of polynomials p_i may be integers as well as rational numbers (or, accordingly, polynomials and rational functions in the parametric case). The computations modulo prime numbers are also available. For this purpose one should type the REDUCE commands

```
ON MODULAR; SETMOD p;
```

where p is a prime number. The value of the `INVBASE` function is a list of integer polynomials $\{g_1, ..., g_n\}$ representing an involutive basis of a given ideal.

```
INVTORDER REVGRADLEX, {x,y,z};

g:= INVBASE {4*x**2 + x*y**2 - z + 1/4,
             2*x + y**2*z + 1/2,
             x**2*z - 1/2*x - y**2};

             3          2      3        2
g := {8*x*y*z  - 2*x*y*z  + 4*y  - 4*y*z  + 16*x*y + 17*y*z - 4*y,

         4         2        2                2
      8*y  - 8*x*z  - 256*y  + 2*x*z + 64*z  - 96*x + 20*z - 9,
```

```
   3
2*y *z + 4*x*y + y,

     3        2      2      2
8*x*z  - 2*x*z  + 4*y  - 4*z  + 16*x + 17*z - 4,

        3      3                2
 - 4*y*z  - 8*y  + 6*x*y*z + y*z  - 36*x*y - 8*y,

     2       2      2
4*x*y  + 32*y  - 8*z  + 12*x - 2*z + 1,

   2
2*y *z + 4*x + 1,

      3      2            2
 - 4*z  - 8*y  + 6*x*z + z  - 36*x - 8,

   2       2      2
8*x  - 16*y  + 4*z  - 6*x - z}
```

To convert it into a lexicographical Gröbner basis one should type

```
h := INVLEX g;

                      6        5         4           3
h := {3976*x + 37104*z  - 600*z  + 2111*z  + 122062*z

                2
       + 232833*z  - 680336*z + 288814,

          2           6          5         4           3
      1988*y  - 76752*z  + 1272*z  - 4197*z  - 251555*z

                2
       - 481837*z  + 1407741*z - 595666,

         7      6    5       4       3        2
      16*z  - 8*z  + z  + 52*z  + 75*z  - 342*z  + 266*z

       - 60}
```

Chapter 49

LAPLACE: Laplace and inverse Laplace transforms

C. Kazasov, M. Spiridonova, V. Tomov
Sofia, Bulgaria

The LAPLACE package provides both Laplace Transforms and Inverse Laplace Transforms, with the two operators

```
LAPLACE(exp, s_var, t_var)
INVLAP(exp, s_var, t_var)
```

The action is to transform the expression from the `s_var` or source variable into the `t_var` or target variable. If `t_var` is omitted, the package uses an internal variable `lp!&` or `il!&` respectively.

Three switches control the transformations. If `lmon` is on then sine, cosine, hyperbolic sine and hyperbolic cosines are converted by LAPLACE into exponentials. If `lhyp` is on then exponential functions are converted into hyperbolic form. The last switch `ltrig` has the same effect except it uses trigonometric functions.

The system can be extended by adding Laplace transformation rules for single functions by rules or rule sets. In such a rule the source variable **must** be free, the target variable **must** be `il!&` for LAPLACE and `lp!&` for INVLAP, with the third parameter omitted. Also rules for transforming derivatives are entered in such a form. For example

```
let {laplace(log(~x),x) => -log(gam * il!&)/il!&,
```

```
          invlap(log(gam * ~x)/x,x) => -log(lp!&)};
    operator f;
    let {
      laplace(df(f(~x),x),x) => il!&*laplace(f(x),x) - sub(x=0,f(x)),

      laplace(df(f(~x),x,~n),x) => il!&**n*laplace(f(x),x) -
        for i:=n-1 step -1 until 0 sum
          sub(x=0, df(f(x),x,n-1-i)) * il!&**i
            when fixp n,

      laplace(f(~x),x) = f(il!&)
    };
```

The LAPLACE system knows about the functions `DELTA` and `GAMMA`, and used the operator `ONE` for the unit step function and `INTL` stands for the parameterised integral function, for instance `intl(2*y**2,y,0,x)` stands for $\int_0^x 2y^2 dx$.

```
load_package laplace;

laplace(sin(17*x),x,p);

    17
----------
  2
 p  + 289

on lmon;

laplace(-1/4*e**(a*x)*(x-k)**(-1/2), x, p);

     1            a*k
  - ---*sqrt(pi)*e
     4
----------------------
  k*p
 e   *sqrt( - a + p)

invlap(c/((p-a)*(p-b)), p, t);

     a*t    b*t
 c*(e    - e   )
-----------------
      a - b
```

```
invlap(p**(-7/3), p, t);

    1/3
   t   *t
------------
        7
 gamma(---)
        3
```

Chapter 50

LIE: Functions for the classification of real n-dimensional Lie algebras

Carsten and Franziska Schöbel
The Leipzig University, Computer Science Department
Augustusplatz 10/11,
O-7010 Leipzig, Germany
e–mail: cschoeb@aix550.informatik.uni-leipzig.de

LIE is a package of functions for the classification of real n-dimensional Lie algebras. It consists of two modules: **liendmc1** and **lie1234**.

50.1 liendmc1

With the help of the functions in this module real n-dimensional Lie algebras L with a derived algebra $L^{(1)}$ of dimension 1 can be classified. L has to be defined by its structure constants c_{ij}^k in the basis $\{X_1, \ldots, X_n\}$ with $[X_i, X_j] = c_{ij}^k X_k$. The user must define an ARRAY LIENSTRUCIN(n, n, n) with n being the dimension of the Lie algebra L. The structure constants LIENSTRUCIN(i, j, k):=c_{ij}^k for $i < j$ should be given. Then the procedure LIENDIMCOM1 can be called. Its syntax is:

```
LIENDIMCOM1(<number>).
```

`<number>` corresponds to the dimension n. The procedure simplifies the structure of L performing real linear transformations. The returned value is a list of the form

```
(i) {LIE_ALGEBRA(2),COMMUTATIVE(n-2)} or
(ii) {HEISENBERG(k),COMMUTATIVE(n-k)}
```

with $3 \leq k \leq n$, k odd.

The returned list is also stored as`LIE_LIST`. The matrix LIENTRANS gives the transformation from the given basis $\{X_1, \ldots, X_n\}$ into the standard basis $\{Y_1, \ldots, Y_n\}$: $Y_j = (\text{LIENTRANS})_j^k X_k$.

50.2 lie1234

This part of the package classifies real low-dimensional Lie algebras L of the dimension $n := \dim L = 1, 2, 3, 4$. L is also given by its structure constants c_{ij}^k in the basis $\{X_1, \ldots, X_n\}$ with $[X_i, X_j] = c_{ij}^k X_k$. An ARRAY LIESTRIN$(n, n, n)$ has to be defined and LIESTRIN(i, j, k):=c_{ij}^k for $i < j$ should be given. Then the procedure LIECLASS can be performed whose syntax is:

```
LIECLASS(<number>).
```

`<number>` should be the dimension of the Lie algebra L. The procedure stepwise simplifies the commutator relations of L using properties of invariance like the dimension of the centre, of the derived algebra, unimodularity *etc.* The returned value has the form:

```
{LIEALG(n),COMTAB(m)},
```

where the value m corresponds to the number of the standard form (basis: $\{Y_1, \ldots, Y_n\}$) in an enumeration scheme.

This returned value is also stored as LIE_CLASS. The linear transformation from the basis $\{X_1, \ldots, X_n\}$ into the basis of the standard form $\{Y_1, \ldots, Y_n\}$ is given by the matrix LIEMAT: $Y_j = (\text{LIEMAT})_j^k X_k$.

Chapter 51

LIMITS: A package for finding limits

Stanley L. Kameny
Los Angeles, U.S.A.

LIMITS is a fast limit package for REDUCE for functions which are continuous except for computable poles and singularities, based on some earlier work by Ian Cohen and John P. Fitch. The Truncated Power Series package is used for non-critical points, at which the value of the function is the constant term in the expansion around that point. l'Hôpital's rule is used in critical cases, with preprocessing of $\infty - \infty$ forms and reformatting of product forms in order to apply l'Hôpital's rule. A limited amount of bounded arithmetic is also employed where applicable.

51.1 Normal entry points

`LIMIT`(EXPRN:*algebraic*, VAR:*kernel*, LIMPOINT:*algebraic*):*algebraic*

This is the standard way of calling limit, applying all of the methods. The result is the limit of EXPRN as VAR approaches LIMPOINT.

51.2 Direction-dependent limits

`LIMIT!+`(EXPRN:*algebraic*, VAR:*kernel*, LIMPOINT:*algebraic*):*algebraic*
`LIMIT!-`(EXPRN:*algebraic*, VAR:*kernel*, LIMPOINT:*algebraic*):*algebraic*

If the limit depends upon the direction of approach to the `LIMPOINT`, the functions `LIMIT!+` and `LIMIT!-` may be used. They are defined by:

`LIMIT!+` (EXP,VAR,LIMPOINT) $\rightarrow$ `LIMIT`(EXP*,ϵ,0)
where EXP* = sub(VAR=VAR+ϵ^2,EXP)

and

`LIMIT!-` (EXP,VAR,LIMPOINT) $\rightarrow$ `LIMIT`(EXP*,ϵ,0)
where EXP* = sub(VAR=VAR-ϵ^2,EXP)

Examples:

```
load_package misc;

limit(sin(x)/x,x,0);

1

limit((a^x-b^x)/x,x,0);

log(a) - log(b)

limit(x/(e**x-1), x, 0);

1

limit!-(sin x/cos x,x,pi/2);

infinity

limit!+(sin x/cos x,x,pi/2);

 - infinity

limit(x^log(1/x),x,infinity);

0
```

```
limit((x^(1/5) + 3*x^(1/4))^2/(7*(sqrt(x + 9) - 3 - x/6))^(1/5),x,0);

     3/5
  - 6
---------
   1/5
  7
```

Chapter 52

LINALG: Linear algebra package

Matt Rebbeck
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraŝe 7
D–14195 Berlin–Dahlem, Germany

52.1 Introduction

This package provides a selection of functions that are useful in the world of linear algebra. They can be classified into four sections:

52.1.1 Basic matrix handling

add_columns	add_rows	add_to_columns	add_to_rows
augment_columns	char_poly	column_dim	copy_into
diagonal	extend	find_companion	get_columns
get_rows	hermitian_tp	matrix_augment	matrix_stack
minor	mult_columns	mult_rows	pivot
remove_columns	remove_rows	row_dim	rows_pivot
stack_rows	sub_matrix	swap_columns	swap_entries
swap_rows			

52.1.2 Constructors

Functions that create matrices.

band_matrix	block_matrix	char_matrix	coeff_matrix
companion	hessian	hilbert	jacobian
jordan_block	make_identity	random_matrix	toeplitz
vandermonde	Kronecker_Product		

52.1.3 High level algorithms

char_poly	cholesky	gram_schmidt	lu_decom
pseudo_inverse	simplex	svd	triang_adjoint

There is a separate NORMFORM package (chapter 57) for computing the matrix normal forms smithex, smithex_int, frobenius, ratjordan, jordansymbolic and jordan in REDUCE.

52.1.4 Predicates

matrixp squarep symmetricp

52.2 Explanations

In the examples the matrix $\mathcal{A}$ will be

$$\mathcal{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Throughout $\mathcal{I}$ is used to indicate the identity matrix and $\mathcal{A}^T$ to indicate the transpose of the matrix $\mathcal{A}$.

Many of the functions have a fairly obvious meaning. Others need a little explanation.

52.3 Basic matrix handling

The functions `ADD_COLUMNS` and `ADD_ROWS` provide basic operations between rows and columns. The form is

`add_columns(`$\mathcal{A}$`,c1,c2,expr);`

and it replaces column c2 of the matix by expr $*$ column($\mathcal{A}$,c1) $+$ column($\mathcal{A}$,c2).

`ADD_TO_COLUMNS` and `ADD_TO_ROWS` do a similar task, adding an expression to each of a number of columns (or rows) specified by a list.

$$\texttt{add\_to\_columns}(\mathcal{A},\{1,2\},10) \quad = \quad \begin{pmatrix} 11 & 12 & 3 \\ 14 & 15 & 6 \\ 17 & 18 & 9 \end{pmatrix}$$

The functions `MULT_COLUMNS` and `MULT_ROW` are equivalent to multiply columns and rows.

`COLUMN_DIM` and `ROW_DIM` find the column dimension and row dimension of their argument.

Parts of a matrix can be replaced from another by using `COPY_INTO`; the last two arguments are row and column counters for to where to copy the matrix.

$$\mathcal{G} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\texttt{copy\_into}(\mathcal{A},\mathcal{G},1,2) \quad = \quad \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 4 & 5 & 6 \\ 0 & 7 & 8 & 9 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

A diagonal matrix can be created with `DIAGONAL`. The argument is a list of expressions of matrices which form the diagonal.

An existing matrix can be extended; the call `EXTEND`(A,r,c,exp) returns the matrix A extended by r rows and c columns, with the new entries all exp.

The function `GET_COLUMNS` extracts from a matrix a list of the specified columns as matrices. `GET_ROWS` does the equivalent for rows.

$$\texttt{get\_columns}(\mathcal{A},\{1,3\}) \quad = \quad \left\{ \left(\begin{array}{c} 1 \\ 4 \\ 7 \end{array} \right), \left(\begin{array}{c} 3 \\ 6 \\ 9 \end{array} \right) \right\}$$

The Hermitian transpose, that is a matrix in which the (i,j) entry is the conjugate of the (j, i) entry of the input is returned by `HERMITIAN_TP`.

`MATRIX_AUGMENT`($\{\text{mat}_1, \text{mat}_2, \ldots, \text{mat}_n\}$) produces a new matrix from the list joined as new columns. `MATRIX_STACK` joins a list of matrices by stacking them.

$$\texttt{matrix\_stack}(\{\mathcal{A},\mathcal{A}\}) \quad = \quad \left(\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \right)$$

`MINOR`(A,r,c) calculates the (r,c) minor of A.

`PIVOT` pivots a matrix about its (r,c) entry. To do this, multiples of the r^{th} row are added to every other row in the matrix. This means that the c^{th} column will be 0 except for the (r,c) entry.

A variant on this operation is provided by `ROWS_PIVOT`. It applies the pivot only to the rows specified as the last argument.

A sub matrix can be extracted, giving a list or the rows and columns to keep.

$$\texttt{sub\_matrix}(\mathcal{A},\{1,3\},\{2,3\}) \quad = \quad \left(\begin{array}{cc} 2 & 3 \\ 8 & 9 \end{array} \right)$$

The basic operation of swapping rows or columns is provided by `SWAP_ROWS` and `SWAP_COLUMNS`. Individual entries can be swapped with `SWAP_ENTRIES`.

$$\texttt{swap\_columns}(\mathcal{A},2,3) \quad = \quad \left(\begin{array}{ccc} 1 & 3 & 2 \\ 4 & 6 & 5 \\ 7 & 9 & 8 \end{array} \right)$$

$$\texttt{swap\_entries}(\mathcal{A},\{1,1\},\{3,3\}) \quad = \quad \begin{pmatrix} 9 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 1 \end{pmatrix}$$

52.4 Constructors

`AUGMENT_COLUMNS` allows just specified columns to be selected; `STACK_ROWS` does a similar job for rows.

$$\texttt{stack\_rows}(\mathcal{A},\{1,3\}) \quad = \quad \begin{pmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \end{pmatrix}$$

Rows or columns can be removed with `REMOVE_COLUMNS` and `REMOVE_ROWS`.

$$\texttt{remove\_columns}(\mathcal{A},2) \quad = \quad \begin{pmatrix} 1 & 3 \\ 4 & 6 \\ 7 & 9 \end{pmatrix}$$

`BAND_MATRIX` creates a square matrix of dimension its second argument. The diagonal consists of the middle expressions of the first argument, which is an expression list. The expressions to the left of this fill the required number of sub_diagonals and the expressions to the right the super_diagonals.

$$\texttt{band\_matrix}(\{x,y,z\},6) \quad = \quad \begin{pmatrix} y & z & 0 & 0 & 0 & 0 \\ x & y & z & 0 & 0 & 0 \\ 0 & x & y & z & 0 & 0 \\ 0 & 0 & x & y & z & 0 \\ 0 & 0 & 0 & x & y & z \\ 0 & 0 & 0 & 0 & x & y \end{pmatrix}$$

Related to the band matrix is a block matrix, which can be created by

`BLOCK_MATRIX(r,c,matrix_list)`.

The resulting matrix consists of r by c matrices filled from the matrix_list row wise.

$$\mathcal{B} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \mathcal{C} = \begin{pmatrix} 5 \\ 5 \end{pmatrix}, \quad \mathcal{D} = \begin{pmatrix} 22 & 33 \\ 44 & 55 \end{pmatrix}$$

$$\texttt{block\_matrix}(2,3,\{\mathcal{B},\mathcal{C},\mathcal{D},\mathcal{D},\mathcal{C},\mathcal{B}\}) \quad = \quad \begin{pmatrix} 1 & 0 & 5 & 22 & 33 \\ 0 & 1 & 5 & 44 & 55 \\ 22 & 33 & 5 & 1 & 0 \\ 44 & 55 & 5 & 0 & 1 \end{pmatrix}$$

Characteristic polynomials and characteristic matrices are created by the functions CHAR_POLY and CHAR_MATRIX.

A set of linear equations can be turned into the associated coefficient matrix and vector of unknowns and the righthandside. COEFF_MATRIX returns a list $\{\mathcal{C},\mathcal{X},\mathcal{B}\}$ such that $\mathcal{C}\mathcal{X} = \mathcal{B}$.

$$\texttt{coeff\_matrix}(\{x+y+4*z=10, y+x-z=20, x+y+4\}) =$$

$$\left\{ \begin{pmatrix} 4 & 1 & 1 \\ -1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}, \begin{pmatrix} z \\ y \\ x \end{pmatrix}, \begin{pmatrix} 10 \\ 20 \\ -4 \end{pmatrix} \right\}$$

COMPANION(poly,x) creates the companion matrix $\mathcal{C}$ of a polynomial. That is the square matrix of dimension n, where n is the degree of polynomial with respect to x, and the entries of $\mathcal{C}$ are: $\mathcal{C}$(i,n) = -coeffn(poly,x,i-1) for i = 1 ... n, $\mathcal{C}$(i,i-1) = 1 for i = 2 ... n and the rest are 0.

$$\texttt{companion}(x^4+17*x^3-9*x^2+11,x) \quad = \quad \begin{pmatrix} 0 & 0 & 0 & -11 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 9 \\ 0 & 0 & 1 & -17 \end{pmatrix}$$

The polynomial associated with a companion matrix can be recovered by calling FIND_COMPANION.

HESSIAN(expr, var_list) calculates the Hessian matrix of the expressions with respect to the variables in the list, or the single variable. That is the matrix with the (i,j) element the j^{th} derivative of the expressions with respect to the i^{th} variable.

$$\texttt{hessian}(x*y*z+x^2,\{w,x,y,z\}) \quad = \quad \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & z & y \\ 0 & z & 0 & x \\ 0 & y & x & 0 \end{pmatrix}$$

Hilbert's matrix, that is where the (i, j) element is $1/(i+j-x)$ is constructed by `HILBERT`(n,x).

The Jacobian of an expression list with respect to a variable list is calculated by `JACOBIAN`(expr_list,variable_list). This is a matrix whose (i, j) entry is df(expr_list(i),variable_list(j)).

The square Jordan block matrix of dimension n is calculated by the function `JORDAN_BLOCK`(exp,n). The entries of the Jordan_block matrix are $\mathcal{J}$(i,i) = expr for i=1 ... n, $\mathcal{J}$(i,i+1) = 1 for i=1 ... n-1, and all other entries are 0.

$$\texttt{jordan\_block}(\texttt{x},5) \quad = \quad \begin{pmatrix} x & 1 & 0 & 0 & 0 \\ 0 & x & 1 & 0 & 0 \\ 0 & 0 & x & 1 & 0 \\ 0 & 0 & 0 & x & 1 \\ 0 & 0 & 0 & 0 & x \end{pmatrix}$$

`MAKE_IDENTITY`(n) generates the $n \times n$ identity matrix.

`RANDOM_MATRIX`(r,c,limit) generates and $r \times c$ matrix with random values limited by `limit`. The type of entries is controlled by a number of switches.

`IMAGINARY` If on then matrix entries are $x+i*y$ where $-limit < x, y < limit$.

`NOT_NEGATIVE` If on then $0 < entry < limit$. In the imaginary case we have $0 < x, y < limit$.

`ONLY_INTEGER` If on then each entry is an integer. In the imaginary case x and y are integers. If off the values are rounded.

`SYMMETRIC` If on then the matrix is symmetric.

`UPPER_MATRIX` If on then the matrix is upper triangular.

`LOWER_MATRIX` If on then the matrix is lower triangular.

$$\texttt{random\_matrix}(3,3,10) \quad = \quad \begin{pmatrix} -4.729721 & 6.987047 & 7.521383 \\ -5.224177 & 5.797709 & -4.321952 \\ -9.418455 & -9.94318 & -0.730980 \end{pmatrix}$$

```
on only_integer, not_negative, upper_matrix, imaginary;
```

$$\texttt{random\_matrix}(4,4,10) \quad = \quad \begin{pmatrix} 2*i+5 & 3*i+7 & 7*i+3 & 6 \\ 0 & 2*i+5 & 5*i+1 & 2*i+1 \\ 0 & 0 & 8 & i \\ 0 & 0 & 0 & 5*i+9 \end{pmatrix}$$

`TOEPLITZ` creates the Toeplitz matrix from the given expression list. This is a square symmetric matrix in which the first expression is placed on the diagonal and the i^{th} expression is placed on the $(i-1)^{th}$ sub- and super-diagonals. It has dimension equal to the number of expressions.

$$\texttt{toeplitz}(\{w,x,y,z\}) \quad = \quad \begin{pmatrix} w & x & y & z \\ x & w & x & y \\ y & x & w & x \\ z & y & x & w \end{pmatrix}$$

`VANDERMONDE` creates the Vandermonde matrix from the expression list; the square matrix in which the (i,j) entry is expr_list(i) $^{(j-1)}$.

$$\texttt{vandermonde}(\{x,2*y,3*z\}) \quad = \quad \begin{pmatrix} 1 & x & x^2 \\ 1 & 2*y & 4*y^2 \\ 1 & 3*z & 9*z^2 \end{pmatrix}$$

The direct product (or tensor product) is created by the `KRONECKER_PRODUCT` function.

```
a1 := mat((1,2),(3,4),(5,6))$
a2 := mat((1,1,1),(2,z,2),(3,3,3))$
kronecker_product(a1,a2);
```

$$\begin{pmatrix} 1 & 1 & 1 & 2 & 2 & 2 \\ 2 & z & 2 & 4 & 2*z & 4 \\ 3 & 3 & 3 & 6 & 6 & 6 \\ 3 & 3 & 3 & 4 & 4 & 4 \\ 6 & 3*z & 6 & 8 & 4*z & 8 \\ 9 & 9 & 9 & 12 & 12 & 12 \\ 5 & 5 & 5 & 6 & 6 & 6 \\ 10 & 5*z & 10 & 12 & 6*z & 12 \\ 15 & 15 & 15 & 18 & 18 & 18 \end{pmatrix}$$

52.5 Higher Algorithms

The Cholesky decomposition of a matrix can be calculated with the function CHOLESKY. It returns $\{\mathcal{L},\mathcal{U}\}$ where $\mathcal{L}$ is a lower matrix, $\mathcal{U}$ is an upper matrix, and $\mathcal{A} = \mathcal{L}\mathcal{U}$, and $\mathcal{U} = \mathcal{L}^T$.

Gram–Schmidt orthonormalisation can be calculated by GRAM_SCHMIDT. It accepts a list of linearly independent vectors, written as lists, and returns a list of orthogonal normalised vectors.

```
gram_schmidt({{1,0,0},{1,1,0},{1,1,1}});

{{1,0,0},{0,1,0},{0,0,1}}

gram_schmidt({{1,2},{3,4}});

      1          2         2*sqrt(5)     - sqrt(5)
{{---------,---------},{-----------,------------}}
   sqrt(5)   sqrt(5)          5             5
```

The LU decomposition of a real or imaginary matrix with numeric entries is performed by LU_DECOM($\mathcal{A}$). It returns $\{\mathcal{L},\mathcal{U}\}$ where $\mathcal{L}$ is a lower diagonal matrix, $\mathcal{U}$ an upper diagonal matrix and $\mathcal{A} = \mathcal{L}\mathcal{U}$.

Note: the algorithm used can swap the rows of $\mathcal{A}$ during the calculation. This means that $\mathcal{L}\mathcal{U}$ does not equal $\mathcal{A}$ but a row equivalent of it. Due to this, lu_decom returns $\{\mathcal{L},\mathcal{U},\text{vec}\}$. The call CONVERT($\mathcal{A}$,vec) will return the matrix that has been decomposed, *i.e.* $\mathcal{L}\mathcal{U}$ = convert($\mathcal{A}$,vec).

$$\mathcal{K} = \begin{pmatrix} 1 & 3 & 5 \\ -4 & 3 & 7 \\ 8 & 6 & 4 \end{pmatrix}$$

$$\texttt{lu\_decom}(\mathcal{K}) \quad = \quad \left\{ \begin{pmatrix} 8 & 0 & 0 \\ -4 & 6 & 0 \\ 1 & 2.25 & 1.1251 \end{pmatrix}, \begin{pmatrix} 1 & 0.75 & 0.5 \\ 0 & 1 & 1.5 \\ 0 & 0 & 1 \end{pmatrix}, [\,3\;2\;3\,] \right\}$$

PSEUDO_INVERSE, also known as the Moore–Penrose inverse, computes the pseudo inverse of $\mathcal{A}$. Given the singular value decomposition of $\mathcal{A}$, *i.e.* $\mathcal{A} = \mathcal{U}\sum\mathcal{V}^T$, then the pseudo inverse $\mathcal{A}^{-1}$ is defined by $\mathcal{A}^{-1} = \mathcal{V}^T\sum^{-1}\mathcal{U}$.

Thus $\mathcal{A} * \texttt{pseudo\_inverse}(\mathcal{A}) = \mathcal{I}$.

$$\texttt{pseudo\_inverse}(\mathcal{A}) \quad = \quad \begin{pmatrix} -0.2 & 0.1 \\ -0.05 & 0.05 \\ 0.1 & 0 \\ 0.25 & -0.05 \end{pmatrix}$$

The simplex linear programming algorithm for maximising or minimising a function subject to lineal inequalities can be used with the function `SIMPLEX`. It requires three arguments, the first indicates where the action is to maximising or minimising, the second is the test expressions, and the last is a list of linear inequalities. It returns {optimal value,{ values of variables at this optimal}}. The algorithm implies that all the variables are non-negative.

`simplex(`$max, x + y, \{x >= 10, y >= 20, x + y <= 25\}$`);`

`***** Error in simplex:  Problem has no feasible solution.`

`simplex(`$max, 10x + 5y + 5.5z, \{5x + 3z <= 200, x + 0.1y + 0.5z <= 12,$
$0.1x + 0.2y + 0.3z <= 9, 30x + 10y + 50z <= 1500\}$`);`

$$\{525.0, \{x = 40.0, y = 25.0, z = 0\}\}$$

`SVD` computes the singular value decomposition of $\mathcal{A}$ with numeric entries. It returns $\{\mathcal{U}, \sum, \mathcal{V}\}$ where $\mathcal{A} = \mathcal{U} \sum \mathcal{V}^T$ and $\sum = diag(\sigma_1, \ldots, \sigma_n)$. σ_i for $i = (1 \ldots n)$ are the singular values of $\mathcal{A}$. The singular values of $\mathcal{A}$ are the non-negative square roots of the eigenvalues of $\mathcal{A}^T\mathcal{A}$.

$\mathcal{U}$ and $\mathcal{V}$ are such that $\mathcal{U}\mathcal{U}^T = \mathcal{V}\mathcal{V}^T = \mathcal{V}^T\mathcal{V} = \mathcal{I}_n$.

$$\mathcal{Q} = \begin{pmatrix} 1 & 3 \\ -4 & 3 \end{pmatrix}$$

$$\begin{aligned} \texttt{svd}(\mathcal{Q}) \quad = \quad & \left\{ \begin{pmatrix} 0.289784 & 0.957092 \\ -0.957092 & 0.289784 \end{pmatrix}, \begin{pmatrix} 5.149162 & 0 \\ 0 & 2.913094 \end{pmatrix}, \right. \\ & \left. \begin{pmatrix} -0.687215 & 0.726453 \\ -0.726453 & -0.687215 \end{pmatrix} \right\} \end{aligned}$$

`TRIANG_ADJOINT` computes the trianglarizing adjoint of the given matrix. The triangularizing adjoint is a lower triangular matrix. The multiplication of the triangularizing adjoint with the given matrix results in an upper triangular matrix. The i-th entry in the diagonal of this matrix is the determinant of the principal i-th minor of the given matrix.

$$\texttt{triang\_adjoint}(\mathcal{A}) \quad = \quad \begin{pmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ -3 & 6 & -3 \end{pmatrix}$$

The multiplication of this matrix with $\mathcal{A}$ results in an upper triangular matrix.

$$\begin{pmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ -3 & 6 & -3 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad = \quad \begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 0 \end{pmatrix}$$

52.6 Fast Linear Algebra

By turning the `FAST_LA` switch on, the speed of the following functions will be increased:

add_columns	add_rows	augment_columns	column_dim
copy_into	make_identity	matrix_augment	matrix_stack
minor	mult_column	mult_row	pivot
remove_columns	remove_rows	rows_pivot	squarep
stack_rows	sub_matrix	swap_columns	swap_entries
swap_rows	symmetricp		

The increase in speed will be insignificant unless you are making a thousands of calls. When using this switch, error checking is minimised, and thus illegal input may give strange error messages.

Chapter 53

MATHML : MathML Interface for REDUCE

Luis Alvarez-Sobreviela
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraše 7
D-14195 Berlin-Dahlem, Germany

MathML is intended to facilitate the use and re-use of mathematical and scientific content on the Web, and for other applications such as computer algebra systems.
This package contains the MathML-REDUCE interface. This interface provides an easy to use series of commands, allowing to evaluate and output MathML.

The principal features of this package can be resumed as:

- Evaluation of MathML code. Allows REDUCE to parse MathML expressions and evaluate them.
- Generation of MathML compliant code. Provides the printing of REDUCE expressions in MathML source code, to be used directly in web page production.

We assume that the reader is familiar with MathML. If not, the specification[1] is available at: `http://www.w3.org/TR/WD-math/`

[1] This specification is subject to change, since it is not yet a final draft. During the

The MathML-REDUCE interface package is loaded by supplying `load mathml;`.

Switches

There are two switches which can be used alternatively and incrementally. These are `MATHML` and `BOTH`. Their use can be described as follows:

`mathml`: All output will be printed in MathML.

`both`: All output will be printed in both MathML and normal REDUCE.

`web`: All output will be printed within an HTML <embed> tag. This is for direct use in an HTML web page. Only works when `mathml` is on.

MathML has often been said to be too verbose. If `BOTH` is on, an easy interpretation of the results is possible, improving MathML readability.

Operators of Package MathML

`mml`(filename): This function opens and reads the file filename containing the MathML.

`parseml()`: To introduce a series of valid mathml tokens you can use this function. It takes no arguments and will prompt you to enter mathml tags stating with <mathml> and ending with </mathml>. It returns an expression resulting from evaluating the input.

Example

```
1:  load mathml;

3:  on both;

3:  int(2*x+1,x);;

      x*(x + 1)
```

---

two month period in which this package was developed, the specification changed, forcing a review of the code. This package is based on the Nov 98 version.

```
<mathml>
   <apply><plus/>
      <apply><power/>
         <ci>x</ci>
         <cn type="integer">2</cn>
      </apply>
      <ci>x</ci>
   </apply>
</mathml>
```

4:

Chapter 54

MODSR: Modular solve and roots

Herbert Melenk
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany
e–mail: melenk@zib.de

This package supports solve (M_SOLVE) and roots (M_ROOTS) operators for modular polynomials and modular polynomial systems. The moduli need not be primes. M_SOLVE requires a modulus to be set. M_ROOTS takes the modulus as a second argument. For example:

```
on modular; setmod 8;
m_solve(2x=4);            ->  {{X=2},{X=6}}
m_solve({x^2-y^3=3});
    ->  {{X=0,Y=5}, {X=2,Y=1}, {X=4,Y=5}, {X=6,Y=1}}
m_solve({x=2,x^2-y^3=3}); ->  {{X=2,Y=1}}
off modular;
m_roots(x^2-1,8);         ->  {1,3,5,7}
m_roots(x^3-x,7);         ->  {0,1,6}
```

Chapter 55

MRVLIMIT: Package for Computing Limits of "Exp-Log" Functions

Neil Langmead
Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)
Takustraŝe 7
D - 14195 Berlin-Dahlem, Germany

Using the LIMITS package to compute the limits of functions containing exponential and logarithmic expressions may raise a problem. For the computation of indefinite forms (such as 0/0,or $\frac{\infty}{\infty}$) L'Hospital's rule may only be applied a finite number of times in a CAS. In REDUCE it is applied 3 times. This algorithm of Dominik Gruntz of the ETH Zürich solves this particular problem, and enables the computation of many more limit calculations in REDUCE.

```
1: load limits;

2: limit(x^7/e^x,x,infinity);

             7
            x
      limit(----,x,infinity)
             x
            e
```

```
3: load mrvlimit;

4: mrv_limit(x^7/e^x,x,infinity);

      0
```

For this example, the MRVLIMIT package is able to compute the correct limit.

`MRV_LIMIT`(EXPRN:*algebraic*, VAR:*kernel*, LIMPOINT:*algebraic*):*algebraic*

The result is the limit of EXPRN as VAR approaches LIMPOINT.

A switch `TRACELIMIT` is available to inform the user about the computed Taylor expansion, all recursive calls and the return value of the internally called function `MRV`.

Examples:

```
5: b:=e^x*(e^(1/x-e^-x)-e^(1/x));

                -1         - x
           x + x      - e
      b:= e       *(e          - 1)


6: mrv_limit(b,x,infinity);


      -1

                                          -1
7: ex:= - log(log(log(log(x))) + log(x))  *log(x)

                     *(log(log(x)) - log(log(log(x)) + log(x)));


                - log(x)*(log(log(x)) - log(log(log(x)) + log(x)))
      ex:=     ----------------------------------------------------
                           log(log(log(log(x))) + log(x))

8: off mcd;
```

```
9: mrv_limit(ex,x,infinity);

      1
```

Chapter 56

NCPOLY: Non–commutative polynomial ideals

Herbert Melenk
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany
e–mail: melenk@zib.de

Joachim Apel
Institut für Informatik, Universität Leipzig
Augustusplatz 10–11
D–04109 Leipzig, Germany
e–mail: apel@informatik.uni–leipzig.de

REDUCE supports a very general mechanism for computing with objects under a non–commutative multiplication, where commutator relations must be introduced explicitly by rule sets when needed. The package **NCPOLY** allows the user to set up automatically a consistent environment for computing in an algebra where the non–commutativity is defined by Lie-bracket commutators. The package uses the REDUCE **noncom** mechanism for elementary polynomial arithmetic; the commutator rules are automatically computed from the Lie brackets. Polynomial arithmetic may be performed directly, including **division** and **factorisation**. Additionally **NCPOLY** supports computations in a one sided ideal (left or right), especially one sided **Gröbner** bases and **polynomial reduction**.

56.1 Setup, Cleanup

Before the computations can start the environment for a non–commutative computation must be defined by a call to **nc_setup**:

```
nc_setup(<vars>[,<comms>][,<dir>]);
```

where

$< vars >$ is a list of variables; these must include the non–commutative quantities.

$< comms >$ is a list of equations **<u>*<v> - <v>*<u>=<rh>** where $< u >$ and $< v >$ are members of $< vars >$, and $< rh >$ is a polynomial.

$< dir >$ is either $left$ or $right$ selecting a left or a right one sided ideal. The initial direction is $left$.

nc_setup generates from $< comms >$ the necessary rules to support an algebra where all monomials are ordered corresponding to the given variable sequence. All pairs of variables which are not explicitly covered in the commutator set are considered as commutative and the corresponding rules are also activated.

The second parameter in **nc_setup** may be omitted if the operator is called for the second time, *e.g.* with a reordered variable sequence. In such a case the last commutator set is used again.

Remarks:

- The variables need not be declared **noncom** - **nc_setup** performs all necessary declarations.
- The variables need not be formal operator expressions; **nc_setup** encapsulates a variable x internally as nc!*(!_x) expressions anyway where the operator $nc!*$ keeps the noncom property.
- The commands **order** and **korder** should be avoided because **nc_setup** sets these such that the computation results are printed in the correct term order.

Example:

```
nc_setup({KK,NN,k,n},
```

```
            {NN*n-n*NN= NN, KK*k-k*KK= KK});

NN*N;            ->   NN*N
N*NN;            ->   NN*N - NN
nc_setup({k,n,KK,NN});
NN*N - NN        ->   N*NN;
```

Here KK, NN, k, n are non–commutative variables where the commutators are described as $[NN, n] = NN$, $[KK, k] = KK$.

The current term order must be compatible with the commutators: the product $< u > * < v >$ must precede all terms on the right hand side $< rh >$ under the current term order. Consequently

- the maximal degree of $< u >$ or $< v >$ in $< rh >$ is 1,
- in a total degree ordering the total degree of $< rh >$ may be not higher than 1,
- in an elimination degree order ($e.g.$ lex) all variables in $< rh >$ must be below the minimum of $< u >$ and $< v >$.
- If $< rh >$ does not contain any variables or has at most $< u >$ or $< v >$, any term order can be selected.

To use the non–commutative variables or results from non–commutative computations later in commutative operations it might be necessary to switch off the non–commutative evaluation mode because not all operators in REDUCE are prepared for that environment. In such a case use the command

```
nc_cleanup;
```

without parameters. It removes all internal rules and definitions which `nc_setup` had introduced. To reactive non–commutative call `nc_setup` again.

56.2 Left and right ideals

A (polynomial) left ideal L is defined by the axioms

$u \in L, v \in L \Longrightarrow u + v \in L$

$u \in L \Longrightarrow k * u \in L$ for an arbitrary polynomial k

where "*" is the non–commutative multiplication. Correspondingly, a right ideal R is defined by

$u \in R, v \in R \Longrightarrow u + v \in R$

$u \in R \Longrightarrow u * k \in R$ for an arbitrary polynomial k

56.3 Gröbner bases

When a non–commutative environment has been set up by `nc_setup`, a basis for a left or right polynomial ideal can be transformed into a Gröbner basis by the operator `nc_groebner`

```
nc_groebner(<plist>);
```

Note that the variable set and variable sequence must be defined before in the `nc_setup` call. The term order for the Gröbner calculation can be set by using the `torder` declaration.

For details about `torder` see the **REDUCE GROEBNER** manual, or chapter 45.

```
2: nc_setup({k,n,NN,KK},{NN*n-n*NN=NN,KK*k-k*KK=KK},left);

3: p1 := (n-k+1)*NN - (n+1);

p1 :=  - k*nn + n*nn - n + nn - 1

4: p2 := (k+1)*KK -(n-k);

p2 := k*kk + k - n + kk

5: nc_groebner ({p1,p2});

{k*nn - n*nn + n - nn + 1,

 k*kk + k - n + kk,

 n*nn*kk - n*kk - n + nn*kk - kk - 1}
```

Important: Do not use the operators of the GROEBNER package directly as they would not consider the non–commutative multiplication.

56.4 Left or right polynomial division

The operator `nc_divide` computes the one sided quotient and remainder of two polynomials:

```
nc_divide(<p1>,<p2>);
```

The result is a list with quotient and remainder. The division is performed as a pseudo–division, multiplying $< p1 >$ by coefficients if necessary. The result $\{< q >, < r >\}$ is defined by the relation

$< c > * < p1 >=< q > * < p2 > + < r >$ for direction $left$ and

$< c > * < p1 >=< p2 > * < q > + < r >$ for direction $right$,

where $< c >$ is an expression that does not contain any of the ideal variables, and the leading term of $< r >$ is lower than the leading term of $< p2 >$ according to the actual term order.

56.5 Left or right polynomial reduction

For the computation of the one sided remainder of a polynomial modulo a given set of other polynomials the operator `nc_preduce` may be used:

```
nc_preduce(<polynomial>,<plist>);
```

The result of the reduction is unique (canonical) if and only if $< plist >$ is a one sided Gröbner basis. Then the computation is at the same time an ideal membership test: if the result is zero, the polynomial is member of the ideal, otherwise not.

56.6 Factorisation

Polynomials in a non–commutative ring cannot be factored using the ordinary `factorize` command of REDUCE. Instead one of the operators of this section must be used:

```
nc_factorize(<polynomial>);
```

The result is a list of factors of < *polynomial* >. A list with the input expression is returned if it is irreducible.

As non–commutative factorisation is not unique, there is an additional operator which computes all possible factorisations

```
nc_factorize_all(<polynomial>);
```

The result is a list of factor decompositions of < *polynomial* >. If there are no factors at all the result list has only one member which is a list containing the input polynomial.

56.7 Output of expressions

It is often desirable to have the commutative parts (coefficients) in a non–commutative operation condensed by factorisation. The operator

```
nc_compact(<polynomial>)
```

collects the coefficients to the powers of the lowest possible non-commutative variable.

```
load_package ncpoly;

nc_setup({n,NN},{NN*n-n*NN=NN})$
p1 := n**4 + n**2*nn + 4*n**2 + 4*n*nn + 4*nn + 4;

       4    2         2
p1 := n  + n *nn + 4*n  + 4*n*nn + 4*nn + 4

nc_compact p1;

  2     2          2
(n  + 2)  + (n + 2) *nn
```

Chapter 57

NORMFORM: Computation of matrix normal forms

Matt Rebbeck
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany

This package contains routines for computing the following normal forms of matrices:

- smithex_int
- smithex
- frobenius
- ratjordan
- jordansymbolic
- jordan.

By default all calculations are carried out in $\mathcal{Q}$ (the rational numbers). For `smithex`, `frobenius`, `ratjordan`, `jordansymbolic`, and `jordan`, this field can be extended to an algebraic number field using ARNUM (chapter 22). The `frobenius`, `ratjordan`, and `jordansymbolic` normal forms can also be computed in a modular base.

57.1 Smithex

`Smithex`($\mathcal{A}$, x) computes the Smith normal form $\mathcal{S}$ of the matrix $\mathcal{A}$.

It returns $\{\mathcal{S}, \mathcal{P}, \mathcal{P}^{-1}\}$ where $\mathcal{S}, \mathcal{P}$, and $\mathcal{P}^{-1}$ are such that $\mathcal{PSP}^{-1} = \mathcal{A}$.

$\mathcal{A}$ is a rectangular matrix of univariate polynomials in x where x is the variable name.

`load_package normform;`

$$\mathcal{A} = \begin{pmatrix} x & x+1 \\ 0 & 3*x^2 \end{pmatrix}$$

$$\texttt{smithex}(\mathcal{A},\, x) \quad = \quad \left\{ \begin{pmatrix} 1 & 0 \\ 0 & x^3 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 3*x^2 & 1 \end{pmatrix}, \begin{pmatrix} x & x+1 \\ -3 & -3 \end{pmatrix} \right\}$$

57.2 Smithex_int

Given an n by m rectangular matrix $\mathcal{A}$ that contains *only* integer entries, `smithex_int`($\mathcal{A}$) computes the Smith normal form $\mathcal{S}$ of $\mathcal{A}$.

It returns $\{\mathcal{S}, \mathcal{P}, \mathcal{P}^{-1}\}$ where $\mathcal{S}, \mathcal{P}$, and $\mathcal{P}^{-1}$ are such that $\mathcal{PSP}^{-1} = \mathcal{A}$.

`load_package normform;`

$$\mathcal{A} = \begin{pmatrix} 9 & -36 & 30 \\ -36 & 192 & -180 \\ 30 & -180 & 180 \end{pmatrix}$$

`smithex_int`($\mathcal{A}$) =

$$\left\{ \begin{pmatrix} 3 & 0 & 0 \\ 0 & 12 & 0 \\ 0 & 0 & 60 \end{pmatrix}, \begin{pmatrix} -17 & -5 & -4 \\ 64 & 19 & 15 \\ -50 & -15 & -12 \end{pmatrix}, \begin{pmatrix} 1 & -24 & 30 \\ -1 & 25 & -30 \\ 0 & -1 & 1 \end{pmatrix} \right\}$$

57.3 Frobenius

`Frobenius`($\mathcal{A}$) computes the Frobenius normal form $\mathcal{F}$ of the matrix $\mathcal{A}$.

It returns $\{\mathcal{F},\mathcal{P},\mathcal{P}^{-1}\}$ where $\mathcal{F},\mathcal{P}$, and $\mathcal{P}^{-1}$ are such that $\mathcal{P}\mathcal{F}\mathcal{P}^{-1}=\mathcal{A}$.

$\mathcal{A}$ is a square matrix.

`load_package normform;`

$$\mathcal{A}=\begin{pmatrix} \frac{-x^2+y^2+y}{y} & \frac{-x^2+x+y^2-y}{y} \\ \frac{-x^2-x+y^2+y}{y} & \frac{-x^2+x+y^2-y}{y} \end{pmatrix}$$

`frobenius`$(\mathcal{A})=$

$$\left\{\begin{pmatrix} 0 & \frac{x*(x^2-x-y^2+y)}{y} \\ 1 & \frac{-2*x^2+x+2*y^2}{y} \end{pmatrix},\begin{pmatrix} 1 & \frac{-x^2+y^2+y}{y} \\ 0 & \frac{-x^2-x+y^2+y}{y} \end{pmatrix},\begin{pmatrix} 1 & \frac{-x^2+y^2+y}{x^2+x-y^2-y} \\ 0 & \frac{-y}{x^2+x-y^2-y} \end{pmatrix}\right\}$$

57.4 Ratjordan

`Ratjordan`$(\mathcal{A})$ computes the rational Jordan normal form $\mathcal{R}$ of the matrix $\mathcal{A}$.

It returns $\{\mathcal{R},\mathcal{P},\mathcal{P}^{-1}\}$ where $\mathcal{R},\mathcal{P}$, and $\mathcal{P}^{-1}$ are such that $\mathcal{P}\mathcal{R}\mathcal{P}^{-1}=\mathcal{A}$.

$\mathcal{A}$ is a square matrix.

`load_package normform;`

$$\mathcal{A}=\begin{pmatrix} x+y & 5 \\ y & x^2 \end{pmatrix}$$

`ratjordan`$(\mathcal{A})=$

$$\left\{\begin{pmatrix} 0 & -x^3-x^2*y+5*y \\ 1 & x^2+x+y \end{pmatrix},\begin{pmatrix} 1 & x+y \\ 0 & y \end{pmatrix},\begin{pmatrix} 1 & \frac{-(x+y)}{y} \\ 0 & \frac{1}{y} \end{pmatrix}\right\}$$

57.5 Jordansymbolic

`Jordansymbolic`$(\mathcal{A})$ computes the Jordan normal form $\mathcal{J}$ of the matrix $\mathcal{A}$.

It returns $\{\mathcal{J},\mathcal{L},\mathcal{P},\mathcal{P}^{-1}\}$, where $\mathcal{J},\mathcal{P}$, and $\mathcal{P}^{-1}$ are such that $\mathcal{P}\mathcal{J}\mathcal{P}^{-1}=\mathcal{A}$. $\mathcal{L}=\{\ ll,\ \xi\ \}$, where ξ is a name and ll is a list of irreducible factors of $p(\xi)$.

$\mathcal{A}$ is a square matrix.

`load_package normform;`

$$\mathcal{A}=\begin{pmatrix} 1 & y \\ y^2 & 3 \end{pmatrix}$$

$\texttt{jordansymbolic}(\mathcal{A}) =$

$$\left\{\begin{pmatrix} \xi_{11} & 0 \\ 0 & \xi_{12} \end{pmatrix},\left\{\left\{-y^3+\xi^2-4*\xi+3\right\},\xi\right\},\right.$$
$$\left.\begin{pmatrix} \xi_{11}-3 & \xi_{12}-3 \\ y^2 & y^2 \end{pmatrix},\begin{pmatrix} \frac{\xi_{11}-2}{2*(y^3-1)} & \frac{\xi_{11}+y^3-1}{2*y^2*(y^3+1)} \\ \frac{\xi_{12}-2}{2*(y^3-1)} & \frac{\xi_{12}+y^3-1}{2*y^2*(y^3+1)} \end{pmatrix}\right\}$$

$\texttt{solve}(-\texttt{y}^3+\texttt{xi}^2-4*\texttt{xi}+3,\texttt{xi});$

$$\{\xi=\sqrt{y^3+1}+2,\ \xi=-\sqrt{y^3+1}+2\}$$

$\mathcal{J}=\texttt{sub}(\{\texttt{xi}(1,1)=\texttt{sqrt}(\texttt{y}^3+1)+2,\ \texttt{xi}(1,2)=-\texttt{sqrt}(\texttt{y}^3+1)+2\},$
$\quad \texttt{first jordansymbolic }(\mathcal{A}));$

$$\mathcal{J}=\begin{pmatrix} \sqrt{y^3+1}+2 & 0 \\ 0 & -\sqrt{y^3+1}+2 \end{pmatrix}$$

57.6 Jordan

$\texttt{Jordan}(\mathcal{A})$ computes the Jordan normal form $\mathcal{J}$ of the matrix $\mathcal{A}$.

It returns $\{\mathcal{J},\mathcal{P},\mathcal{P}^{-1}\}$, where $\mathcal{J},\mathcal{P}$, and $\mathcal{P}^{-1}$ are such that $\mathcal{P}\mathcal{J}\mathcal{P}^{-1}=\mathcal{A}$.

$\mathcal{A}$ is a square matrix.

`load_package normform;`

$$\mathcal{A} = \begin{pmatrix} -9 & -21 & -15 & 4 & 2 & 0 \\ -10 & 21 & -14 & 4 & 2 & 0 \\ -8 & 16 & -11 & 4 & 2 & 0 \\ -6 & 12 & -9 & 3 & 3 & 0 \\ -4 & 8 & -6 & 0 & 5 & 0 \\ -2 & 4 & -3 & 0 & 1 & 3 \end{pmatrix}$$

$\mathcal{J}$ `= first jordan(`$\mathcal{A}$`);`

$$\mathcal{J} = \begin{pmatrix} 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & i+2 & 0 \\ 0 & 0 & 0 & 0 & 0 & -i+2 \end{pmatrix}$$

Chapter 58

NUMERIC: Solving numerical problems

Herbert Melenk
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany
e–mail: melenk@zib.de

The NUMERIC package implements some numerical (approximative) algorithms for REDUCE based on the REDUCE rounded mode arithmetic. These algorithms are implemented for standard cases. They should not be called for ill-conditioned problems; please use standard mathematical libraries for these.

58.1 Syntax

58.1.1 Intervals, Starting Points

Intervals are generally coded as lower bound and upper bound connected by the operator `..`, usually associated to a variable in an equation.

```
x= (2.5 .. 3.5)
```

means that the variable x is taken in the range from 2.5 up to 3.5. Note, that the bounds can be algebraic expressions, which, however, must evaluate to numeric results. In cases where an interval is returned as the result, the lower and upper bounds can be extracted by the `PART` operator as the first and second part respectively. A starting point is specified by an equation with a numeric righthand side,

```
x=3.0
```

If for multivariate applications several coordinates must be specified by intervals or as a starting point, these specifications can be collected in one parameter (which is then a list) or they can be given as separate parameters alternatively. The list form is more appropriate when the parameters are built from other REDUCE calculations in an automatic style, while the flat form is more convenient for direct interactive input.

58.1.2 Accuracy Control

The keyword parameters $accuracy = a$ and $iterations = i$, where a and i must be positive integer numbers, control the iterative algorithms: the iteration is continued until the local error is below 10^{-a}; if that is impossible within i steps, the iteration is terminated with an error message. The values reached so far are then returned as the result.

58.2 Minima

The function to be minimised must have continuous partial derivatives with respect to all variables. The starting point of the search can be specified; if not, random values are taken instead. The steepest descent algorithms in general find only local minima.

Syntax:

NUM_MIN $(exp, var_1[= val_1][, var_2[= val_2] \ldots]$
$[, accuracy = a][, iterations = i])$
or

NUM_MIN $(exp, \{var_1[= val_1][, var_2[= val_2] \ldots]\}$
$[, accuracy = a][, iterations = i])$

where *exp* is a function expression,

$var_1, var_2, \ldots$ are the variables in *exp* and $val_1, val_2, \ldots$ are the (optional) start values.

NUM_MIN tries to find the next local minimum along the descending path starting at the given point. The result is a list with the minimum function value as first element followed by a list of equations, where the variables are equated to the coordinates of the result point.

Examples:

```
num_min(sin(x)+x/5, x);

{4.9489585606,{X=29.643767785}}

num_min(sin(x)+x/5, x=0);

{ - 1.3342267466,{X= - 1.7721582671}}

% Rosenbrock function (well known as hard to minimize).
fktn := 100*(x1**2-x2)**2 + (1-x1)**2;
num_min(fktn, x1=-1.2, x2=1, iterations=200);

{0.00000021870228295,{X1=0.99953284494,X2=0.99906807238}}
```

58.3 Roots of Functions/ Solutions of Equations

An adaptively damped Newton iteration is used to find an approximative zero of a function, a function vector or the solution of an equation or an equation system. The expressions must have continuous derivatives for all variables. A starting point for the iteration can be given. If not given, random values are taken instead. If the number of forms is not equal to the number of variables, the Newton method cannot be applied. Then the minimum of the sum of absolute squares is located instead.

With `ON COMPLEX` solutions with imaginary parts can be found, if either the expression(s) or the starting point contain a nonzero imaginary part.

Syntax:

NUM_SOLVE $(exp_1, var_1[= val_1][, accuracy = a][, iterations = i])$

or

NUM_SOLVE $(\{exp_1,\ldots,exp_n\}, var_1[=val_1],\ldots,var_1[=val_n]$

$[, accuracy = a][, iterations = i])$

or

NUM_SOLVE $(\{exp_1,\ldots,exp_n\}, \{var_1[=val_1],\ldots,var_1[=val_n]\}$

$[, accuracy = a][, iterations = i])$

where $exp_1,\ldots,exp_n$ are function expressions,

$var_1,\ldots,var_n$ are the variables,

$val_1,\ldots,val_n$ are optional start values.

NUM_SOLVE tries to find a zero/solution of the expression(s). Result is a list of equations, where the variables are equated to the coordinates of the result point.

The Jacobian matrix is stored as a side effect in the shared variable JACOBIAN.

Example:

```
num_solve({sin x=cos y, x + y = 1},{x=1,y=2});

{X= - 1.8561957251,Y=2.856195584}

jacobian;

[COS(X)  SIN(Y)]
[              ]
[  1       1   ]
```

58.4 Integrals

Numerical integration uses a polyalgorithm, explained in the full documentation.

NUM_INT $(exp, var_1 = (l_1..u_1)[, var_2 = (l_2..u_2)\ldots]$

$[, accuracy = a][, iterations = i])$

where exp is the function to be integrated,

$var_1, var_2, \ldots$ are the integration variables,

$l_1, l_2, \ldots$ are the lower bounds,

$u_1, u_2, \ldots$ are the upper bounds.

Result is the value of the integral.

Example:

```
num_int(sin x,x=(0 .. pi));

2.0000010334
```

58.5 Ordinary Differential Equations

A Runge-Kutta method of order 3 finds an approximate graph for the solution of a ordinary differential equation real initial value problem.

Syntax:

NUM_ODESOLVE (*exp*,*depvar* = *dv*,*indepvar*=($from..to$)

$[, accuracy = a][, iterations = i])$

where

exp is the differential expression/equation,

depvar is an identifier representing the dependent variable (function to be found),

indepvar is an identifier representing the independent variable,

exp is an equation (or an expression implicitly set to zero) which contains the first derivative of *depvar* wrt *indepvar*,

$from$ is the starting point of integration,

to is the endpoint of integration (allowed to be below $from$),

dv is the initial value of *depvar* in the point $indepvar = from$.

The ODE *exp* is converted into an explicit form, which then is used for a Runge-Kutta iteration over the given range. The number of steps is controlled by the value of i (default: 20). If the steps are too coarse to reach the desired accuracy in the neighbourhood of the starting point, the number is increased automatically.

Result is a list of pairs, each representing a point of the approximate solution of the ODE problem.

Example:

```
   num_odesolve(df(y,x)=y,y=1,x=(0 .. 1), iterations=5);

{{0.0,1.0},{0.2,1.2214},{0.4,1.49181796},{0.6,1.8221064563},

 {0.8,2.2255208258},{1.0,2.7182511366}}
```

58.6 Bounds of a Function

Upper and lower bounds of a real valued function over an interval or a rectangular multivariate domain are computed by the operator `BOUNDS`. Some knowledge about the behaviour of special functions like ABS, SIN, COS, EXP, LOG, fractional exponentials etc. is integrated and can be evaluated if the operator BOUNDS is called with rounded mode on (otherwise only algebraic evaluation rules are available).

If BOUNDS finds a singularity within an interval, the evaluation is stopped with an error message indicating the problem part of the expression.

Syntax:

BOUNDS $(exp, var_1 = (l_1..u_1)[, var_2 = (l_2..u_2) \ldots])$

BOUNDS $(exp, \{var_1 = (l_1..u_1)[, var_2 = (l_2..u_2) \ldots]\})$

where exp is the function to be investigated,

$var_1, var_2, \ldots$ are the variables of exp,

$l_1, l_2, \ldots$ and $u_1, u_2, \ldots$ specify the area (intervals).

`BOUNDS` computes upper and lower bounds for the expression in the given area. An interval is returned.

Example:

```
bounds(sin x,x=(1 .. 2));

{-1,1}

on rounded;
bounds(sin x,x=(1 .. 2));

0.84147098481 .. 1

bounds(x**2+x,x=(-0.5 .. 0.5));

 - 0.25 .. 0.75
```

58.7 Chebyshev Curve Fitting

The operator family $Chebyshev\_\ldots$ implements approximation and evaluation of functions by the Chebyshev method.

The operator `Chebyshev_fit` computes this approximation and returns a list, which has as first element the sum expressed as a polynomial and as second element the sequence of Chebyshev coefficients c_i. `Chebyshev_df` and `Chebyshev_int` transform a Chebyshev coefficient list into the coefficients of the corresponding derivative or integral respectively. For evaluating a Chebyshev approximation at a given point in the basic interval the operator

`Chebyshev_eval` can be used. Note that `Chebyshev_eval` is based on a recurrence relation which is in general more stable than a direct evaluation of the complete polynomial.

CHEBYSHEV_FIT $(fcn, var = (lo..hi), n)$

CHEBYSHEV_EVAL $(coeffs, var = (lo..hi), var = pt)$

CHEBYSHEV_DF $(coeffs, var = (lo..hi))$

CHEBYSHEV_INT $(coeffs, var = (lo..hi))$

> where fcn is an algebraic expression (the function to be fitted), var is the variable of fcn, lo and hi are numerical real values which describe an interval ($lo < hi$), n is the approximation order,an integer > 0, set to 20 if missing, pt is a numerical value in the interval and $coeffs$ is a series of Chebyshev coefficients, computed by one of $CHEBYSHEV\_COEFF$, $\_DF$ or $\_INT$.

Example:

```
on rounded;

w:=chebyshev_fit(sin x/x,x=(1 .. 3),5);

                3            2
w := {0.03824*x  - 0.2398*x  + 0.06514*x + 0.9778,

      {0.8991,-0.4066,-0.005198,0.009464,-0.00009511}}

chebyshev_eval(second w, x=(1 .. 3), x=2.1);

0.4111
```

58.8 General Curve Fitting

The operator `NUM_FIT` finds for a set of points the linear combination of a given set of functions (function basis) which approximates the points best under the objective of the least squares criterion (minimum of the sum of the squares of the deviation). The solution is found as zero of the gradient vector of the sum of squared errors.

Syntax:

NUM_FIT $(vals, basis, var = pts)$

> where $vals$ is a list of numeric values,
>
> var is a variable used for the approximation,
>
> pts is a list of coordinate values which correspond to var,
>
> $basis$ is a set of functions varying in var which is used for the approximation.

The result is a list containing as first element the function which approximates the given values, and as second element a list of coefficients which were used to build this function from the basis.

Example:

```
 % approximate a set of factorials by a polynomial
pts:=for i:=1 step 1 until 5 collect i$
vals:=for i:=1 step 1 until 5 collect
        for j:=1:i product j$

num_fit(vals,{1,x,x**2},x=pts);

              2
{14.571428571*X  - 61.428571429*X + 54.6,{54.6,

     - 61.428571429,14.571428571}}

num_fit(vals,{1,x,x**2,x**3,x**4},x=pts);

              4                  3
{2.2083333234*X  - 20.249999879*X

                 2
  + 67.791666154*X  - 93.749999133*X

  + 44.999999525,

 {44.999999525, - 93.749999133,67.791666154,
```

```
   - 20.249999879,2.2083333234}}
```

58.9 Function Bases

The following procedures compute sets of functions for example to be used for approximation. All procedures have two parameters, the expression to be used as *variable* (an identifier in most cases) and the order of the desired system. The functions are not scaled to a specific interval, but the *variable* can be accompanied by a scale factor and/or a translation in order to map the generic interval of orthogonality to another (*e.g.* $(x - 1/2) * 2pi$). The result is a function list with ascending order, such that the first element is the function of order zero and (for the polynomial systems) the function of order n is the $n + 1$-th element.

```
monomial_base(x,n)        {1,x,...,x**n}
trigonometric_base(x,n)   {1,sin x,cos x,sin(2x),cos(2x)...}
Bernstein_base(x,n)       Bernstein polynomials
Legendre_base(x,n)        Legendre polynomials
Laguerre_base(x,n)        Laguerre polynomials
Hermite_base(x,n)         Hermite polynomials
Chebyshev_base_T(x,n)     Chebyshev polynomials first kind
Chebyshev_base_U(x,n)     Chebyshev polynomials second kind
```

Example:

```
Bernstein_base(x,5);

       5      4       3       2
  { - X  + 5*X  - 10*X  + 10*X  - 5*X + 1,

         4      3      2
   5*X*(X  - 4*X  + 6*X  - 4*X + 1),

       2      3      2
   10*X *( - X  + 3*X  - 3*X + 1),
```

```
    3   2
10*X *(X  - 2*X + 1),

   4
5*X *( - X + 1),

 5
X }
```

Chapter 59

ODESOLVE: Ordinary differential equations solver

Malcolm A.H. MacCallum
School of Mathematical Sciences, Queen Mary and Westfield College
University of London
Mile End Road
London E1 4NS, England
e–mail: mm@maths.qmw.ac.uk

The ODESOLVE package is a solver for ordinary differential equations. At the present time it has very limited capabilities,

1. it can handle only a single scalar equation presented as an algebraic expression or equation, and
2. it can solve only first-order equations of simple types, linear equations with constant coefficients and Euler equations.

These solvable types are exactly those for which Lie symmetry techniques give no useful information.

59.1 Use

The only top-level function the user should normally invoke is:

`ODESOLVE`(EXPRN:*expression, equation*,
VAR1:*variable*,
VAR2:*variable*):*list-algebraic*

`ODESOLVE` returns a list containing an equation (like solve):

EXPRN is a single scalar expression such that EXPRN = 0 is the ordinary differential equation (ODE for short) to be solved, or is an equivalent equation.

VAR1 is the name of the dependent variable.

VAR2 is the name of the independent variable

(For simplicity these will be called y and x in the sequel) The returned value is a list containing the equation giving the general solution of the ODE (for simultaneous equations this will be a list of equations eventually). It will contain occurrences of the operator `ARBCONST` for the arbitrary constants in the general solution. The arguments of `ARBCONST` should be new, as with `ARBINT` etc. in SOLVE. A counter `!!ARBCONST` is used to arrange this (similar to the way `ARBINT` is implemented).

Some other top-level functions may be of use elsewhere, especially:

`SORTOUTODE`(EXPRN:*algebraic*, Y:*var*, X:*var*): *expression*

which finds the order and degree of the EXPRN as a differential equation for Y with respect to Y and sets the linearity and highest derivative occurring in reserved variables ODEORDER, ODEDEGREE,ODELINEARITY and HIGHESTDERIV. An expression equivalent to the ODE is returned, or zero if EXPRN (equated to 0) is not an ODE in the given variables.

59.2 Commentary

The methods used by this package are described in detail in the full documentation, which should be inspected together with the examples file.

Chapter 60

ORTHOVEC: Three-dimensional vector analysis

James W. Eastwood
AEA Technology, Culham Laboratory
Abingdon
Oxon OX14 3DB, England
e–mail: jim_eastwood@aeat.co.uk

The ORTHOVEC package is a collection of REDUCE procedures and operations which provide a simple to use environment for the manipulation of scalars and vectors. Operations include addition, subtraction, dot and cross products, division, modulus, div, grad, curl, laplacian, differentiation, integration, $\mathbf{a} \cdot \nabla$ and Taylor expansion.

60.1 Initialisation

The procedure `START` initialises ORTHOVEC. VSTART provides a menu of standard coordinate systems:-

1. cartesian $(x, y, z) =$ `(x, y, z)`
2. cylindrical $(r, \theta, z) =$ `(r, th, z)`

3. spherical (r, θ, ϕ) = `(r, th, ph)`
4. general (u_1, u_2, u_3) = `(u1, u2, u3)`
5. others

which the user selects by number. Selecting options (1)-(4) automatically sets up the coordinates and scale factors. Selection option (5) shows the user how to select another coordinate system. If VSTART is not called, then the default cartesian coordinates are used. ORTHOVEC may be re-initialised to a new coordinate system at any time during a given REDUCE session by typing

```
VSTART $.
```

60.2 Input-Output

ORTHOVEC assumes all quantities are either scalars or 3 component vectors. To define a vector a with components (c_1, c_2, c_3) use the procedure SVEC:

```
a := svec(c1, c2, c3);
```

The procedure `VOUT` (which returns the value of its argument) can be used to give labelled output of components in algebraic form:

```
b := svec (sin(x)**2, y**2, z)$
vout(b)$
```

The operator _ can be used to select a particular component (1, 2 or 3) for output *e.g.*

```
b_1 ;
```

60.3 Algebraic Operations

Six infix operators, sum, difference, quotient, times, exponentiation and cross product, and four prefix operators, plus, minus, reciprocal and modulus are defined in ORTHOVEC. These operators can take suitable combinations of scalar and vector arguments, and in the case of scalar arguments reduce to the usual definitions of $+, -, *, /$, etc.

The operators are represented by symbols

`+, -, /, *, ^, ><`

The composite `><` is an attempt to represent the cross product symbol × in ASCII characters. If we let **v** be a vector and s be a scalar, then valid combinations of arguments of the procedures and operators and the type of the result are as summarised below. The notation used is
result :=procedure(left argument, right argument) or
result :=(left operand) operator (right operand) .

<u>Vector Addition</u>

v	:=	VECTORPLUS(**v**)	or	**v**	:=	+ **v**
s	:=	VECTORPLUS(s)	or	s	:=	+ s
v	:=	VECTORADD(**v**,**v**)	or	**v**	:=	**v** + **v**
s	:=	VECTORADD(s,s)	or	s	:=	s + s

<u>Vector Subtraction</u>

v	:=	VECTORMINUS(**v**)	or	**v**	:=	- **v**
s	:=	VECTORMINUS(s)	or	s	:=	- s
v	:=	VECTORDIFFERENCE(**v**,**v**)	or	**v**	:=	**v** - **v**
s	:=	VECTORDIFFERENCE(s,s)	or	s	:=	s - s

<u>Vector Division</u>

v	:=	VECTORRECIP(**v**)	or	**v**	:=	/ **v**
s	:=	VECTORRECIP(s)	or	s	:=	/ s
v	:=	VECTORQUOTIENT(**v**,**v**)	or	**v**	:=	**v** / **v**
v	:=	VECTORQUOTIENT(**v**, s)	or	**v**	:=	**v** / s
v	:=	VECTORQUOTIENT(s ,**v**)	or	**v**	:=	s / **v**
s	:=	VECTORQUOTIENT(s,s)	or	s	:=	s / s

<u>Vector Multiplication</u>

v	:=	VECTORTIMES(s ,**v**)	or	**v**	:=	s * **v**
v	:=	VECTORTIMES(**v**, s)	or	**v**	:=	**v** * s
s	:=	VECTORTIMES(**v**,**v**)	or	s	:=	**v** * **v**
s	:=	VECTORTIMES(s , s)	or	s	:=	s * s

<u>Vector Cross Product</u>

v	:=	VECTORCROSS(**v**,**v**)	or	**v**	:=	**v** × **v**

Vector Exponentiation

s	:=	VECTOREXPT ($\mathbf{v}$, s)	or	s	:=	$\mathbf{v}$ ^ s
s	:=	VECTOREXPT (s , s)	or	s	:=	s ^ s

Vector Modulus

s	:=	VMOD (s)
s	:=	VMOD ($\mathbf{v}$)

All other combinations of operands for these operators lead to error messages being issued. The first two instances of vector multiplication are scalar multiplication of vectors, the third is the product of two scalars and the last is the inner (dot) product. The prefix operators `+, -, /` can take either scalar or vector arguments and return results of the same type as their arguments. VMOD returns a scalar.

In compound expressions, parentheses may be used to specify the order of combination. If parentheses are omitted the ordering of the operators, in increasing order of precedence is

```
+ | - | dotgrad | * | >< | ^ | _
```

and these are placed in the precedence list defined in REDUCE after $<$.

Vector divisions are defined as follows: If $\mathbf{a}$ and $\mathbf{b}$ are vectors and c is a scalar, then

$$\mathbf{a}/\mathbf{b} = \frac{\mathbf{a}\cdot\mathbf{b}}{\mid \mathbf{b} \mid^2}$$
$$c/\mathbf{a} = \frac{c\mathbf{a}}{\mid \mathbf{a} \mid^2}$$

Both scalar multiplication and dot products are given by the same symbol, braces are advisable to ensure the correct precedences in expressions such as $(\mathbf{a}\cdot\mathbf{b})(\mathbf{c}\cdot\mathbf{d})$.

Vector exponentiation is defined as the power of the modulus:
$\mathbf{a}^n \equiv \mathrm{VMOD}(a)^n = \mid \mathbf{a} \mid^n$

60.4 Differential Operations

Differential operators provided are div, grad, curl, delsq, and dotgrad. All but the last of these are prefix operators having a single vector or scalar

s	:=	div ($\mathbf{v}$)
$\mathbf{v}$	:=	grad(s)
$\mathbf{v}$	:=	curl($\mathbf{v}$)
$\mathbf{v}$	:=	delsq($\mathbf{v}$)
s	:=	delsq(s)
$\mathbf{v}$	:=	$\mathbf{v}$ dotgrad $\mathbf{v}$
s	:=	$\mathbf{v}$ dotgrad s

Table 60.1: ORTHOVEC valid combinations of operator and argument

argument as appropriate. Valid combinations of operator and argument, and the type of the result are shown in table 60.1.

All other combinations of operator and argument type cause error messages to be issued. The differential operators have their usual meanings. The coordinate system used by these operators is set by invoking VSTART (cf. Sec. 60.1). The names `h1`, `h2` and `h3` are reserved for the scale factors, and `u1`, `u2` and `u3` are used for the coordinates.

A vector extension, VDF, of the REDUCE procedure DF allows the differentiation of a vector (scalar) with respect to a scalar to be performed. Allowed forms are VDF($\mathbf{v}$, s) $\rightarrow$ $\mathbf{v}$ and VDF(s, s) $\rightarrow$ s , where, for example

$$\mathtt{vdf(B,x)} \equiv \frac{\partial \mathbf{B}}{\partial x}$$

The standard REDUCE procedures DEPEND and NODEPEND have been redefined to allow dependences of vectors to be compactly defined. For example

```
a := svec(a1,a2,a3)$;
depend a,x,y;
```

causes all three components `a1`,`a2` and `a3` of `a` to be treated as functions of `x` and `y`. Individual component dependences can still be defined if desired.

```
depend a3,z;
```

The procedure VTAYLOR gives truncated Taylor series expansions of scalar or vector functions:-

```
vtaylor(vex,vx,vpt,vorder);
```

VEX	VX	VPT	VORDER
v	**v**	**v**	**v**
v	**v**	**v**	s
v	s	s	s
s	**v**	**v**	**v**
s	**v**	**v**	s
s	s	s	s

Table 60.2: ORTHOVEC valid combination of argument types.

returns the series expansion of the expression VEX with respect to variable VX about point VPT to order VORDER. Valid combinations of argument types are shown in table 60.2.

Any other combinations cause error messages to be issued. Elements of VORDER must be non-negative integers, otherwise error messages are issued. If scalar VORDER is given for a vector expansion, expansions in each component are truncated at the same order, VORDER.

The new version of Taylor expansion applies l'Hôpital's rule in evaluating coefficients, so handle cases such as $\sin(x)/(x)$, etc. which the original version of ORTHOVEC could not. The procedure used for this is LIMIT, which can be used directly to find the limit of a scalar function `ex` of variable `x` at point `pt`:-

```
ans := limit(ex,x,pt);
```

60.5 Integral Operations

Definite and indefinite vector, volume and scalar line integration procedures are included in ORTHOVEC. They are defined as follows:

$$\begin{aligned}
\mathrm{VINT}(\mathbf{v},x) &= \int \mathbf{v}(x)dx \\
\mathrm{DVINT}(\mathbf{v},x,a,b) &= \int_a^b \mathbf{v}(x)dx \\
\mathrm{VOLINT}(\mathbf{v}) &= \int \mathbf{v}h_1h_2h_3du_1du_2du_3
\end{aligned}$$

$$
\begin{aligned}
\mathrm{DVOLINT}(\mathbf{v},\mathbf{l},\mathbf{u},n) &= \int_{\mathbf{l}}^{\mathbf{u}} \mathbf{v} h_1 h_2 h_3 du_1 du_2 du_3 \\
\mathrm{LINEINT}(\mathbf{v},\omega,t) &= \int \mathbf{v}\cdot\mathbf{dr} \equiv \int v_i h_i \frac{\partial \omega_i}{\partial t} dt \\
\mathrm{DLINEINT}(\mathbf{v},\omega t,a,b) &= \int_a^b v_i h_i \frac{\partial \omega_i}{\partial t} dt
\end{aligned}
$$

In the vector and volume integrals, $\mathbf{v}$ are vector or scalar, a, b, x and n are scalar. Vectors $\mathbf{l}$ and $\mathbf{u}$ contain expressions for lower and upper bounds to the integrals. The integer index n defines the order in which the integrals over u_1, u_2 and u_3 are performed in order to allow for functional dependencies in the integral bounds:

n	order
1	$u_1\ u_2\ u_3$
2	$u_3\ u_1\ u_2$
3	$u_2\ u_3\ u_1$
4	$u_1\ u_3\ u_2$
5	$u_2\ u_1\ u_3$
otherwise	$u_3\ u_2\ u_1$

The vector ω in the line integral's arguments contain explicit parameterisation of the coordinates u_1, u_2, u_3 of the line $\mathbf{u}(t)$ along which the integral is taken.

Chapter 61

PHYSOP: Operator calculus in quantum theory

Mathias Warns
Physikalisches Institut der Universität Bonn
Endenicher Allee 11–13
D–5300 BONN 1, Germany
e–mail: UNP008@DBNRHRZ1.bitnet

The package PHYSOP has been designed to meet the requirements of theoretical physicists looking for a computer algebra tool to perform complicated calculations in quantum theory with expressions containing operators. These operations consist mainly in the calculation of commutators between operator expressions and in the evaluations of operator matrix elements in some abstract space.

61.1 The NONCOM2 Package

The package NONCOM2 redefines some standard REDUCE routines in order to modify the way noncommutative operators are handled by the system. It redefines the `NONCOM` statement in a way more suitable for calculations in physics. Operators have now to be declared noncommutative pairwise, *i.e.* coding:

```
NONCOM A,B;
```

declares the operators `A` and `B` to be noncommutative but allows them to commute with any other (noncommutative or not) operator present in the expression. In a similar way if one wants *e.g.* `A(X)` and `A(Y)` not to commute, one has now to code:

```
NONCOM A,A;
```

A final example should make the use of the redefined `NONCOM` statement clear:

```
NONCOM A,B,C;
```

declares `A` to be noncommutative with `B` and `C`, `B` to be noncommutative with `A` and `C` and `C` to be noncommutative with `A` and `B`. Note that after these declaration *e.g.* `A(X)` and `A(Y)` are still commuting kernels.

Finally to keep the compatibility with standard REDUCE declaring a single identifier using the `NONCOM` statement has the same effect as in standard REDUCE.

From the user's point of view there are no other new commands implemented by the package.

61.2 The PHYSOP package

The package PHYSOP implements a new REDUCE data type to perform calculations with physical operators. The noncommutativity of operators is implemented using the NONCOM2 package so this file should be loaded prior to the use of PHYSOP.

61.2.1 Type declaration commands

The new REDUCE data type PHYSOP implemented by the package allows the definition of a new kind of operators (*i.e.* kernels carrying an arbitrary number of arguments). Throughout this manual, the name "operator" will refer, unless explicitly stated otherwise, to this new data type. This data type is in turn divided into 5 subtypes. For each of this subtype, a declaration command has been defined:

`SCALOP A;` declares `A` to be a scalar operator. This operator may carry an arbitrary number of arguments; after the declaration: `SCALOP A;` all kernels of the form `A(J)`, `A(1,N)`, `A(N,L,M)` are recognised by the system as being scalar operators.

`VECOP V;` declares `V` to be a vector operator. As for scalar operators, the vector operators may carry an arbitrary number of arguments. For example `V(3)` can be used to represent the vector operator $\vec{V}_3$. Note that the dimension of space in which this operator lives is arbitrary. One can however address a specific component of the vector operator by using a special index declared as `PHYSINDEX` (see below). This index must then be the first in the argument list of the vector operator.

`TENSOP C(3);` declares `C` to be a tensor operator of rank 3. Tensor operators of any fixed integer rank larger than 1 can be declared. Again this operator may carry an arbitrary number of arguments and the space dimension is not fixed. The tensor components can be addressed by using special `PHYSINDEX` indices (see below) which have to be placed in front of all other arguments in the argument list.

`STATE U;` declares `U` to be a state, *i.e.* an object on which operators have a certain action. The state U can also carry an arbitrary number of arguments.

`PHYSINDEX X;` declares `X` to be a special index which will be used to address components of vector and tensor operators.

A command `CLEARPHYSOP` removes the PHYSOP type from an identifier in order to use it for subsequent calculations. However it should be remembered that no substitution rule is cleared by this function. It is therefore left to the user's responsibility to clear previously all substitution rules involving the identifier from which the PHYSOP type is removed.

61.2.2 Ordering of operators in an expression

The ordering of kernels in an expression is performed according to the following rules:

1. Scalars are always ordered ahead of PHYSOP operators in an expression. The REDUCE statement `KORDER` can be used to control the ordering of scalars but has no effect on the ordering of operators.

2. The default ordering of operators follows the order in which they have been declared (not the alphabetical one). This ordering scheme can be changed using the command `OPORDER`. Its syntax is similar to the `KORDER` statement, *i.e.* coding: `OPORDER A,V,F;` means that all occurrences of the operator `A` are ordered ahead of those of `V` etc. It is also possible to include operators carrying indices (both normal and special ones) in the argument list of `OPORDER`. However including objects not defined as operators (*i.e.* scalars or indices) in the argument list of the `OPORDER` command leads to an error.

3. Adjoint operators are placed by the declaration commands just after the original operators on the `OPORDER` list. Changing the place of an operator on this list means not that the adjoint operator is moved accordingly. This adjoint operator can be moved freely by including it in the argument list of the `OPORDER` command.

61.2.3 Arithmetic operations on operators

The following arithmetic operations are possible with operator expressions:

1. Multiplication or division of an operator by a scalar.

2. Addition and subtraction of operators of the same type.

3. Multiplication of operators is only defined between two scalar operators.

4. The scalar product of two VECTOR operators is implemented with a new function `DOT`. The system expands the product of two vector operators into an ordinary product of the components of these operators by inserting a special index generated by the program. To give an example, if one codes:

```
VECOP V,W;
V DOT W;
```

the system will transform the product into:

```
V(IDX1) * W(IDX1)
```

where `IDX1` is a `PHYSINDEX` generated by the system (called a DUMMY

INDEX in the following) to express the summation over the components. The identifiers `IDXn` (`n` is a nonzero integer) are reserved variables for this purpose and should not be used for other applications. The arithmetic operator `DOT` can be used both in infix and prefix form with two arguments.

5. Operators (but not states) can only be raised to an integer power. The system expands this power expression into a product of the corresponding number of terms inserting dummy indices if necessary. The following examples explain the transformations occurring on power expressions (system output is indicated with an `-->`):

```
SCALOP A; A**2;
  --> A*A
VECOP V; V**4;
  --> V(IDX1)*V(IDX1)*V(IDX2)*V(IDX2)
TENSOP C(2); C**2;
  --> C(IDX3,IDX4)*C(IDX3,IDX4)
```

Note in particular the way how the system interprets powers of tensor operators which is different from the notation used in matrix algebra.

6. Quotients of operators are only defined between scalar operator expressions. The system transforms the quotient of 2 scalar operators into the product of the first operator times the inverse of the second one.

```
SCALOP A,B;   A / B;
            -1
      A *( B   )
```

7. Combining the last 2 rules explains the way how the system handles negative powers of operators:

```
SCALOP B;
B**(-3);
        -1     -1     -1
  --> (B   )*(B   )*(B   )
```

The method of inserting dummy indices and expanding powers of operators has been chosen to facilitate the handling of complicated operator expressions and particularly their application on states. However it may be useful to get rid of these dummy indices in order to enhance the readability of the system's final output. For this purpose the switch `CONTRACT` has to be turned on (`CONTRACT` is normally set to `OFF`). The system in this case contracts over dummy indices reinserting the DOT operator and reassembling the expanded powers. However due to the predefined operator ordering the system may not remove all the dummy indices introduced previously.

61.2.4 Special functions

Commutation relations

If two PHYSOPs have been declared noncommutative using the (redefined) NONCOM statement, it is possible to introduce in the environment elementary (anti-) commutation relations between them. For this purpose, two scalar operators COMM and ANTICOMM are available. These operators are used in conjunction with LET statements. Example:

```
SCALOP A,B,C,D;
LET COMM(A,B)=C;
FOR ALL N,M LET ANTICOMM(A(N),B(M))=D;
VECOP U,V,W; PHYSINDEX X,Y,Z;
FOR ALL X,Y LET COMM(V(X),W(Y))=U(Z);
```

Note that if special indices are used as dummy variables in FOR ALL ... LET constructs then these indices should have been declared previously using the PHYSINDEX command.

Every time the system encounters a product term involving two noncommutative operators which have to be reordered on account of the given operator ordering, the list of available (anti-) commutators is checked in the following way: First the system looks for a commutation relation which matches the product term. If it fails then the defined anticommutation relations are checked. If there is no successful match the product term A*B is replaced by:

```
A*B;
 --> COMM(A,B) + B*A
```

so that the user may introduce the commutation relation later on.

The user may want to force the system to look for anticommutators only; for this purpose a switch ANTICOM is defined which has to be turned on (ANTICOM is normally set to OFF). In this case, the above example is replaced by:

```
ON ANTICOM;
A*B;
 -->  ANTICOMM(A,B) - B*A
```

For the calculation of (anti-) commutators between complex operator expressions, the functions `COMMUTE` and `ANTICOMMUTE` have been defined.

```
VECOP P,A,K;
PHYSINDEX X,Y;
FOR ALL X,Y LET COMM(P(X),A(Y))=K(X)*A(Y);
COMMUTE(P**2,P DOT A);
```

Adjoint expressions

As has been already mentioned, for each operator and state defined using the declaration commands, the system generates automatically the corresponding adjoint operator. For the calculation of the adjoint representation of a complicated operator expression, a function `ADJ` has been defined.

```
SCALOP A,B;
ADJ(A*B);
        +    +
  --> (A )*(B )
```

Application of operators on states

A function `OPAPPLY` has been defined for the application of operators to states. It has two arguments and is used in the following combinations:

(i) `LET OPAPPLY(`*operator, state*`) =` *state*; This is to define a elementary action of an operator on a state in analogy to the way elementary commutation relations are introduced to the system.

```
SCALOP A; STATE U;
FOR ALL N,P LET OPAPPLY((A(N),U(P))= EXP(I*N*P)*U(P);
```

(ii) `LET OPAPPLY(`*state, state*`) =` *scalar exp.*; This form is to define scalar products between states and normalisation conditions.

```
STATE U;
```

```
FOR ALL N,M LET OPAPPLY(U(N),U(M)) = IF N=M THEN 1 ELSE 0;
```

(iii) *state* `:=` `OPAPPLY(`*operator expression, state*`)`; In this way, the action of an operator expression on a given state is calculated using elementary relations defined as explained in **(i)**. The result may be assigned to a different state vector.

(iv) `OPAPPLY(`*state*`, OPAPPLY(`*operator expression, state*`))`; This is the way how to calculate matrix elements of operator expressions. The system proceeds in the following way: first the rightmost operator is applied on the right state, which means that the system tries to find an elementary relation which match the application of the operator on the state. If it fails the system tries to apply the leftmost operator of the expression on the left state using the adjoint representations. If this fails also, the system prints out a warning message and stops the evaluation. Otherwise the next operator occuring in the expression is taken and so on until the complete expression is applied. Then the system looks for a relation expressing the scalar product of the two resulting states and prints out the final result. An example of such a calculation is given in the test file.

The infix version of the `OPAPPLY` function is the vertical bar $|$. It is right associative and placed in the precedence list just above the minus $(-)$ operator.

Chapter 62

PM: A REDUCE pattern matcher

Kevin McIsaac
The University of Western Australia
Australia
e–mail: kevin@wri.com

PM is a general pattern matcher similar in style to those found in systems such as SMP and Mathematica.

A template is any expression composed of literal elements (*e.g.* `5`, `a` or `a+1`) and specially denoted pattern variables (*e.g.* `?a` or `??b`). Atoms beginning with '?' are called generic variables and match any expression. Atoms beginning with '??' are called multi-generic variables and match any expression or any sequence of expressions including the null or empty sequence. A sequence is an expression of the form '[a1, a2,...]'. When placed in a function argument list the brackets are removed, *i.e.* f([a,1]) $\rightarrow$ f(a,1) and f(a,[1,2],b) $\rightarrow$ f(a,1,2,b).

A template is said to match an expression if the template is literally equal to the expression or if by replacing any of the generic or multi-generic symbols occurring in the template, the template can be made to be literally equal to the expression. These replacements are called the bindings for the generic variables. A replacement is an expression of the form `exp1 -> exp2`, which means exp1 is replaced by exp2, or `exp1 --> exp2`, which is the same except exp2 is not simplified until after the substitution for exp1 is made. If the expression has any of the properties; associativity, commutativity, or an identity element, they are used to determine if the expressions match. If an attempt to match the template to the expression fails the matcher backtracks, unbinding generic variables, until it reached a place were it can make a different choice.

The matcher also supports semantic matching. Briefly, if a subtemplate does not match the corresponding subexpression because they have different structures then the two are equated and the matcher continues matching the rest of the expression until all the generic variables in the subexpression are bound. The equality is then checked. This is controlled by the switch `semantic`. By default it is on.

62.1 The Match Function

`M(exp,template)`

The template is matched against the expression. If the template is literally equal to the expression `T` is returned. If the template is literally equal to the expression after replacing the generic variables by their bindings then the set of bindings is returned as a set of replacements. Otherwise `NIL` is returned.

```
OPERATOR F;

M(F(A),F(A));

T

M(F(A,B),F(A,?A));

{?A->B}

M(F(A,B),F(??A));

{??A->[A,B]}

m(a+b+c,c+?a+?b);

{?a->a,?b->b}

m(a+b+c,b+?a);
```

```
{?a->a + c}
```

This example shows the effects of semantic matching, using the associativity and commutativity of `+`.

62.2 Qualified Matching

A template may be qualified by the use of the conditional operator `_=',` standing for **such that**. When a such-that condition is encountered in a template it is held until all generic variables appearing in logical-exp are bound. On the binding of the last generic variable logical-exp is simplified and if the result is not `T` the condition fails and the pattern matcher backtracks. When the template has been fully parsed any remaining held such-that conditions are evaluated and compared to `T`.

```
load_package pm;

operator f;

if (m(f(a,b),f(?a,?b_=(?a=?b)))) then write "yes" else write"no";

no

m(f(a,a),f(?a,?b_=(?a=?b)));

{?B->A,?A->A}
```

62.3 Substituting for replacements

The operator `S` substitutes the replacements in an expression.

`S(exp,temp1->sub1,temp2->sub2,...,rept, depth);`

will do the substitutions for a maximum of `rept` and to a depth of `depth`, using a breadth-first search and replace. `rept` and `depth` may be omitted when they default to 1 and infinity.

`SI(exp,temp1->sub1,temp2->sub2,..., depth)`

will substitute infinitely many times until expression stops changing.

`SD(exp,temp1->sub1,temp2->sub2,...,rept, depth)`

is a depth-first version of `S`.

```
s(f(a,b),f(a,?b)->?b^2);

 2
b

s(a+b,a+b->a*b);

a*b


operator nfac;

s(nfac(3),{nfac(0)->1,nfac(?x)->?x*nfac(?x-1)});

3*nfac(2)

s(nfac(3),{nfac(0)->1,nfac(?x)->?x*nfac(?x-1)},2);

6*nfac(1)

si(nfac(4),{nfac(0)->1,nfac(?x)->?x*nfac(?x-1)});

24

s(a+b+f(a+b),a+b->a*b,inf,0);

f(a + b) + a*b
```

62.4 Programming with Patterns

There are also facilities to use this pattern-matcher as a programming language. The operator `:-` can be used to declare that while simplifying all matches of a template should be replaced by some expression. The operator `::-` is the same except that the left hand side is not simplified.

```
operator fac, gamma;
```

```
fac(?x_=Natp(?x)) ::- ?x*fac(?x-1);

HOLD(FAC(?X-1)*?X)

fac(0)  :- 1;

1

fac(?x) :- Gamma(?x+1);

GAMMA(?X + 1)

fac(3);

6

fac(3/2);

GAMMA(5/2)
```

Chapter 63

QSUM : Package for q-hypergeometric sums

Harald Böing
Wolfram Koepf
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraše 7
D-14195 Berlin-Dahlem
e-mail: koepf@zib.de

This package is an implementation of the q-analogues of Gosper's and Zeilberger's [1] algorithm for indefinite and definite summation of q-hypergeometric terms, respectively.

An expression a_k is called a *q-hypergeometric term*, if a_k/a_{k-1} is a rational function with respect to q^k. Most q-terms are based on the *q-shifted factorial* or *qpochhammer*. Other typical q-hypergeometric terms are ratios of products of powers, q-factorials, q-binomial coefficients, and q-shifted factorials that are integer-linear in their arguments.
The package is loaded with `load_package qsum`.

63.1 Elementary q-Functions

The package supports the input of the following elementary q-functions:

[1]The `ZEILBERG` package (Chap. 89 p. 599, see also [13]) contains the hypergeometric versions.

- `qpochhammer(a,q,infinity)`

$$(a;\, q)_\infty := \prod_{j=0}^{\infty} \left(1 - a\, q^j\right)$$

- `qpochhammer(a,q,k)`

$$(a;\, q)_k := \begin{cases} \prod_{j=0}^{k-1} \left(1 - a\, q^j\right) & \text{if } k > 0 \\ 1 & \text{if } k = 0 \\ \prod_{j=1}^{k} \left(1 - a\, q^{-j}\right)^{-1} & \text{if } k < 0 \end{cases}$$

- `qbrackets(k,q)`

$$[q, k] := \frac{q^k - 1}{q - 1}$$

- `qfactorial(k,q)`

$$[k]_q! := \frac{(q;\, q)_k}{(1-q)^k}$$

- `qbinomial(n,k,q)`

$$\binom{n}{k}_q := \frac{(q;\, q)_n}{(q;\, q)_k \cdot (q;\, q)_{n-k}}$$

- `qphihyperterm({a1,a2,...,ar},{b1,b2,...,bs},q,z,k)`

$$\sum_{k=0}^{\infty} \frac{(a_1, a_2, \ldots, a_r;\, q)_k}{(b_1, b_2, \ldots, b_s;\, q)_k} \frac{z^k}{(q;\, q)_k} \left[(-1)^k\, q^{\binom{k}{2}}\right]^{1+s-r}$$

- `qpsihyperterm({a1,a2,...,ar},{b1,b2,...,bs},q,z,k)`

$$\sum_{k=-\infty}^{\infty} \frac{(a_1, a_2, \ldots, a_r;\, q)_k}{(b_1, b_2, \ldots, b_s;\, q)_k} z^k \left[(-1)^k\, q^{\binom{k}{2}}\right]^{s-r}$$

where $(a_1, a_2, \ldots, a_r;\, q)_k$ stands for the product $\prod_{j=1}^{r} (a_j;\, q)_k$.

63.2 The QGOSPER operator

The `qgosper` operator is an implementation of the q-Gosper algorithm [14].

- `qgosper(a,q,k)` determines a q-hypergeometric antidifference. (By default it returns a *downward* antidifference, which may be changed by the switch `qgosper_down`.) If it does not return a q-hypergeometric antidifference, then such an antidifference does not exist.
- `qgosper(a,q,k,m,n)` determines a closed formula for the definite sum

$$\sum_{k=m}^{n} a_k$$

using the q-analogue of Gosper's algorithm. This is only successful if q-Gosper's algorithm applies.

Example:

```
1: qgosper(qpochhammer(a,q,k)*q^k/qpochhammer(q,q,k),q,k);

   k
 (q *a - 1)*qpochhammer(a,q,k)
-------------------------------
  (a - 1)*qpochhammer(q,q,k)
```

63.3 The QSUMRECURSION operator

The `QSUMRECURSION` operator is an implementation of the q-Zeilberger algorithm [14]. It tries to determine a homogeneous recurrence equation for $summ(n)$ wrt. n with polynomial coefficients (in n), where

$$summ(n) := \sum_{k=-\infty}^{\infty} \mathrm{f}(n,k).$$

There are three different ways to pass a summand $\mathrm{f}(n,k)$ to `qsumrecursion`:

- `qsumrecursion(f,q,k,n)`, where `f` is a q-hypergeometric term wrt. `k` and `n`, `k` is the summation variable and `n` the recursion variable, `q` is a symbol.

- `qsumrecursion(upper,lower,q,z,n)` is a shortcut for `qsumrecursion(qphihyperterm(upper,lower,q,z,k),q,k,n)`
- `qsumrecursion(f,upper,lower,q,z,n)` is a similar shortcut for `qsumrecursion(f*qphihyperterm(upper,lower,q,z,k),q,k,n)`,

i. e. `upper` and `lower` are lists of upper and lower parameters of the generalized q-hypergeometric function. The third form is handy if you have any additional factors.

For all three instances it is possible to pass the order, if known in advance, as additional argument at the end of the parameter sequence. You can also specifiy a range by a list of two positive integers, the first one specifying the lowest and the second one the highest order. By default `QSUMRECURSION` will search for recurrences of order from 1 to 5. Usually it uses `summ` as name for the summ-function. If you want to change this behaviour then use the following syntax: `QSUMRECURSION(f,q,k,s(n))`.

```
2: qsumrecursion(qpochhammer(q^(-n),q,k)*z^k /
                 qpochhammer(q,q,k),q,k,n);

      n                      n
- ((q   - z)*summ(n - 1) - q *summ(n))
```

63.4 Global Variables and Switches

There are several switches defined in the `QSUM` package. Please take a look in the accompanying documentation file `qsum.tex` in $REDUCEPATH/packages/.

The most important switches are:

- `qgosper_down`, default setting is on. It determines whether `qgosper` returns a downward or an upward antidifference g_k for the input term a_k, .e. $a_k = g_k - g_{k-1}$ or $a_k = g_{k+1} - g_k$ respectively.
- `qsumrecursion_certificate`, default off. As Zeilberger's algorithm delivers a recurrence equation for a q-hypergeometric term f(n, k) this switch is used to get all necessary informations for proving this recurrence equation.

If it is set on, instead of simply returning the resulting recurrence equation (for the sum)—if one exists—`qsumrecursion` returns a list `{rec,cert,f,k,dir}` with five items: The first entry contains the recurrence equation, while the other items enable you to prove the recurrence a posteriori by rational arithmetic.

If we denote by `r` the recurrence `rec` where we substituted the `summ`-function by the input term `f` (with the corresponding shifts in `n`) then the following equation is valid:

```
r = cert*f - sub(k=k-1,cert*f)
```

or

```
r = sub(k=k+1,cert*f) - cert*f
```

if `dir=downward_antidifference` or `dir=upward_antidifference` respectively.

There is one global variable:

- `qsumrecursion_recrange!*` controls for which recursion orders the procedure `qsumrecursion` looks. It has to be a list with two entries, the first one representing the lowest and the second one the highest order of a recursion to search for. By default it is set to `{1,5}`.

Chapter 64

RANDPOLY: A random polynomial generator

Francis J. Wright
School of Mathematical Sciences, Queen Mary and Westfield College
University of London
Mile End Road
London E1 4NS, England
e–mail: F.J.Wright@QMW.ac.uk

The operator `RANDPOLY` requires at least one argument corresponding to the polynomial variable or variables, which must be either a single expression or a list of expressions. In effect, `RANDPOLY` replaces each input expression by an internal variable and then substitutes the input expression for the internal variable in the generated polynomial (and by default expands the result as usual). The rest of this document uses the term "variable" to refer to a general input expression or the internal variable used to represent it, and all references to the polynomial structure, such as its degree, are with respect to these internal variables. The actual degree of a generated polynomial might be different from its degree in the internal variables.

By default, the polynomial generated has degree 5 and contains 6 terms. Therefore, if it is univariate it is dense whereas if it is multivariate it is sparse.

64.1 Optional arguments

Other arguments can optionally be specified, in any order, after the first compulsory variable argument. All arguments receive full algebraic evaluation, subject to the current switch settings etc. The arguments are processed in the order given, so that if more than one argument relates to the same property then the last one specified takes effect. Optional arguments are either keywords or equations with keywords on the left.

In general, the polynomial is sparse by default, unless the keyword `dense` is specified as an optional argument. (The keyword `sparse` is also accepted, but is the default.) The default degree can be changed by specifying an optional argument of the form

`degree =` *natural number*.

In the multivariate case this is the total degree, *i.e.* the sum of the degrees with respect to the individual variables. More complicated monomial degree bounds can be constructed by using the coefficient function described below to return a monomial or polynomial coefficient expression. Moreover, `randpoly` respects internally the REDUCE "asymptotic" commands `let`, `weight` *etc.* described in section 10.4, which can be used to exercise additional control over the polynomial generated.

In the sparse case (only), the default maximum number of terms generated can be changed by specifying an optional argument of the form

`terms =` *natural number*.

The actual number of terms generated will be the minimum of the value of `terms` and the number of terms in a dense polynomial of the specified degree, number of variables, *etc.*

64.2 Advanced use of RANDPOLY

The default order (or minimum or trailing degree) can be changed by specifying an optional argument of the form

`ord =` *natural number*.

The order normally defaults to 0.

The input expressions to `randpoly` can also be equations, in which case the order defaults to 1 rather than 0. Input equations are converted to the difference of their two sides before being substituted into the generated polynomial. This makes it easy to generate polynomials with a specified zero – for example

```
randpoly(x = a);
```

generates a polynomial that is guaranteed to vanish at $x = a$, but is otherwise random.

The operator `randpoly` accepts two further optional arguments in the form of equations with the keywords `coeffs` and `expons` on the left. The right sides of each of these equations must evaluate to objects that can be applied as functions of no variables. These functions should be normal algebraic procedures; the `coeffs` procedure may return any algebraic expression, but the `expons` procedure must return an integer. The values returned by the functions should normally be random, because it is the randomness of the coefficients and, in the sparse case, of the exponents that makes the constructed polynomial random.

A convenient special case is to use the function `rand` on the right of one or both of these equations; when called with a single argument `rand` returns an anonymous function of no variables that generates a random integer. The single argument of `rand` should normally be an integer range in the form a .. b, where a, b are integers such that $a < b$. For example, the `expons` argument might take the form

```
expons = rand(0 .. n)
```

where `n` will be the maximum degree with respect to each variable *independently*. In the case of `coeffs` the lower limit will often be the negative of the upper limit to give a balanced coefficient range, so that the `coeffs` argument might take the form

```
coeffs = rand(-n .. n)
```

which will generate random integer coefficients in the range $[-n, n]$.

Further information on the the auxiliary functions of RANDPOLY can be found in the extended documentation and examples.

64.3 Examples

```
randpoly(x);

       5       4       3       2
 - 54*x  - 92*x  - 30*x  + 73*x  - 69*x - 67


randpoly({x, y}, terms = 20);

    5       4         4       3  2       3         3
31*x  - 17*x *y - 48*x  - 15*x *y  + 80*x *y + 92*x

       2  3      2         2         4         3         2
 + 86*x *y  + 2*x *y - 44*x  + 83*x*y  + 85*x*y  + 55*x*y

                       5       4      3       2
 - 27*x*y + 33*x - 98*y  + 51*y  - 2*y  + 70*y  - 60*y - 10
```

```
randpoly({x, sin(x), cos(x)});

                   4            3              3
sin(x)*( - 4*cos(x)  - 85*cos(x) *x + 50*sin(x)

                    2
         - 20*sin(x) *x + 76*sin(x)*x + 96*sin(x))
```

Chapter 65

RATAPRX : Rational Approximations Package

Lisa Temme
Wolfram Koepf
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraše 7
D-14195 Berlin-Dahlem, Germany
e-mail: koepf@zib.de

This package provides functions to

- convert rational numbers in their periodic representation and vice versa,
- to compute continued fractions and
- to compute the Padé approximant of a function.

The package can be loaded using `load_package rataprx;` it supersedes the `contfr` package.

65.1

65.1.1 Periodic Representation

The function `rational2periodic(n)` converts a rational number `n` in its periodic representation. For example 59/70 is converted to $0.8\overline{428571}$.
Depending on the print function of your REDUCE system, calling the function `rational2periodic` might result in an expression of the form `periodic({a,b},{c`$_1$`,...,c`$_n$`})`. `a` and `b` is the non-periodic part of the rational number `n` and `c`$_1$`,...,c`$_n$ are the digits of the periodic part. In this case 59/70 would result in `periodic({8,10},{4,2,8,5,7,1})`.
The function `periodic2rational(periodic({a,b},{c`$_1$`,...,c`$_n$`}))` is the inverse function and computes the rational expression for a periodic one. Note that `b` is 1,-1 or a integer multiple of 10. If `a` is zero, then the input number `b` indicates how many places after the decimal point the period occurs.

```
rational2periodic(6/17);

periodic({0,1},{3,5,2,9,4,1,1,7,6,4,7,0,5,8,8,2})

periodic2rational(ws);

 6
----
 17
```

65.1.2 Continued Fractions

A continued fraction (see [1] §4.2) has the general form

$$b_0 + \frac{a_1}{b_1 + \frac{a_2}{b_2 + \frac{a_3}{b_3 + \ldots}}} .$$

A more compact way of writing this is as

$$b_0 + \frac{a_1|}{|b_1} + \frac{a_2|}{|b_2} + \frac{a_3|}{|b_3} + \ldots .$$

This is represented in REDUCE as

$$\texttt{contfrac}(Rational\quad approximant, \{\mathtt{b}_0, \{\mathtt{a}_1, \mathtt{b}_1\}, \{\mathtt{a}_2, \mathtt{b}_2\}, ...\}).$$

There are four different functions to determine the continued fractions for real numbers and functions `f` in the variable `var`:

```
cfrac(number);  cfrac(number,length);
cfrac(f, var);  cfrac(f, var, length);
```

The `length` argument is optional and specifies the number of ordered pairs $\{a_i, b_i\}$ to be returned. It's default value is five.

```
cfrac pi;

          1146408
contfrac(---------),
          364913

         {3,{1,7},{1,15},{1,1},{1,292},{1,1},{1,1},{1,1},
            {1,2},{1,1}})
```

```
cfrac((x+2/3)^2/(6*x-5),x);

              2
          9*x  + 12*x + 4     6*x + 13      24*x - 20
contfrac(-----------------,{----------,{1,-----------}})
             54*x - 45          36             9

cfrac(e^x,x);

            3      2
          x  + 9*x  + 36*x + 60
contfrac(-----------------------,
               2
            3*x  - 24*x + 60

         {1,{x,1},{ - x,2},{x,3},{ - x,2},{x,5}})
```

65.1.3 Padé Approximation

The Padé approximant represents a function by the ratio of two polynomials. The coefficients of the powers occuring in the polynomials are determined by the coefficients in the Taylor series expansion of the function (see [1]). Given a power series

$$f(x) = c_0 + c_1(x-h) + c_2(x-h)^2 \ldots$$

and the degree of numerator, n, and of the denominator, d, the `pade` function finds the unique coefficients a_i, b_i in the Padé approximant

$$\frac{a_0 + a_1 x + \cdots + a_n x^n}{b_0 + b_1 x + \cdots + b_d x^d} .$$

The function `pade(f, x, h ,n ,d)` takes as input the function `f` in the variable `x` to be approximated , where `h` is the point at which the approximation is evaluated. `n` and `d` are the (specified) degrees of the numerator and the denominator. It returns the Padé Approximant, ie. a rational function.

Error Messages may occur in the following different cases:

- The Taylor series expansion for the function `f` has not yet been implemented in the REDUCE Taylor Package.
- A Padé Approximant of this function does not exist.

- A Padé Approximant of this order (ie. the specified numerator and denominator orders) does not exist. Please note, there might exist an approximant of a different order.

```
pade(sin(x),x,0,3,3);

          2
 x*( - 7*x  + 60)
------------------
       2
   3*(x  + 20)

pade(tanh(x),x,0,5,5);

     4        2
 x*(x  + 105*x  + 945)
-----------------------
      4       2
 15*(x  + 28*x  + 63)

pade(exp(1/x),x,0,5,5);

***** no Pade Approximation exists

pade(factorial(x),x,1,3,3);

***** not yet implemented

30: pade(sin(x)/x^2,x,0,10,0);

***** Pade Approximation of this order does not exist

31:  pade(sin(x)/x^2,x,0,10,2);

     10        8         6           4            2
  - x   + 110*x  - 7920*x  + 332640*x  - 6652800*x + 39916800
--------------------------------------------------------------
                          39916800*x
```

Chapter 66

REACTEQN: Support for chemical reaction equations

Herbert Melenk
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany
e–mail: melenk@zib.de

The REDUCE package REACTEQN allows one to transform chemical reaction systems into ordinary differential equation systems corresponding to the laws of pure mass action.

It provides the single function

```
reac2ode {   <reaction> [,<rate> [,<rate>]]
           [,<reaction> [,<rate> [,<rate>]]]
                 ....
              };
```

A rate is any REDUCE expression, and two rates are applicable only for forward and backward reactions. A reaction is coded as a linear sum of the series variables, with the operator $->$ for forward reactions and $<>$ for two-way reactions.

The result is a system of explicit ordinary differential equations with polynomial righthand sides. As side effect the following variables are set:

`rates` A list of the rates in the system.

`species` A list of the species in the system.

`inputmat` A matrix of the input coefficients.

`outputmat` A matrix of the output coefficients.

In the matrices the row number corresponds to the input reaction number, while the column number corresponds to the species index.

If the rates are numerical values, it will be in most cases appropriate to select a REDUCE evaluation mode for floating point numbers.

`Inputmat` and `outputmat` can be used for linear algebra type investigations of the reaction system. The classical reaction matrix is the difference of these matrices; however, the two matrices contain more information than their differences because the appearance of a species on both sides is not reflected by the reaction matrix.

Chapter 67

REDLOG: Logic System

Andreas Dolzmann
Thomas Sturm
University of Passau, Germany
e-mail: dolzmann@uni-passau.de, sturm@uni-passau.de

67.1 Introduction

This package extends REDUCE to a computer logic system implementing symbolic algorithms on first-order formulas wrt. temporarily fixed first-order languages and theories.

67.1.1 Contexts

REDLOG is designed for working with several languages and theories in the sense of first-order logic. Both a language and a theory make up a context. There are the following contexts available:

OFSF OF stands for *ordered fields*, which is a little imprecise. The quantifier elimination actually requires the more restricted class of *real closed fields*, while most of the tool-like algorithms are generally correct for ordered fields. One usually has in mind real numbers with ordering when using OFSF.

DVFSF *Discretely valued fields.* This is for computing with formulas over classes of p-adic valued extension fields of the rationals, usually the fields of p-adic numbers for some prime p.

ACFSF *Algebraically closed fields* such as the complex numbers.

67.1.2 Overview

REDLOG origins from the implementation of quantifier elimination procedures. Successfully applying such methods to both academic and real-world problems, the authors have developed over the time a large set of formula-manipulating tools, many of which are meanwhile interesting in their own right:

- Numerous tools for comfortably inputing, decomposing, and analyzing formulas.
- Several techniques for the *simplification* of formulas.
- Various *normal form computations.* The CNF/DNF computation includes both Boolean and algebraic simplification strategies. The *prenex normal form* computation minimizes the number of quantifier changes.
- *Quantifier elimination* computes quantifier-free equivalents for given first-order formulas. For OFSF and DVFSF the formulas have to obey certain degree restrictions.
- The context OFSF allows a variant of quantifier elimination called *generic quantifier elimination*: There are certain non-degeneracy assumptions made on the parameters, which considerably speeds up the elimination.
- The contexts OFSF and DVFSF provide variants of (generic) quantifier elimination that additionally compute *answers* such as satisfying sample points for existentially quantified formulas.
- OFSF includes linear *optimization* techniques based on quantifier elimination.

To avoid ambiguities with other packages, all REDLOG functions and switches are prefixed by "`RL`".

The package is loaded by typing: `load_package redlog;`

It is recommended to read the documentation which comes with this package. This manual chapter gives an overview on the features of REDLOG, which is by no means complete.

67.2 Context Selection

The context to be used has to be selected explicitly. One way to do this is using the command `RLSET`. As argument it takes one of the valid choices `ACFSF` (algebraically closed fields standard form), `OFSF` (ordered fields standard form), and `DVFSF` (discretely valued fields standard form). By default, `DVFSF` computes uniformly over the class of all p-adic valued fields. For the sake of efficiency, this can be restricted by means of an extra `RLSET` argument. `RLSET` returns the old setting as a list.

67.3 Format and Handling of Formulas

67.3.1 First-order Operators

REDLOG knows the following operators for constructing Boolean combinations and quantifications of atomic formulas:

`NOT`: Unary	`AND`: N-ary Infix	`OR`: N-ary Infix	`IMPL`: Binary Infix
`REPL`: Binary Infix	`EQUIV`: Binary Infix	`EX`: Binary	
`ALL`: Binary	`TRUE`: Variable	`FALSE`: Variable	

The `EX` and the `ALL` operators are the quantifiers. Their first argument is the quantified variable, the second one a matrix formula.

There are operators `MKAND` and `MKOR` for the construction of large systematic conjunctions/disjunctions via for loops available. They are used in the style of `SUM` and `COLLECT`.

Example:

```
1: load_package redlog;

2: rlset ofsf;

{}

3: g := for i:=1:3 mkand

          for j:=1:3 mkor

            if j<>i then mkid(x,i) + mkid(x,j)=0;

true and (false or false or x1 + x2 = 0 or x1 + x3 = 0)

 and (false or x1 + x2 = 0 or false or x2 + x3 = 0)

 and (false or x1 + x3 = 0 or x2 + x3 = 0 or false)
```

67.3.2 OFSF Operators

The `OFSF` context implements *ordered fields* over the language of *ordered rings*. There are the following binary operators available:

`EQUAL   NEQ   LEQ   GEQ   LESSP   GREATERP`

They can also be written as `=`, `<>`, `<=`, `>=`, `<`, and `>`. For OFSF there is specified that all right hand sides must be zero. Non-zero right hand sides are immediately subtracted.

67.3.3 DVFSF Operators

Discretely valued fields are implemented as a one-sorted language using in addition to `=` and `<>` the binary operators `|`, `||`, `~`, and `/~`, which encode $\leq$, $<$, $=$, and $\neq$ in the value group, respectively.

`EQUAL   NEQ   DIV   SDIV   ASSOC   NASSOC`

67.3.4 ACFSF Operators

For algebraically closed fields there are only equations and inequalities allowed:

`EQUAL` `NEQ`

As in OFSF, they can be conveniently written as `=` and `<>`, respectively. All right hand sides are zero.

67.3.5 Extended Built-in Commands

The operators `SUB`, `PART`, and `LENGTH` work on formulas in a reasonable way.

67.3.6 Global Switches

The switch `RLSIMPL` causes the function `RLSIMPL` to be automatically applied at the expression evaluation stage.

The switch `RLREALTIME` protocols the wall clock time needed for REDLOG commands in seconds.

The switch `RLVERBOSE` toggles verbosity output with some REDLOG procedures.

67.4 Simplification

REDLOG knows three types of simplifiers to reduce the size of a given first-order formula: the standard simplifier, tableau simplifiers, and Gröbner simplifiers.

67.4.1 Standard Simplifier

The standard simplifier `RLSIMPL` returns a simplified equivalent of its argument formula. It is much faster though less powerful than the other simplifiers.

As an optional argument there can be a *theory* passed. This is a list of atomic formulas assumed to hold. Simplification is then performed on the basis of these assumptions.

Example:

```
4: rlsimpl g;

(x1 + x2 = 0 or x1 + x3 = 0) and (x1 + x2 = 0 or x2 + x3 = 0)

 and (x1 + x3 = 0 or x2 + x3 = 0)
```

67.4.2 Tableau Simplifier

The standard simplifier preserves the basic Boolean structure of a formula. The tableau methods, in contrast, provide a technique for changing the Boolean structure of a formula by constructing case distinctions.

The function `RLATAB` automatically finds a suitable case distinction. Based on `RLATAB`, the function `RLITAB` iterates this process until no further simplification can be detected. There is a more fundamental entry point `RLTAB` for manually entering case distinctions.

67.4.3 Gröbner Simplifier

The Gröbner simplifier considers algebraic simplification rules between the atomic formulas of the input formula. The usual procedure called for Gröbner simplification is `RLGSN`. Similar to the standard simplifier, there is an optional theory argument.

Example:

```
5: rlgsn(x*y+1<>0 or y*z+1<>0 or x-z=0);

true
```

67.5 Normal Forms

67.5.1 Boolean Normal Forms

`RLCNF` and `RLDNF` compute conjunctive resp. disjunctive normal forms of their formula arguments. Subsumption and cut strategies are applied to decrease the number of clauses.

67.5.2 Miscellaneous Normal Forms

`RLNNF` computes a negation normal form. This is an `and`-`or`-combination of atomic formulas.

`RLPNF` computes a prenex normal form of its argument. That is, all quantifiers are moved outside such that they form a block in front of a quantifier-free matrix formula.

67.6 Quantifier Elimination and Variants

Quantifier elimination computes quantifier-free equivalents for given first-order formulas. For OFSF and DVFSF, REDLOG uses a technique based on elimination set ideas. The OFSF implementation is restricted to at most quadratic occurrences of the quantified variables, but includes numerous heuristic strategies for coping with higher degrees. The DVFSF implementation is restricted to formulas that are linear in the quantified variables. The ACFSF quantifier elimination is based on comprehensive Gröbner basis computation; there are no degree restrictions for this context

67.6.1 Quantifier Elimination

`RLQE` performs quantifier elimination on its argument formula. There is an optional theory argument in the style of `RLSIMPL` supported.

Example:

```
6: rlqe(ex(x,a*x**2+b*x+c>0),{a<0});

         2
4*a*c - b  < 0
```

For OFSF and DVFSF there is a variant `RLQEA` available. It returns instead of a quantifier-free equivalent, a list of condition-solution pairs containing, e.g., satisfying sample points for outermost existential quantifier blocks.

Example:

```
7: rlqea(ex(x,a*x**2+b*x+c>0),{a<0});

           2
{{4*a*c - b  < 0,

                            2
         - sqrt( - 4*a*c + b ) - 2*a*epsilon1 - b
  {x = -------------------------------------------}}}
                           2*a
```

67.6.2 Generic Quantifier Elimination

OFSF allows generic quantifier elimination `RLGQE`, which enlarges the theory by disequations, i.e. `<>`-atomic formulas, wherever this supports the quantifier elimination. There is also generic quantifier elimination with answer available: `RLGQEA`.

Example:

```
8: rlgqe ex(x,a*x**2+b*x+c>0);

{{a <> 0},

          2
 4*a*c - b  < 0 or a >= 0}
```

67.6.3 Linear Optimization

`RLOPT` uses quantifier elimination for linear optimization. It takes as arguments a list of constraints and the target function. The target function is minimized subject to the constraints.

Chapter 68

RESET: Reset REDUCE to its initial state

J. P. Fitch
School of Mathematical Sciences, University of Bath
BATH BA2 7AY, England
e–mail: jpff@maths.bath.ac.uk

This package defines a command `RESETREDUCE` that works through the history of previous commands, and clears any values which have been assigned, plus any rules, arrays and the like. It also sets the various switches to their initial values. It is not complete, but does work for most things that cause a gradual loss of space.

Chapter 69

RESIDUE: A residue package

Wolfram Koepf
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraße 7
D–14195 Berlin–Dahlem, Germany
e–mail: Koepf@zib.de

This package supports the calculation of residues. The residue $\operatorname*{Res}_{z=a} f(z)$ of a function $f(z)$ at the point $a \in \mathbb{C}$ is defined as

$$\operatorname*{Res}_{z=a} f(z) = \frac{1}{2\pi i} \oint f(z)\, dz \ ,$$

with integration along a closed curve around $z = a$ with winding number 1.

It contains two REDUCE operators:

- `residue(f,z,a)` determines the residue of f at the point $z = a$ if f is meromorphic at $z = a$. The calculation of residues at essential singularities of f is not supported.
- `poleorder(f,z,a)` determines the pole order of f at the point $z = a$ if f is meromorphic at $z = a$.

Note that both functions use the `TAYLOR` package (chapter 82).

```
load_package residue;

residue(x/(x^2-2),x,sqrt(2));

 1
---
 2

poleorder(x/(x^2-2),x,sqrt(2));

1

residue(sin(x)/(x^2-2),x,sqrt(2));

 sqrt(2)*sin(sqrt(2))
----------------------
          4

poleorder(sin(x)/(x^2-2),x,sqrt(2));

1

residue((x^n-y^n)/(x-y)^2,x,y);

  n
 y *n
------
  y

poleorder((x^n-y^n)/(x-y)^2,x,y);

1
```

Chapter 70

RLFI: REDUCE LaTeX formula interface

Richard Liska, Ladislav Drska
Computational Physics Group
Faculty of Nuclear Sciences and Physical Engineering
Czech Technical University in Prague, Brehova 7, 115 19 Prague 1
Czech Republic
e–mail: liska@siduri.fjfi.cvut.cz

The RLFI package provides the printing of REDUCE expressions in LaTeX format, so it can be used directly for document production. Various mathematical constructions are supported by the interface including subscripts, superscripts, font changing, Greek letters, divide-bars, integral and sum signs, derivatives etc.

The interface is connected to REDUCE by three new switches and several statements. To activate the LaTeX output mode the switch `latex` must be set `on`. This switch causes all outputs to be written in the LaTeX syntax of formulas. The switch `VERBATIM` is used for input printing control. If it is `on` input to REDUCE system is typeset in LaTeX verbatim environment after the line containing the string `REDUCE Input:`.

The switch `lasimp` controls the algebraic evaluation of input formulas. If it is `on` every formula is evaluated, simplified and written in the form given by ordinary REDUCE statements and switches such as `factor`, `order`, `rat` etc. In the case when the `lasimp` switch is `off` evaluation, simplification or reordering of formulas is not performed and REDUCE acts only as a formula parser and the form of the formula output is exactly the same as that of the input, the only difference remains in the syntax. The mode `off lasimp` is designed especially for typesetting of formulas for which the user needs preservation of their structure. This switch has no meaning if the switch `Latex` is `off` and thus is working only for LaTeX output.

For every identifier used in the typeset REDUCE formula the following properties can be defined by the statement `defid`:

- its printing symbol (Greek letters can be used).
- the font in which the symbol will be typeset.
- accent which will be typeset above the symbol.

Symbols with indexes are treated in REDUCE as operators. Each index corresponds to an argument of the operator. The meaning of operator arguments (where one wants to typeset them) is declared by the statement `defindex`. This statement causes the arguments to be typeset as subscripts or superscripts (on left or right-hand side of the operator) or as arguments of the operator.

The statement `mathstyle` defines the style of formula typesetting. The variable `laline!*` defines the length of output lines.

The fractions with horizontal divide bars are typeset by using the new REDUCE infix operator `\`. This operator is not algebraically simplified. During typesetting of powers the checking on the form of the power base and exponent is performed to determine the form of the typeset expression (*e.g.* sqrt symbol, using parentheses).

Some special forms can be typeset by using REDUCE prefix operators. These are as follows:

- `int` - integral of an expression.
- `dint` - definite integral of an expression.
- `df` - derivative of an expression.
- `pdf` - partial derivative of an expression.
- `sum` - sum of expressions.
- `product` - product of expressions.

- `sqrt` - square root of expression.

There are still some problems unsolved in the present version of the interface as follows:

- breaking the formulas which do not fit on one line.
- automatic decision where to use divide bars in fractions.
- distinction of two- or more-character identifiers from the product of one-character symbols.
- typesetting of matrices.

Chapter 71

ROOTS: A REDUCE root finding package

Stanley L. Kameny
Los Angeles, U.S.A.

The root finding package is designed so that it can be used as an independent package, or it can be integrated with and called by `SOLVE`.

71.1 Top Level Functions

The top level functions can be called either as symbolic operators from algebraic mode, or they can be called directly from symbolic mode with symbolic mode arguments. Outputs are expressed in forms that print out correctly in algebraic mode.

71.1.1 Functions that refer to real roots only

The three functions `REALROOTS`, `ISOLATER` and `RLROOTNO` can receive 1, 2 or 3 arguments.

The first argument is the polynomial p, that can be complex and can have multiple or zero roots. If arg2 and arg3 are not present, all real roots are found. If the additional arguments are present, they restrict the region of consideration.

- If there are two arguments the second is either POSITIVE or NEGATIVE. The function will only find positive or negative roots
- If arguments are (p,arg2,arg3) then Arg2 and Arg3 must be r (a real number) or EXCLUDE r, or a member of the list POSITIVE, NEGATIVE, INFINITY, -INFINITY. EXCLUDE r causes the value r to be excluded from the region. The order of the sequence arg2, arg3 is unimportant. Assuming that arg2 $\leq$ arg3 when both are numeric, then

{-INFINITY,INFINITY}	(or {})	all roots;
{arg2,NEGATIVE}	represents	$-\infty < r < arg2$;
{arg2,POSITIVE}	represents	$arg2 < r < \infty$;

  In each of the following, replacing an *arg* with EXCLUDE *arg* converts the corresponding inclusive $\leq$ to the exclusive $<$

{arg2,-INFINITY}	represents	$-\infty < r \leq arg2$;
{arg2,INFINITY}	represents	$arg2 \leq r < \infty$;
{arg2,arg3}	represents	$arg2 \leq r \leq arg3$;

- If zero is in the interval the zero root is included.

REALROOTS finds the real roots of the polynomial p. Precision of computation is guaranteed to be sufficient to separate all real roots in the specified region. (cf. MULTIROOT for treatment of multiple roots.)

ISOLATER produces a list of rational intervals, each containing a single real root of the polynomial p, within the specified region, but does not find the roots.

RLROOTNO computes the number of real roots of p in the specified region, but does not find the roots.

71.1.2 Functions that return both real and complex roots

ROOTS p; This is the main top level function of the roots package. It will find all roots, real and complex, of the polynomial p to an accuracy that is sufficient to separate them and which is a minimum of 6 decimal places. The value returned by ROOTS is a list of equations for all roots. In addition, ROOTS stores separate lists of real roots and complex roots in the global variables ROOTSREAL and ROOTSCOMPLEX.

The output of ROOTS is normally sorted into a standard order: a root with smaller real part precedes a root with larger real part; roots with identical real parts are sorted so that larger imaginary part precedes smaller imaginary part.

However, when a polynomial has been factored algebraically then the root sorting is applied to each factor separately. This makes the final resulting order less obvious.

ROOTS_AT_PREC p; Same as ROOTS except that roots values are returned to a minimum of the number of decimal places equal to the current system precision.

ROOT_VAL p; Same as ROOTS_AT_PREC, except that instead of returning a list of equations for the roots, a list of the root value is returned. This is the function that SOLVE calls.

NEARESTROOT(p,s); This top level function finds the root to which the method converges given the initial starting origin s, which can be complex. If there are several roots in the vicinity of s and s is not significantly closer to one root than it is to all others, the convergence could arrive at a root that is not truly the nearest root. This function should therefore be used only when the user is certain that there is only one root in the immediate vicinity of the starting point s.

FIRSTROOT p; ROOTS is called, but only a single root is computed.

71.1.3 Other top level functions

GETROOT(n,rr); If rr has the form of the output of ROOTS, REALROOTS, or NEARESTROOTS; GETROOT returns the rational, real, or complex value of the root equation. An error occurs if $n < 1$ or $n >$ the number of roots in rr.

MKPOLY rr; This function can be used to reconstruct a polynomial whose root equation list is rr and whose denominator is 1. Thus one can verify that if $rr := ROOTS\ p$, and $rr1 := ROOTS\ MKPOLY\ rr$, then $rr1 = rr$. (This will be true if `MULTIROOT` and `RATROOT` are ON, and `ROUNDED` is off.) However, $MKPOLY\ rr - NUM\ p = 0$ will be true if and only if all roots of p have been computed exactly.

71.2 Switches Used in Input

The input of polynomials in algebraic mode is sensitive to the switches `COMPLEX`, `ROUNDED`, and `ADJPREC`. The correct choice of input method is important since incorrect choices will result in undesirable truncation or rounding of the input coefficients.

Truncation or rounding may occur if `ROUNDED` is on and one of the following is true:

1. a coefficient is entered in floating point form or rational form.
2. `COMPLEX` is on and a coefficient is imaginary or complex.

Therefore, to avoid undesirable truncation or rounding, then:

1. `ROUNDED` should be off and input should be in integer or rational form; or
2. `ROUNDED` can be on if it is acceptable to truncate or round input to the current value of system precision; or both `ROUNDED` and `ADJPREC` can be on, in which case system precision will be adjusted to accommodate the largest coefficient which is input; or
3. if the input contains complex coefficients with very different magnitude for the real and imaginary parts, then all three switches `ROUNDED`, `ADJPREC` and `COMPLEX` must be on.

integer and complex modes (off `ROUNDED`) any real polynomial can be input using integer coefficients of any size; integer or rational coefficients can be used to input any real or complex polynomial, independent of the setting of the switch `COMPLEX`. These are the most versatile input modes, since any real or complex polynomial can be input exactly.

modes rounded and complex-rounded (on `ROUNDED`) polynomials can be input using integer coefficients of any size. Floating point coefficients will be truncated or rounded, to a size dependent upon the system. If complex is on, real coefficients can be input to any precision using integer form, but coefficients of imaginary parts of complex coefficients will be rounded or truncated.

71.3 Root Package Switches

RATROOT (Default OFF) If `RATROOT` is on all root equations are output in rational form. Assuming that the mode is `COMPLEX` (*i.e.* `ROUNDED` is off,) the root equations are guaranteed to be able to be input into REDUCE without truncation or rounding errors. (Cf. the function MKPOLY described above.)

MULTIROOT (Default ON) Whenever the polynomial has complex coefficients or has real coefficients and has multiple roots, as determined by the Sturm function, the function `SQFRF` is called automatically to factor the polynomial into square-free factors. If `MULTIROOT` is on, the multiplicity of the roots will be indicated in the output of ROOTS or REALROOTS by printing the root output repeatedly, according to its multiplicity. If `MULTIROOT` is off, each root will be printed once, and all roots should be normally be distinct. (Two identical roots should not appear. If the initial precision of the computation or the accuracy of the output was insufficient to separate two closely-spaced roots, the program attempts to increase accuracy and/or precision if it detects equal roots. If, however, the initial accuracy specified was too low, and it was not possible to separate the roots, the program will abort.)

Chapter 72

RSOLVE: Rational/integer polynomial solvers

Francis J. Wright
School of Mathematical Sciences, Queen Mary and Westfield College
University of London
Mile End Road
London E1 4NS, England
e–mail: F.J.Wright@QMW.ac.uk

The exact rational zeros of a single univariate polynomial using fast modular methods can be calculated. The operator `r_solve` computes all rational zeros and the operator `i_solve` computes only integer zeros in a way that is slightly more efficient than extracting them from the rational zeros.

The first argument is either a univariate polynomial expression or equation with integer, rational or rounded coefficients. Symbolic coefficients are not allowed. The argument is simplified to a quotient of integer polynomials and the denominator is silently ignored.

Subsequent arguments are optional. If the polynomial variable is to be specified then it must be the first optional argument. However, since the variable in a non-constant univariate polynomial can be deduced from the polynomial it is unnecessary to specify it separately, except in the degenerate case that the first argument simplifies to either 0 or $0 = 0$. In this case the result is returned by `i_solve` in terms of the operator `arbint` and by r_solve in terms of the (new) analogous operator arbrat. The operator i_solve will generally run slightly faster than r_solve.

The (rational or integer) zeros of the first argument are returned as a list and the default output format is the same as that used by solve. Each distinct zero is returned in the form of an equation with the variable on the left and the multiplicities of the zeros are assigned to the variable root_multiplicities as a list. However, if the switch multiplicities is turned on then each zero is explicitly included in the solution list the appropriate number of times (and root_multiplicities has no value).

Optional keyword arguments acting as local switches allow other output formats. They have the following meanings:

separate: assign the multiplicity list to the global variable root_multiplicities (the default);

expand or multiplicities: expand the solution list to include multiple zeros multiple times (the default if the multiplicities switch is on);

together: return each solution as a list whose second element is the multiplicity;

nomul: do not compute multiplicities (thereby saving some time);

noeqs: do not return univariate zeros as equations but just as values.

72.1 Examples

```
r_solve((9x^2 - 16)*(x^2 - 9), x);
```

$$\left\{x = \frac{-4}{3}, x = 3, x = -3, x = \frac{4}{3}\right\}$$

```
i_solve((9x^2 - 16)*(x^2 - 9), x);
```

$$\{x = 3, x = -3\}$$

Chapter 73

SCOPE: REDUCE source code optimisation package

J.A. van Hulzen
University of Twente, Department of Computer Science
P.O. Box 217, 7500 AE Enschede
The Netherlands
e–mail: infhvh@cs.utwente.nl

SCOPE is a package to produce optimised versions of algebraic expressions. It can be used in two distinct fashions, as an adjunct to numerical code generation (using GENTRAN, described in chapter 42) or as a stand alone way of investigating structure in an expression.

When used with GENTRAN it is sufficient to set the switch `GENTRANOPT` on, and GENTRAN will then use SCOPE internally. This is described in its internal detail in the GENTRAN manual and the SCOPE documentation.

As a stand-alone package SCOPE provides the operator `OPTIMIZE`.

A SCOPE application is easily performed and based on the use of the following syntax:

<SCOPE_application>	⇒	`OPTIMIZE` <object_seq> [`INAME` <cse_prefix>]
<object_seq>	⇒	<object>[,<object_seq>]
<object>	⇒	<stat> | <alglist> | <alglist_production>
<stat>	⇒	<name> <assignment operator> <expression>
<assignment operator>	⇒	:= | ::= | ::=: | :=:
<alglist>	⇒	{<eq_seq>}
<eq_seq>	⇒	<name> = <expression>[,<eq_seq>]
<alglist_production>	⇒	<name> | <function_application>
<name>	⇒	<id> | <id> (<a_subscript_seq>)
<a_subscript_seq>	⇒	<a_subscript>[,<a_subscript_seq>]
<a_subscript>	⇒	<integer> | <integer infix_expression>
<cse_prefix>	⇒	<id>

A SCOPE action can be applied on one assignment statement, or to a sequence of such statements, separated by commas, or a list of expressions.

The optional use of the `INAME` extension in an `OPTIMIZE` command is introduced to allow the user to influence the generation of cse-names. The cse_prefix is an identifier, used to generate cse-names, by extending it with an integer part. If the cse_prefix consists of letters only, the initially selected integer part is 0. If the user-supplied cse_prefix ends with an integer its value functions as initial integer part.

```
z:=a^2*b^2+10*a^2*m^6+a^2*m^2+2*a*b*m^4+2*b^2*m^6+b^2*m^2;

      2  2         2  6     2  2           4       2  6     2  2
z := a *b  + 10*a *m  + a *m  + 2*a*b*m  + 2*b *m  + b *m

OPTIMIZE z:=:z ;


G0 := b*a
G4 := m*m
G1 := G4*b*b
G2 := G4*a*a
G3 := G4*G4
z := G1 + G2 + G0*(2*G3 + G0) + G3*(2*G1 + 10*G2)
```

it can be desirable to rerun an optimisation request with a restriction on the minimal size of the righthandsides. The command

`SETLENGTH` <integer>$

can be used to produce rhs's with a minimal arithmetic complexity, dictated by the value of its integer argument. Statements, used to rename function applications, are not affected by the `SETLENGTH` command. The default setting is restored with the command

```
RESETLENGTH$
```

Example:

```
SETLENGTH 2$

OPTIMIZE z:=:z INAME s$

       2  2
s1 := b *m
       2  2
s2 := a *m
               4                        4
z := (a*b + 2*m )*a*b + 2*(s1 + 5*s2)*m  + s1 + s2
```

Details of the algorithm used is given in the Scope User's Manual.

Chapter 74

SETS: A basic set theory package

Francis J. Wright
School of Mathematical Sciences, Queen Mary and Westfield College
University of London
Mile End Road
London E1 4NS, England
e–mail: F.J.Wright@QMW.ac.uk

The SETS package provides set theoretic operations on lists and represents the results as normal algebraic-mode lists, so that all other REDUCE facilities that apply to lists can still be applied to lists that have been constructed by explicit set operations.

74.1 Infix operator precedence

The set operators are currently inserted into the standard REDUCE precedence list (see section 2.7) as follows:

```
or and not member memq = set_eq neq eq >= > <= < subset_eq
subset freeof + - setdiff union intersection * / ^ .
```

74.2 Explicit set representation and MKSET

Explicit sets are represented by lists, and there is a need to convert standard REDUCE lists into a set by removing duplicates. The package also orders the members of the set so the standard = predicate will provide set equality.

```
mkset {1,2,y,x*y,x+y};

{x + y,x*y,y,1,2}
```

The empty set is represented by the empty list {}.

74.3 Union and intersection

The intersection operator has the name `intersect`, and set union is denotes by`union`. These operators will probably most commonly be used as binary infix operators applied to explicit sets,

```
{1,2,3} union {2,3,4};

{1,2,3,4}

{1,2,3} intersect {2,3,4};

{2,3}
```

74.4 Symbolic set expressions

If one or more of the arguments evaluates to an unbound identifier then it is regarded as representing a symbolic implicit set, and the union or intersection will evaluate to an expression that still contains the union or intersection operator. These two operators are symmetric, and so if they remain symbolic their arguments will be sorted as for any symmetric operator. Such symbolic set expressions are simplified, but the simplification may not be complete in non-trivial cases. For example:

```
a union b union {} union b union {7,3};

{3,7} union a union b
```

```
a intersect {};
```

```
{}
```

Intersection distributes over union, which is not applied by default but is implemented as a rule list assigned to the variable `set_distribution_rule`, *e.g.*

```
a intersect (b union c);
```

```
(b union c) intersection a
```

```
a intersect (b union c) where set_distribution_rule;
```

```
a intersection b union a intersection c
```

74.5 Set difference

The set difference operator is represented by the symbol `\` and is always output using this symbol, although it can also be input using `setdiff`. It is a binary operator.

```
{1,2,3} \ {2,4};
```

```
{1,3}
```

```
a \ {1,2};
```

```
a\{1,2}
```

```
a \ a;
```

```
{}
```

74.6 Predicates on sets

Set membership, inclusion or equality are all binary infix operators. They can only be used within conditional statements or within the argument of the `evalb` operator provided by this package, and they cannot remain symbolic

– a predicate that cannot be evaluated to a Boolean value causes a normal REDUCE error.

The `evalb` operator provides a convenient shorthand for an `if` statement designed purely to display the value of any Boolean expression (not only predicates defined in this package).

```
if a = a then true else false;

true

evalb(a = a);

true

if a = b then true else false;

false
```

74.6.1 Set membership

Set membership is tested by the predicate `member`. Its left operand is regarded as a potential set element and its right operand *must* evaluate to an explicit set. There is currently no sense in which the right operand could be an implicit set.

```
evalb(1 member {1,2,3});

true

evalb(2 member {1,2} intersect {2,3});

true

evalb(a member b);

***** b invalid as list
```

74.6.2 Set inclusion

Set inclusion is tested by the predicate `subset_eq` where `a subset_eq b` is true if the set a is either a subset of or equal to the set b; strict inclusion is tested by the predicate `subset` where `a subset b` is true if the set a is *strictly* a subset of the set b and is false is a is equal to b. These predicates provide some support for symbolic set expressions, but is incomplete.

```
evalb({1,2} subset_eq {1,2,3});
```

```
true
```

```
evalb({1,2} subset_eq {1,2});
```

```
true
```

```
evalb({1,2} subset {1,2});
```

```
false
```

```
evalb(a subset a union b);
```

```
true
```

```
evalb(a\b subset a);

true
```

An undecidable predicate causes a normal REDUCE error, *e.g.*

```
evalb(a subset_eq {b});

***** Cannot evaluate a subset_eq {b} as Boolean-valued set
 expression
```

74.6.3 Set equality

As explained above, equality of two sets in canonical form can be reliably tested by the standard REDUCE equality predicate (=).

Chapter 75

SPARSE: Sparse Matrices

Stephen Scowcroft
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraše 7
D-14195 Berlin-Dahlem, Germany

75.1 Introduction

This package extends the available matrix feature to enable calculations with sparse matrices. It also provides a selection of functions that are useful in the world of linear algebra with respect to sparse matrices.
The package is loaded by: `load_package sparse;`

75.2 Sparse Matrix Calculations

To extend the syntax of this class of calculations an expression type `sparse` is added. An identifier may be declared a sparse variable by the declaration `sparse`. The size of the sparse matrix must be declared explicitly in the matrix declaration. This declaration `SPARSE` is similar to the declaration `MATRIX`. Once a matrix has been declared a sparse matrix all elements of the matrix are treated as if they were initialized to 0. When printing out a sparse matrix only the non-zero elements are printed due to the fact that only the non-zero elements of the matrix are stored. To assign values to the elements of the declared sparse matrix we use the same syntax as for matrices.

```
sparse aa(10,1),bb(200,200);
aa(1,1):=10;
bb(100,150):=a;
```

75.3 Linear Algebra Package for Sparse Matrices

Most of the functions of this package are related to the functions of the linear algebra package LINALG. For further explanation and examples of the various functions please refer to the LINALG package.

75.3.1 Basic matrix handling

spadd_columns	spadd_rows	spadd_to_columns	spadd_to_rows
spaugment_columns	spchar_poly	spcol_dim	spcopy_into
spdiagonal	spextend	spfind_companion	spget_columns
spget_rows	sphermitian_tp	spmatrix_augment	spmatrix_stack
spminor	spmult_columns	spmult_rows	sppivot
spremove_columns	spremove_rows	sprow_dim	sprows_pivot
spstack_rows	spsub_matrix	spswap_columns	spswap_entries
spswap_rows			

75.3.2 Constructors

Functions that create sparse matrices.

spband_matrix	spblock_matrix	spchar_matrix	spcoeff_matrix
spcompanion	sphessian	spjacobian	spjordan_block
spmake_identity			

75.3.3 High level algorithms

spchar_poly	spcholesky	spgram_schmidt	splu_decom
sppseudo_inverse	svd		

75.3.4 Predicates

matrixp sparsematp squarep symmetricp

Chapter 76

SPDE: A package for finding symmetry groups of PDE's

Fritz Schwarz
GMD, Institut F1
Postfach 1240
5205 St. Augustin, Germany
e–mail: fritz.schwarz@gmd.de

The package SPDE provides a set of functions which may be applied to determine the symmetry group of Lie- or point-symmetries of a given system of partial differential equations. Preferably it is used interactively on a computer terminal. In many cases the determining system is solved completely automatically. In some other cases the user has to provide some additional input information for the solution algorithm to terminate.

76.1 System Functions and Variables

The symmetry analysis of partial differential equations logically falls into three parts. Accordingly the most important functions provided by the package are:

Some other useful functions for obtaining various kinds of output are:

SPDE expects a system of differential equations to be defined as the values of the operator `deq` and other operators. A simple example follows.

Function name	Operation
CRESYS(<*arguments*>)	Constructs determining system
SIMPSYS()	Solves determining system
RESULT()	Prints infinitesimal generators and commutator table

Table 76.1: SPDE Functions

Function name	Operation
PRSYS()	Prints determining system
PRGEN()	Prints infinitesimal generators
COMM(U,V)	Prints commutator of generators U and V

Table 76.2: SPDE Useful Output Functions

```
load_package spde;

deq 1:=u(1,1)+u(1,2,2);

deq(1) := u(1,2,2) + u(1,1)

CRESYS deq 1;

PRSYS();


GL(1):=2*df(eta(1),u(1),x(2)) - df(xi(2),x(2),2) - df(xi(2),x(1))

GL(2):=df(eta(1),u(1),2) - 2*df(xi(2),u(1),x(2))

GL(3):=df(eta(1),x(2),2) + df(eta(1),x(1))

GL(4):=df(xi(2),u(1),2)

GL(5):=df(xi(2),u(1)) - df(xi(1),u(1),x(2))

GL(6):=2*df(xi(2),x(2)) - df(xi(1),x(2),2) - df(xi(1),x(1))

GL(7):=df(xi(1),u(1),2)

GL(8):=df(xi(1),u(1))
```

```
GL(9):=df(xi(1),x(2))


The remaining dependencies

xi(2) depends on u(1),x(2),x(1)

xi(1) depends on u(1),x(2),x(1)

eta(1) depends on u(1),x(2),x(1)
```

A detailed description can be found in the SPDE documentation and examples.

Chapter 77

SPECFN: Package for special functions

Chris Cannam & Winfried Neun
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany
e–mail: neun@zib.de

This package is designed to provide algebraic and numeric manipulations of several common special functions, namely:

- Bernoulli Numbers and Polynomials;
- Euler numbers and Polynomials;
- Fibonacci numbers and Polynomials;
- Stirling Numbers;
- Binomial Coefficients;
- Pochhammer notation;
- The Gamma function;
- The Psi function and its derivatives;
- The Riemann Zeta function;

- The Bessel functions J and Y of the first and second kinds;
- The modified Bessel functions I and K;
- The Hankel functions H1 and H2;
- The Kummer hypergeometric functions M and U;
- The Beta function, and Struve, Lommel and Whittaker functions;
- The Airy functions;
- The Exponential Integral, the Sine and Cosine Integrals;
- The Hyperbolic Sine and Cosine Integrals;
- The Fresnel Integrals and the Error function;
- The Dilog function;
- The Polylogarithm and Lerch Phi function;
- Hermite Polynomials;
- Jacobi Polynomials;
- Legendre Polynomials;
- Associated Legendre Functions (Spherical and Solid Harmonics);
- Laguerre Polynomials;
- Chebyshev Polynomials;
- Gegenbauer Polynomials;
- Lambert's ω function;
- Jacobi Elliptic Functions and Integrals;
- 3j symbols, 6j symbols and Clebsch Gordan coefficients;
- and some well-known constants.

77.1 Simplification and Approximation

All of the operators supported by this package have certain algebraic simplification rules to handle special cases, poles, derivatives and so on. Such rules are applied whenever they are appropriate. However, if the `ROUNDED` switch is on, numeric evaluation is also carried out. Unless otherwise stated below, the result of an application of a special function operator to real or complex numeric arguments in rounded mode will be approximated numerically whenever it is possible to do so. All approximations are to the current precision.

77.2 Constants

Some well-known constants are defined in the special function package. Important properties of these constants which can be used to define them are also known. Numerical values are computed at arbitrary precision if the switch ROUNDED is on.

- Euler_Gamma : Euler's constants, also available as -$\psi(1)$;
- Catalan : Catalan's constant;
- Khinchin : Khinchin's constant;
- Golden_Ratio : $\frac{1+\sqrt{5}}{2}$

77.3 Functions

The functions provided by this package are given in the following tables.

Function	Operator
$\binom{n}{m}$	`Binomial(n,m)`
Motzkin(n)	`Motzkin(n)`
Bernoulli(n) or B_n	`Bernoulli(n)`
Euler(n) or E_n	`Euler(n)`
Fibonacci(n) or F_n	`Fibonacci(n)`
$S_n^{(m)}$	`Stirling1(n,m)`
$\mathbf{S}_n^{(m)}$	`Stirling2(n,m)`
$B(z,w)$	`Beta(z,w)`
$\Gamma(z)$	`Gamma(z)`
incomplete Beta $B_x(a,b)$	`iBeta(a,b,x)`
incomplete Gamma $\Gamma(a,z)$	`iGamma(a,z)`
$(a)_k$	`Pochhammer(a,k)`
$\psi(z)$	`Psi(z)`
$\psi^{(n)}(z)$	`Polygamma(n,z)`
Riemann's $\zeta(z)$	`Zeta(z)`
$J_\nu(z)$	`BesselJ(nu,z)`
$Y_\nu(z)$	`BesselY(nu,z)`
$I_\nu(z)$	`BesselI(nu,z)`
$K_\nu(z)$	`BesselK(nu,z)`
$H_\nu^{(1)}(z)$	`Hankel1(nu,z)`
$H_\nu^{(2)}(z)$	`Hankel2(nu,z)`
$B(z,w)$	`Beta(z,w)`

Function	Operator
$\mathbf{H}_\nu(z)$	`StruveH(nu,z)`
$\mathbf{L}_\nu(z)$	`StruveL(nu,z)`
$s_{a,b}(z)$	`Lommel1(a,b,z)`
$S_{a,b}(z)$	`Lommel2(a,b,z)`
$Ai(z)$	`Airy_Ai(z)`
$Bi(z)$	`Airy_Bi(z)`
$Ai'(z)$	`Airy_Aiprime(z)`
$Bi'(z)$	`Airy_Biprime(z)`
$M(a,b,z)$ or ${}_1F_1(a,b;z)$ or $\Phi(a,b;z)$	`KummerM(a,b,z)`
$U(a,b,z)$ or $z^{-a}{}_2F_0(a,b;z)$ or $\Psi(a,b;z)$	`KummerU(a,b,z)`
$M_{\kappa,\mu}(z)$	`WhittakerM(kappa,mu,z)`
$W_{\kappa,\mu}(z)$	`WhittakerW(kappa,mu,z)`
$B_n(x)$	`BernoulliP(n,x)`
$E_n(x)$	`EulerP(n,x)`
Fibonacci Polynomials $F_n(x)$	`FibonacciP(n,x)`
$C_n^{(\alpha)}(x)$	`GegenbauerP(n,alpha,x)`
$H_n(x)$	`HermiteP(n,x)`
$L_n(x)$	`LaguerreP(n,x)`
$L_n^{(m)}(x)$	`LaguerreP(n,m,x)`
$P_n(x)$	`LegendreP(n,x)`
$P_n^{(m)}(x)$	`LegendreP(n,m,x)`
$P_n^{(\alpha,\beta)}(x)$	`JacobiP(n,alpha,beta,x)`
$U_n(x)$	`ChebyshevU(n,x)`
$T_n(x)$	`ChebyshevT(n,x)`

Function	Operator
$Y_n^m(x,y,z,r2)$	`SolidHarmonicY(n,m,x,y,z,r2)`
$Y_n^m(\theta,\phi)$	`SphericalHarmonicY(n,m,theta,phi)`
$\begin{pmatrix} j_1 & j_2 & j_3 \\ m_1 & m_2 & m_3 \end{pmatrix}$	`ThreeJSymbol({j1,m1},{j2,m2},{j3,m3})`
$(j_1 m_1 j_2 m_2 \| j_1 j_2 j_3 - m_3)$	`Clebsch_Gordan({j1,m1},{j2,m2},{j3,m3})`
$\begin{Bmatrix} j_1 & j_2 & j_3 \\ l_1 & l_2 & l_3 \end{Bmatrix}$	`SixJSymbol({j1,j2,j3},{l1,l2,l3})`

Function	Operator
$Si(z)$	`Si(z)`
$si(z)$	`s_i(z)`
$Ci(z)$	`Ci(z)`
$Shi(z)$	`Shi(z)`
$Chi(z)$	`Chi(z)`
$erf(z)$	`erf(z)`
$erfc(z)$	`erfc(z)`
$Ei(z)$	`Ei(z)`
$li(z)$	`li(z)`
$C(x)$	`Fresnel_C(x)`
$S(x)$	`Fresnel_S(x)`
$dilog(z)$	`dilog(z)`
$Li_n(z)$	`Polylog(n,z)`
Lerch $\Phi(z,s,a)$	`Lerch_Phi(z,s,a)`
$sn(u\|m)$	`Jacobisn(u,m)`
$dn(u\|m)$	`Jacobidn(u,m)`
$cn(u\|m)$	`Jacobicn(u,m)`
$cd(u\|m)$	`Jacobicd(u,m)`
$sd(u\|m)$	`Jacobisd(u,m)`
$nd(u\|m)$	`Jacobind(u,m)`
$dc(u\|m)$	`Jacobidc(u,m)`
$nc(u\|m)$	`Jacobinc(u,m)`
$sc(u\|m)$	`Jacobisc(u,m)`
$ns(u\|m)$	`Jacobins(u,m)`
$ds(u\|m)$	`Jacobids(u,m)`
$cs(u\|m)$	`Jacobics(u,m)`
$F(\phi\|m)$	`EllipticF(phi,m)`
$K(m)$	`EllipticK(m)`
$E(\phi\|m) or E(m)$	`EllipticE(phi,m) or EllipticE(m)`
$H(u\|m), H_1(u\|m), \Theta_1(u\|m), \Theta(u\|m)$	`EllipticTheta(a,u,m)`
$\theta_1(u\|m), \theta_2(u\|m), \theta_3(u\|m), \theta_4(u\|m)$	`EllipticTheta(a,u,m)`
$Z(u\|m)$	`Zeta_function(u,m)`
Lambert $\omega(z)$	`Lambert_W(z)`

Chapter 78

SPECFN2: Special special functions

Victor S. Adamchik
Byelorussian University
Minsk, Belorus

and

Winfried Neun
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany
e–mail: neun@zib.de

The (generalised) hypergeometric functions

$$ {}_pF_q\left(\left.\begin{matrix} a_1, \ldots, a_p \\ b_1, \ldots, b_q \end{matrix}\right| z\right) $$

are defined in textbooks on special functions.

78.1 REDUCE operator HYPERGEOMETRIC

The operator `hypergeometric` expects 3 arguments, namely the list of upper parameters (which may be empty), the list of lower parameters (which may be empty too), and the argument, e.g:

```
hypergeometric ({},{},z);

 Z
E

hypergeometric ({1/2,1},{3/2},-x^2);

 ATAN(X)
---------
    X
```

78.2 Enlarging the HYPERGEOMETRIC operator

Since hundreds of particular cases for the generalised hypergeometric functions can be found in the literature, one cannot expect that all cases are known to the `hypergeometric` operator. Nevertheless the set of special cases can be augmented by adding rules to the REDUCE system, *e.g.*

```
let {hypergeometric({1/2,1/2},{3/2},-(~x)^2) => asinh(x)/x};
```

Chapter 79

SUM: A package for series summation

Fujio Kako
Department of Mathematics, Faculty of Science
Hiroshima University
Hiroshima 730, JAPAN
e–mail: kako@ics.nara-wu.ac.jp

This package implements the Gosper algorithm for the summation of series. It defines operators SUM and PROD. The operator SUM returns the indefinite or definite summation of a given expression, and the operator PROD returns the product of the given expression. These are used with the syntax:

`SUM`(EXPR:*expression*, K:*kernel*, [LOLIM:*expression* [, UPLIM:*expression*]])
`PROD`(EXPR:*expression*, K:*kernel*, [LOLIM:*expression* [, UPLIM:*expression*]])

If there is no closed form solution, these operators return the input unchanged. UPLIM and LOLIM are optional parameters specifying the lower limit and upper limit of the summation (or product), respectively. If UPLIM is not supplied, the upper limit is taken as K (the summation variable itself).

For example:

```
sum(n**3,n);

sum(a+k*r,k,0,n-1);
```

```
sum(1/((p+(k-1)*q)*(p+k*q)),k,1,n+1);

prod(k/(k-2),k);
```

Gosper's algorithm succeeds whenever the ratio

$$\frac{\sum_{k=n_0}^{n} f(k)}{\sum_{k=n_0}^{n-1} f(k)}$$

is a rational function of n. The function SUM!-SQ handles basic functions such as polynomials, rational functions and exponentials.

The trigonometric functions sin, cos, *etc.* are converted to exponentials and then Gosper's algorithm is applied. The result is converted back into sin, cos, sinh and cosh.

Summations of logarithms or products of exponentials are treated by the formula:

$$\sum_{k=n_0}^{n} \log f(k) = \log \prod_{k=n_0}^{n} f(k)$$

$$\prod_{k=n_0}^{n} \exp f(k) = \exp \sum_{k=n_0}^{n} f(k)$$

Other functions can be summed by providing LET rules which must relate the functions evaluated at k and $k-1$ (k being the summation variable).

```
operator f,gg;  % gg used to avoid possible conflict with high energy
                % physics operator.

for all n,m such that fixp m let
        f(n+m)=if m > 0 then f(n+m-1)*(b*(n+m)**2+c*(n+m)+d)
                   else f(n+m+1)/(b*(n+m+1)**2+c*(n+m+1)+d);

for all n,m such that fixp m let
        gg(n+m)=if m > 0 then gg(n+m-1)*(b*(n+m)**2+c*(n+m)+e)
                   else gg(n+m+1)/(b*(n+m+1)**2+c*(n+m+1)+e);
```

```
sum(f(n-1)/gg(n),n);

     f(n)
---------------
 gg(n)*(d - e)
```

Chapter 80

SUSY2: Super Symmetry

Ziemowit Popowicz
Institute of Theoretical Physics, University of Wroclaw
pl. M. Borna 9 50-205 Wroclaw, Poland
e-mail: ziemek@ift.uni.wroc.pl

This package deals with supersymmetric functions and with algebra of supersymmetric operators in the extended N=2 as well as in the nonextended N=1 supersymmetry. It allows us to make the realization of SuSy algebra of differential operators, compute the gradients of given SuSy Hamiltonians and to obtain SuSy version of soliton equations using the SuSy Lax approach. There are also many additional procedures encountered in the SuSy soliton approach, as for example: conjugation of a given SuSy operator, computation of general form of SuSy Hamiltonians (up to SuSy-divergence equivalence), checking of the validity of the Jacobi identity for some SuSy Hamiltonian operators.

To load the package, type `load susy2;`

For full explanation and further examples, please refer to the detailed documentation and the susy2.tst which comes with this package.

80.1 Operators

80.1.1 Operators for constructing Objects

The superfunctions are represented in this package by `BOS`(f,n,m) for superbosons and `FER`(f,n,m) for superfermions. The first index denotes the name of the given superobject, the second denotes the value of SuSy derivatives, and the last gives the value of usual derivative.
In addition to the definitions of the superfunctions, also the inverse and the exponential of superbosons can be defined (where the inverse is defined as `BOS`(f,n,m,-1) with the property *bos(f,n,m,-1)*bos(f,n,m,1)=1*). The exponential of the superboson function is `AXP`(`BOS`(f,0,0)).
The operator `FUN` and `GRAS` denote the classical and the Grassmann function.
Three different realizations of supersymmetric derivatives are implemented. To select traditional realization declare `LET TRAD`. In order to select chiral or chiral1 algebra declare `LET CHIRAL` or `LET CHIRAL1`. For usual differentiation the operator `D`(1) stands for right and `D`(2) for left differentiation. SuSy derivatives are denoted as *der* and *del*. `DER` and `DEL` are one component argument operations and represent the left and right operators. The action of these operators on the superfunctions depends on the choice of the supersymmetry algebra.

BOS(f,n,m)	BOS(f,n,m,k)	FER(f,n,m)	AXP(f)	FUN(f,n)	FUN(f,n,m)
GRAS(f,n)	AXX(f)	D(1)	D(2)	D(3)	D(-1)
D(-2)	D(-3)	D(-4)	DR(-n)	DER(1)	DER(2)
DEL(1)	DEL(2)				

Example:

```
1: load susy2;

2: bos(f,0,2,-2)*axp(fer(k,1,2))*del(1);  %first susy derivative

2*fer(f,1,2)*bos(f,0,2,-3)*axp(fer(k,1,2))

 - bos(k,0,3)*bos(f,0,2,-2)*axp(fer(k,1,2))

 + del(1)*bos(f,0,2,-2)*axp(fer(k,1,2))

3: sub(del=der,ws);
```

```
bos(f,0,2,-2)*axp(fer(k,1,2))*der(1)
```

80.1.2 Commands

There are plenty of operators on superfunction objects. Some of them are introduced here briefly.

- By using the operators `FPART`, `BPART`, `BFPART` and `BF_PART` it is possible to compute the coordinates of the arbitrary SuSy expressions.
- With `W_COMB`, `FCOMB` and `PSE_ELE` there are three operators to be able to construct different possible combinations of superfunctions and super-pseudo-differential elements with the given conformal dimensions .
- The three operators `S_PART`, `D_PART` and `SD_PART` are implemented to obtain the components of the (pseudo)-SuSy element.
- `RZUT` is used to obtain the projection onto the invariant subspace (with respect to commutator) of algebra of pseudo-SuSy-differential algebra.
- To obtain the list of the same combinations of some superfunctions and (SuSy) derivatives from some given operator-valued expression, the operators `LYST`, `LYST1` and `LYST2` are constructed.

`FPART`(expression)	`BPART`(expression)
`BF_PART`(expression,n)	`B_PART`(expression,n)
`PR`(n,expression)	`PG`(n,expression)
`W_COMB`({{f,n,x},...},m,z,y)	`FCOMB`({{f,n,x},...},m,z,y)
`PSE_ELE`(n,{{f,n},...},z)	
`S_PART`(expression,n)	`D_PART`(expression,n)
`SD_PART`(expression,n,m)	`CP`(expression)
`RZUT`(expression,n)	`LYST`(expression)
`LYST1`(expression)	`LYST2`(expression)
`CHAN`(expression)	`ODWA`(expression)
`GRA`(expression,f)	`DYW`(expression,f)
`WAR`(expression,f)	`DOT_HAM`(equations,expression)
`N_GAT`(operator,list)	`FJACOB`(operator,list)
`JACOB`(operator,list,{α, β, γ})	`MACIERZ`(expression,x,y)
`S_INT`(number,expression,list)	

Example:

```
4: xxx:=fer(f,2,3);

xxx := fer(f,2,3)

5: fpart(xxx);  % all components

                - fun(f0,4) + 2*fun(f1,3)      gras(ff2,4)
{gras(ff2,3), ----------------------------,0, -------------}
                            2                       2
6: bpart(xxx);  % bosonic sector

     - fun(f0,4) + 2*fun(f1,3)
{0,----------------------------,0,0}
                2

9: b_part(xxx,1); %the given component in the bosonic sector

  - fun(f0,4) + 2*fun(f1,3)
----------------------------
             2
```

80.2 Options

The are several options defined in this package. Please note that they are activated by typing `let <option>`. See also above.
The `TRAD`, `CHIRAL` and `CHIRAL1` select the different realizations of the supersymmetric derivatives. By default traditional algebra is selected.
If the command `LET INVERSE` is used, then three indices *bos* objects are transformed onto four indices objects.

```
TRAD  CHIRAL  CHIRAL1  INVERSE  DRR  NODRR
```

Example:

```
10: let inverse;

11: bos(f,0,3)**3*bos(k,3,1)**40*bos(f,0,3,-2);

bos(k,3,1,40)*bos(f,0,3,1);

12: clearrules inverse;

13: xxx:=fer(f,1,2)*bos(k,0,2,-2);

xxx := fer(f,1,2)*bos(k,0,2,-2)

14: pr(1,xxx); % first susy derivative

- 2*fer(k,1,2)*fer(f,1,2)*bos(k,0,2,-3) + bos(k,0,2,-2)*bos(f,0,3)

15: pr(2,xxx); %second susy derivative

- 2*fer(k,2,2)*fer(f,1,2)*bos(k,0,2,-3) - bos(k,0,2,-2)*bos(f,3,2)

16: clearrules trad;

17: let chiral; % changing to chiral algebra

18: pr(1,xxx);

- 2*fer(k,1,2)*fer(f,1,2)*bos(k,0,2,-3)
```

Chapter 81

SYMMETRY: Symmetric matrices

Karin Gatermann
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany
e–mail: gatermann@zib.de

The SYMMETRY package provides procedures that compute symmetry-adapted bases and block diagonal forms of matrices which have the symmetry of a group.

81.1 Operators for linear representations

The data structure for a linear representation, a *representation*, is a list consisting of the group identifier and equations which assign matrices to the generators of the group.

Example:

```
rr:=mat((0,1,0,0),
        (0,0,1,0),
        (0,0,0,1),
        (1,0,0,0));

sp:=mat((0,1,0,0),
```

```
          (1,0,0,0),
          (0,0,0,1),
          (0,0,1,0));

   representation:={D4,rD4=rr,sD4=sp};
```

For orthogonal (unitarian) representations the following operators are available.

`canonicaldecomposition(representation);`

returns an equation giving the canonical decomposition of the linear representation.

`character(representation);`

computes the character of the linear representation. The result is a list of the group identifier and of lists consisting of a list of group elements in one equivalence class and a real or complex number.

`symmetrybasis(representation,nr);`

computes the basis of the isotypic component corresponding to the irreducible representation of type nr. If the nr-th irreducible representation is multidimensional, the basis is symmetry adapted. The output is a matrix.

`symmetrybasispart(representation,nr);`

is similar as `symmetrybasis`, but for multidimensional irreducible representations only the first part of the symmetry adapted basis is computed.

`allsymmetrybases(representation);`

is similar as `symmetrybasis` and `symmetrybasispart`, but the bases of all isotypic components are computed and thus a complete coordinate transformation is returned.

`diagonalize(matrix,representation);`

returns the block diagonal form of matrix which has the symmetry of the given linear representation. Otherwise an error message occurs.

81.2 Display Operators

Access is provided to the information for a group, and for adding knowledge for other groups. This is explained in detail in the Symmetry on-line documentation.

Chapter 82

TAYLOR: Manipulation of Taylor series

Rainer Schöpf
Zentrum für Datenverarbeitung der Universität Mainz
Anselm-Franz-von-Bentzel-Weg 12
D-55055 Mainz, Germany
e–mail: Schoepf@Uni-Mainz.DE

The TAYLOR package of REDUCE allow Taylor expansion in one or several variables, and efficient manipulation of the resulting Taylor series. Capabilities include basic operations (addition, subtraction, multiplication and division), and also application of certain algebraic and transcendental functions. To a certain extent, Laurent and Puiseux expansions can be performed as well. In many cases, separable singularities are detected and factored out.

`TAYLOR`(EXP:*exprn*[,VAR:*kernel*, VAR_0:*exprn*,ORDER:*integer*]...):*exprn*

where EXP is the expression to be expanded. It can be any REDUCE object, even an expression containing other Taylor kernels. VAR is the kernel with respect to which EXP is to be expanded. VAR_0 denotes the point about which and ORDER the order up to which expansion is to take place. If more than one (VAR, VAR0, ORDER) triple is specified `TAYLOR` will expand its first argument independently with respect to each variable in turn. For example,

```
taylor(e^(x^2+y^2),x,0,2,y,0,2);
```

will calculate the Taylor expansion up to order $X^2 * Y^2$. Note that once the expansion has been done it is not possible to calculate higher orders. Instead of a kernel, VAR may also be a list of kernels. In this case expansion will take place in a way so that the *sum* of the degrees of the kernels does not exceed ORDER. If VAR_0 evaluates to the special identifier `INFINITY` `TAYLOR` tries to expand EXP in a series in 1/VAR.

The expansion is performed variable per variable, *i.e.* in the example above by first expanding $\exp(x^2 + y^2)$ with respect to x and then expanding every coefficient with respect to y.

There are two extra operators to compute the Taylor expansions of implicit and inverse functions:

`IMPLICIT_TAYLOR`(F:*exprn*,VAR1,VAR2:*kernel*,

VAR1_0,VAR2_0:*exprn*, ORDER:*integer*):*exprn*

takes a function F depending on two variables VAR1 and VAR2 and computes the Taylor series of the implicit function VAR2(VAR1) given by the equation F(VAR1,VAR2) = 0. For example,

```
implicit_taylor(x^2 + y^2 - 1,x,y,0,1,5);
```

`INVERSE_TAYLOR`(F:*exprn*,VAR1,VAR2:*kernel*,

VAR1_0:*exprn*, ORDER:*integer*):*exprn*

takes a function F depending on VAR1 and computes the Taylor series of the inverse of F with respect to VAR2. For example,

```
inverse_taylor(exp(x)-1,x,y,0,8);
```

When a Taylor kernel is printed, only a certain number of (non-zero) coefficients are shown. If there are more, an expression of the form `(`n `terms)` is printed to indicate how many non-zero terms have been suppressed. The number of terms printed is given by the value of the shared algebraic variable `TAYLORPRINTTERMS`. Allowed values are integers and the special identifier `ALL`. The latter setting specifies that all terms are to be printed. The default setting is 5.

If the switch `TAYLORKEEPORIGINAL` is set to `ON` the original expression EXP is kept for later reference. It can be recovered by means of the operator

`TAYLORORIGINAL`(EXP:*exprn*):*exprn*

An error is signalled if EXP is not a Taylor kernel or if the original expression was not kept, *i.e.* if `TAYLORKEEPORIGINAL` was `OFF` during expansion. The template of a Taylor kernel, *i.e.* the list of all variables with respect to which expansion took place together with expansion point and order can be extracted using

`TAYLORTEMPLATE`(EXP:*exprn*):*list*

This returns a list of lists with the three elements (VAR,VAR0,ORDER). As with `TAYLORORIGINAL`, an error is signalled if EXP is not a Taylor kernel.

`TAYLORTOSTANDARD`(EXP:*exprn*):*exprn*

converts all Taylor kernels in EXP into standard form and resimplifies the result.

`TAYLORSERIESP`(EXP:*exprn*):*boolean*

may be used to determine if EXP is a Taylor kernel. Note that this operator is subject to the same restrictions as, *e.g.*, ORDP or NUMBERP, *i.e.* it may only be used in boolean expressions in `IF` or `LET` statements. Finally there is

`TAYLORCOMBINE`(EXP:*exprn*):*exprn*

which tries to combine all Taylor kernels found in EXP into one. Operations currently possible are:

- Addition, subtraction, multiplication, and division.
- Roots, exponentials, and logarithms.
- Trigonometric and hyperbolic functions and their inverses.

Application of unary operators like `LOG` and `ATAN` will nearly always succeed. For binary operations their arguments have to be Taylor kernels with the same template. This means that the expansion variable and the expansion point must match. Expansion order is not so important, different order usually means that one of them is truncated before doing the operation.

If `TAYLORKEEPORIGINAL` is set to `ON` and if all Taylor kernels in `exp` have their original expressions kept `TAYLORCOMBINE` will also combine these and store the result as the original expression of the resulting Taylor kernel. There is also the switch `TAYLORAUTOEXPAND` (see below).

There are a few restrictions to avoid mathematically undefined expressions: it is not possible to take the logarithm of a Taylor kernel which has no terms (*i.e.* is zero), or to divide by such a beast. There are some provisions made to detect singularities during expansion: poles that arise because the denominator has zeros at the expansion point are detected and properly treated, *i.e.* the Taylor kernel will start with a negative power. (This is accomplished by expanding numerator and denominator separately and combining the results.) Essential singularities of the known functions (see above) are handled correctly.

Differentiation of a Taylor expression is possible. Differentiating with respect to one of the Taylor variables will decrease the order by one.

Substitution is a bit restricted: Taylor variables can only be replaced by other kernels. There is one exception to this rule: one can always substitute a Taylor variable by an expression that evaluates to a constant. Note that REDUCE will not always be able to determine that an expression is constant.

Only simple Taylor kernels can be integrated. More complicated expressions that contain Taylor kernels as parts of themselves are automatically converted into a standard representation by means of the TAYLORTOSTANDARD operator. In this case a suitable warning is printed.

It is possible to revert a Taylor series of a function f, *i.e.*, to compute the first terms of the expansion of the inverse of f from the expansion of f. This is done by the operator

`TAYLORREVERT`(EXP:*exprn*,OLDVAR:*kernel*, NEWVAR:*kernel*):*exprn*

EXP must evaluate to a Taylor kernel with OLDVAR being one of its expansion variables. Example:

```
taylor (u - u**2, u, 0, 5);
taylorrevert (ws, u, x);
```

This package introduces a number of new switches:

- If `TAYLORAUTOCOMBINE` is set to `ON` REDUCE automatically combines Taylor expressions during the simplification process. This is equivalent to applying `TAYLORCOMBINE` to every expression that contains Taylor kernels. Default is `ON`.

- `TAYLORAUTOEXPAND` makes Taylor expressions "contagious" in the sense that `TAYLORCOMBINE` tries to Taylor expand all non-Taylor subexpressions and to combine the result with the rest. Default is `OFF`.

- `TAYLORKEEPORIGINAL`, if set to `ON`, forces the package to keep the original expression, *i.e.* the expression that was Taylor expanded. All operations performed on the Taylor kernels are also applied to this expression which can be recovered using the operator `TAYLORORIGINAL`. Default is `OFF`.

- `TAYLORPRINTORDER`, if set to `ON`, causes the remainder to be printed in big-O notation. Otherwise, three dots are printed. Default is `ON`.

Chapter 83

TPS: A truncated power series package

Alan Barnes
Dept. of Computer Science and Applied Mathematics
Aston University, Aston Triangle,
Birmingham B4 7ET, England
e–mail: barnesa@aston.ac.uk

and

Julian Padget
School of Mathematics, University of Bath
Bath, BA2 7AY, England
e–mail: jap@maths.bath.ac.uk

This package implements formal Laurent series expansions in one variable using the domain mechanism of REDUCE. This means that power series objects can be added, multiplied, differentiated *etc.* like other first class objects in the system. A lazy evaluation scheme is used in the package and thus terms of the series are not evaluated until they are required for printing or for use in calculating terms in other power series. The series are extendible giving the user the impression that the full infinite series is being manipulated. The errors that can sometimes occur using series that are truncated at some fixed depth (for example when a term in the required series depends on terms of an intermediate series beyond the truncation depth) are thus avoided.

83.1 Basic Truncated Power Series

83.1.1 PS Operator

Syntax:

`PS`(EXPRN:*algebraic*,DEPVAR:*kernel*,ABOUT:*algebraic*):*ps object*

The `PS` operator returns a power series object representing the univariate formal power series expansion of EXPRN with respect to the dependent variable DEPVAR about the expansion point ABOUT. EXPRN may itself contain power series objects.

The algebraic expression ABOUT should simplify to an expression which is independent of the dependent variable DEPVAR, otherwise an error will result. If ABOUT is the identifier `INFINITY` then the power series expansion about DEPVAR $= \infty$ is obtained in ascending powers of 1/DEPVAR.

The power series object representing EXPRN is compiled and then a number of terms of the power series expansion are evaluated. The expansion is carried out as far as the value specified by `PSEXPLIM`. If, subsequently, the value of `PSEXPLIM` is increased, sufficient information is stored in the power series object to enable the additional terms to be calculated without recalculating the terms already obtained.

If the function has a pole at the expansion point then the correct Laurent series expansion will be produced.

The following examples are valid uses of `PS`:

```
psexplim 6;
ps(log x,x,1);
ps(e**(sin x),x,0);
ps(x/(1+x),x,infinity);
ps(sin x/(1-cos x),x,0);
```

New user-defined functions may be expanded provided the user provides LET rules giving

1. the value of the function at the expansion point
2. a differentiation rule for the new function.

For example

```
operator sech;
forall x let df(sech x,x)= - sech x * tanh x;
let sech 0 = 1;
ps(sech(x**2),x,0);
```

The power series expansion of an integral may also be obtained (even if REDUCE cannot evaluate the integral in closed form). An example of this is

```
ps(int(e**x/x,x),x,1);
```

Note that if the integration variable is the same as the expansion variable then REDUCE's integration package is not called; if on the other hand the two variables are different then the integrator is called to integrate each of the coefficients in the power series expansion of the integrand. The constant of integration is zero by default. If another value is desired, then the shared variable `PSINTCONST` should be set to required value.

83.1.2 PSORDLIM Operator

Syntax:

`PSORDLIM`(UPTO:*integer*):*integer*

or

`PSORDLIM`():*integer*

An internal variable is set to the value of `UPTO` (which should evaluate to an integer). The value returned is the previous value of the variable. The default value is 15.

If `PSORDLIM` is called with no argument, the current value is returned.

The significance of this control is that the system attempts to find the order of the power series required, that is the order is the degree of the first non-zero term in the power series. If the order is greater than the value of this variable an error message is given and the computation aborts. This prevents infinite loops in examples such as

```
ps(1 - (sin x)**2 - (cos x)**2,x,0);
```

where the expression being expanded is identically zero, but is not recognised as such by REDUCE.

83.2 Controlling Power Series

83.2.1 PSTERM Operator

Syntax:

`PSTERM`(TPS:*power series object*,NTH:*integer*):*algebraic*

The operator `PSTERM` returns the NTH term of the existing power series object TPS. If NTH does not evaluate to an integer or TPS to a power series object an error results. It should be noted that an integer is treated as a power series.

83.2.2 PSORDER Operator

Syntax:

`PSORDER`(TPS:*power series object*):*integer*

The operator `PSORDER` returns the order, that is the degree of the first non-zero term, of the power series object TPS. TPS should evaluate to a power series object or an error results. If TPS is zero, the identifier `UNDEFINED` is returned.

83.2.3 PSSETORDER Operator

Syntax:

`PSSETORDER`(TPS:*power series object*, ORD:*integer*):*integer*

The operator `PSSETORDER` sets the order of the power series TPS to the value ORD, which should evaluate to an integer. If TPS does not evaluate to a power series object, then an error occurs. The value returned by this operator is the previous order of TPS, or 0 if the order of TPS was undefined. This operator is useful for setting the order of the power series of a function defined by a differential equation in cases where the power series package is inadequate to determine the order automatically.

83.2.4 PSDEPVAR Operator

Syntax:

`PSDEPVAR`(TPS:*power series object*):*identifier*

The operator `PSDEPVAR` returns the expansion variable of the power series object TPS. TPS should evaluate to a power series object or an integer, otherwise an error results. If TPS is an integer, the identifier `UNDEFINED` is returned.

83.2.5 PSEXPANSIONPT operator

Syntax:

`PSEXPANSIONPT`(TPS:*power series object*):*algebraic*

The operator `PSEXPANSIONPT` returns the expansion point of the power series object TPS. TPS should evaluate to a power series object or an integer, otherwise an error results. If TPS is integer, the identifier `UNDEFINED` is returned. If the expansion is about infinity, the identifier `INFINITY` is returned.

83.2.6 PSFUNCTION Operator

Syntax:

`PSFUNCTION`(TPS:*power series object*):*algebraic*

The operator `PSFUNCTION` returns the function whose expansion gave rise to the power series object TPS. TPS should evaluate to a power series object or an integer, otherwise an error results.

83.2.7 PSCHANGEVAR Operator

Syntax:

`PSCHANGEVAR`(TPS:*power series object*, X:*kernel*):*power series object*

The operator `PSCHANGEVAR` changes the dependent variable of the power series object TPS to the variable X. TPS should evaluate to a power series object and X to a kernel, otherwise an error results. Also X should not appear as a parameter in TPS. The power series with the new dependent variable is returned.

83.2.8 PSREVERSE Operator

Syntax:

`PSREVERSE`(TPS:*power series object*):*power series*

Power series reversion. The power series TPS is functionally inverted. Four cases arise:

1. If the order of the series is 1, then the expansion point of the inverted series is 0.
2. If the order is 0 *and* if the first order term in TPS is non-zero, then the expansion point of the inverted series is taken to be the coefficient of the zeroth order term in TPS.
3. If the order is -1 the expansion point of the inverted series is the point at infinity. In all other cases a REDUCE error is reported because the series cannot be inverted as a power series. Puiseux expansion would be required to handle these cases.
4. If the expansion point of TPS is finite it becomes the zeroth order term in the inverted series. For expansion about 0 or the point at infinity the order of the inverted series is one.

If TPS is not a power series object after evaluation an error results.

Here are some examples:

```
ps(sin x,x,0);
psreverse(ws); % produces series for asin x about x=0.
ps(exp x,x,0);
psreverse ws; % produces series for log x about x=1.
ps(sin(1/x),x,infinity);
psreverse(ws); % produces series for 1/asin(x) about x=0.
```

83.2.9 PSCOMPOSE Operator

Syntax:

`PSCOMPOSE`(TPS1:*power series*, TPS2:*power series*):*power series*

`PSCOMPOSE` performs power series composition. The power series TPS1 and TPS2 are functionally composed. That is to say that TPS2 is substituted for the expansion variable in TPS1 and the result expressed as a power series. The dependent variable and expansion point of the result coincide with those of TPS2. The following conditions apply to power series composition:

1. If the expansion point of TPS1 is 0 then the order of the TPS2 must be at least 1.
2. If the expansion point of TPS1 is finite, it should coincide with the coefficient of the zeroth order term in TPS2. The order of TPS2 should also be non-negative in this case.
3. If the expansion point of TPS1 is the point at infinity then the order of TPS2 must be less than or equal to -1.

If these conditions do not hold the series cannot be composed (with the current algorithm terms of the inverted series would involve infinite sums) and a REDUCE error occurs.

Examples of power series composition include the following.

```
a:=ps(exp y,y,0);  b:=ps(sin x,x,0);
pscompose(a,b);
% Produces the power series expansion of exp(sin x)
% about x=0.

a:=ps(exp z,z,1); b:=ps(cos x,x,0);
pscompose(a,b);
% Produces the power series expansion of exp(cos x)
% about x=0.

a:=ps(cos(1/x),x,infinity);  b:=ps(1/sin x,x,0);
pscompose(a,b);
% Produces the power series expansion of cos(sin x)
% about x=0.
```

83.2.10 PSSUM Operator

Syntax:

PSSUM(J:*kernel* = LOWLIM:*integer*, COEFF:*algebraic*, X:*kernel*, ABOUT:*algebraic*, POWER:*algebraic*):*power series*

The formal power series sum for J from LOWLIM to INFINITY of

```
COEFF*(X-ABOUT)**POWER
```

or if ABOUT is given as INFINITY

```
COEFF*(1/X)**POWER
```

is constructed and returned. This enables power series whose general term is known to be constructed and manipulated using the other procedures of the power series package.

J and X should be distinct simple kernels. The algebraics ABOUT, COEFF and POWER should not depend on the expansion variable X, similarly the algebraic ABOUT should not depend on the summation variable J. The algebraic POWER should be a strictly increasing integer valued function of J for J in the range LOWLIM to INFINITY.

```
pssum(n=0,1,x,0,n*n);
% Produces the power series summation for n=0 to
% infinity of x**(n*n).

pssum(m=1,(-1)**(m-1)/(2m-1),y,1,2m-1);
% Produces the power series expansion of atan(y-1)
% about y=1.

pssum(j=1,-1/j,x,infinity,j);
% Produces the power series expansion of log(1-1/x)
% about the point at infinity.

pssum(n=0,1,x,0,2n**2+3n) + pssum(n=1,1,x,0,2n**2-3n);
% Produces the power series summation for n=-infinity
% to +infinity of x**(2n**2+3n).
```

83.2.11 Arithmetic Operations

As power series objects are domain elements they may be combined together in algebraic expressions in algebraic mode of REDUCE in the normal way.

For example if A and B are power series objects then the commands such as:

```
a*b;
a**2+b**2;
```

will produce power series objects representing the product and the sum of the squares of the power series objects A and B respectively.

83.2.12 Differentiation

If A is a power series object depending on X then the input `df(a,x);` will produce the power series expansion of the derivative of A with respect to X.

83.3 Restrictions and Known Bugs

If A and B are power series objects and X is a variable which evaluates to itself then currently expressions such as `a/b` and `a*x` do not evaluate to a single power series object (although the results are in each case formally valid). Instead use `ps(a/b,x,0)` and `ps(a*x,x,0)` *etc.*.

Chapter 84

TRI: TeX REDUCE interface

Werner Antweiler, Andreas Strotmann and Volker Winkelmann
University of Cologne Computer Center, Abt. Anwendungssoftware, Robert-Koch-Straße 10
5000 K"oln 41, Germany
e–mail: antweil@epas.utoronto.ca strotmann@rrz.uni-koeln.de
winkelmann@rrz.uni-koeln.de

The REDUCE-TEX-Interface incorporates three levels of TEX output: without line breaking, with line breaking, and with line breaking plus indentation.

During loading the package some default initialisations are performed. The default page width is set to 15 centimetres, the tolerance for page breaking is set to 20 by default. Moreover, TRI is enabled to translate Greek names, *e.g.* TAU or PSI, into equivalent TEX symbols, *e.g.* τ or ψ, respectively. Letters are printed lowercase as defined through assertion of the set LOWERCASE.

84.1 Switches for TRI

The three TRI modes can be selected by switches, which can be used alternatively and incrementally. Switching `TEX` on gives standard TEX-output; switching `TEXBREAK` gives broken TEX-output, and `TEXINDENT` to give broken TEX-output plus indentation. Thus the three levels of TRI are enabled or disabled according to:

```
On TeX;          % switch TeX is on
On TeXBreak;     % switches TeX and TeXBreak are on
On TeXIndent;    % switches TeX, TeXBreak and TeXIndent are on
Off TeXIndent;   % switch TeXIndent is off
Off TeXBreak;    % switches TeXBreak and TeXIndent are off
Off TeX;         % all three switches are off
```

How TRI breaks multiple lines of TeX-code may be controlled by setting values for page width and tolerance

```
TeXsetbreak(page_width, tolerance);
```

Page width is measured in millimetres, and tolerance is a positive integer in the closed interval $[0 \ldots 10000]$. The higher the tolerance, the more breakpoints become feasible. A tolerance of 0 means that actually no breakpoint will be considered feasible, while a value of 10000 allows any breakpoint to be considered feasible. For line-breaking without indentation, suitable values for the tolerance lie between 10 and 100. As a rule of thumb, use higher values the deeper the term is nested. If using indentation, use much higher tolerance values; reasonable values for tolerance here lie between 700 and 1500.

84.1.1 Adding Translations

Sometimes it is desirable to add special REDUCE-symbol-to-TeX-item translations. For such a task TRI provides a function `TeXlet` which binds any REDUCE-symbol to one of the predefined TeX-items. A call to this function has the following syntax:

`TeXlet`(*REDUCE-symbol*, *TeX-item*);

For example

```
TeXlet('velocity,'!v);
TeXlet('gamma,\verb|'!\!G!a!m!m!a! |);
TeXlet('acceleration,\verb|'!\!v!a!r!t!h!e!t!a! |);
```

Besides this method of single assertions one can assert one of (currently) two standard sets providing substitutions for lowercase and Greek letters. These sets are loaded by default. These sets can be switched on or off using the functions

`TeXassertset` *setname*;
`TeXretractset` *setname*;

where the setnames currently defined are `'GREEK` and `'LOWERCASE`.

There are facilities for creating other sets of substitutions, using the function `TeXitem`.

84.2 Examples of Use

Some representative examples demonstrate the capabilities of TRI.

```
load_package tri;
    % TeX-REDUCE-Interface 0.50
    % set greek asserted
    % set lowercase asserted
    % \tolerance 10
    % \hsize=150mm

TeXsetbreak(150,250);
    % \tolerance 250
    % \hsize=150mm

on TeXindent;

(x+y)^16/(v-w)^16;
    $$\displaylines{\qdd
    \(x^{16}
      +16\cdot x^{15}\cdot y
      +120\cdot x^{14}\cdot y^{2}
      +560\cdot x^{13}\cdot y^{3}
      +1820\cdot x^{12}\cdot y^{4}
      +4368\cdot x^{11}\cdot y^{5}\nl
      \off{327680}
      +8008\cdot x^{10}\cdot y^{6}
      +11440\cdot x^{9}\cdot y^{7}
      +12870\cdot x^{8}\cdot y^{8}
      +11440\cdot x^{7}\cdot y^{9}
      +8008\cdot x^{6}\cdot y^{10}\nl
      \off{327680}
      +4368\cdot x^{5}\cdot y^{11}
```

```
      +1820\cdot x^{4}\cdot y^{12}
      +560\cdot x^{3}\cdot y^{13}
      +120\cdot x^{2}\cdot y^{14}
      +16\cdot x\cdot y^{15}
      +y^{16}
    \)
    /\nl
    \(v^{16}
      -16\cdot v^{15}\cdot w
      +120\cdot v^{14}\cdot w^{2}
      -560\cdot v^{13}\cdot w^{3}
      +1820\cdot v^{12}\cdot w^{4}
      -4368\cdot v^{11}\cdot w^{5}\nl
      \off{327680}
      +8008\cdot v^{10}\cdot w^{6}
      -11440\cdot v^{9}\cdot w^{7}
      +12870\cdot v^{8}\cdot w^{8}
      -11440\cdot v^{7}\cdot w^{9}
      +8008\cdot v^{6}\cdot w^{10}
      -4368\cdot v^{5}\cdot w^{11}\nl
      \off{327680}
      +1820\cdot v^{4}\cdot w^{12}
      -560\cdot v^{3}\cdot w^{13}
      +120\cdot v^{2}\cdot w^{14}
      -16\cdot v\cdot w^{15}
      +w^{16}
    \)
    \Nl}$$
```

A simple example using matrices:

```
load_package ri;
    % TeX-REDUCE-Interface 0.50
    % set greek asserted
    % set lowercase asserted
    % \tolerance 10
    % \hsize=150mm

on Tex;

mat((1,a-b,1/(c-d)),(a^2-b^2,1,sqrt(c)),((a+b)/(c-d),sqrt(d),1));
    $$
    \pmatrix{1&a
            -b&
```

```
            \frac{1}{
                  c
                  -d}\cr
            a^{2}
            -b^{2}&1&
            \sqrt{c}\cr
            \frac{a
                  +b}{
                  c
                  -d}&
            \sqrt{d}&1\cr
            }
    $$
```

Note that the resulting output uses a number of TeX macros which are defined in the file `tridefs.tex` which is distributed with the example file.

Chapter 85

TRIGSIMP: Simplification and factorisation of trigonometric and hyperbolic functions

Wolfram Koepf, Andreas Bernig and Herbert Melenk
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraße 7
D–14195 Berlin–Dahlem, Germany
e–mail: Koepf@zib.de

There are three procedures included in TRIGSIMP: trigsimp, trigfactorize and triggcd. The first is for finding simplifications of trigonometric or hyperbolic expressions with many options, the second for factorising them and the third for finding the greatest common divisor of two trigonometric or hyperbolic polynomials.

85.1 Simplifiying trigonometric expressions

As there is no normal form for trigonometric and hyperbolic functions, the same function can convert in many different directions, *e.g.* $\sin(2x) \leftrightarrow 2\sin(x)\cos(x)$. The user has the possibility to give several parameters to the procedure `trigsimp` in order to influence the direction of transformations. The decision whether a rational expression in trigonometric and hyperbolic functions vanishes or not is possible.

To simplify a function `f`, one uses `trigsimp(f[,options])`. Example:

```
2: trigsimp(sin(x)^2+cos(x)^2);

1
```

Possible options are (* denotes the default):

1. `sin` (*) or `cos`
2. `sinh` (*) or `cosh`
3. `expand` (*) or `combine` or `compact`
4. `hyp` or `trig` or `expon`
5. `keepalltrig`

From each group one can use at most one option, otherwise an error message will occur. The first group fixes the preference used while transforming a trigonometric expression. The second group is the equivalent for the hyperbolic functions. The third group determines the type of transformations. With the default `expand`, an expression is written in a form only using single arguments and no sums of arguments. With `combine`, products of trigonometric functions are transformed to trigonometric functions involving sums of arguments.

```
trigsimp(sin(x)^2,cos);

         2
 - cos(x)  + 1

trigsimp(sin(x)*cos(y),combine);


 sin(x - y) + sin(x + y)
-------------------------
            2
```

With `compact`, the REDUCE operator `compact` (see chapter 31) is applied to `f`. This leads often to a simple form, but in contrast to `expand` one doesn't get a normal form.

```
trigsimp((1-sin(x)**2)**20*(1-cos(x)**2)**20,compact);

      40       40
cos(x)  *sin(x)
```

With the fourth group each expression is transformed to a trigonometric, hyperbolic or exponential form:

```
trigsimp(sin(x),hyp);

 - sinh(i*x)*i

trigsimp(e^x,trig);

     x          x
cos(---) + sin(---)*i
     i          i
```

Usually, `tan`, `cot`, `sec`, `csc` are expressed in terms of `sin` and `cos`. It can be sometimes useful to avoid this, which is handled by the option `keepalltrig`:

```
trigsimp(tan(x+y),keepalltrig);

  - (tan(x) + tan(y))
----------------------
  tan(x)*tan(y) - 1
```

It is possible to use the options of different groups simultaneously.

85.2 Factorising trigonometric expressions

With `trigfactorize(p,x)` one can factorise the trigonometric or hyperbolic polynomial `p` with respect to the argument x. Example:

```
trigfactorize(sin(x),x/2);
```

```
        x        x
{2,cos(---),sin(---)}
        2        2
```

If the polynomial is not coordinated or balanced the output will equal the input. In this case, changing the value for x can help to find a factorisation:

```
trigfactorize(1+cos(x),x);

{cos(x) + 1}

trigfactorize(1+cos(x),x/2);

        x        x
{2,cos(---),cos(---)}
        2        2
```

85.3 GCDs of trigonometric expressions

The operator `triggcd` is an application of `trigfactorize`. With its help the user can find the greatest common divisor of two trigonometric or hyperbolic polynomials. The syntax is: `triggcd(p,q,x)`, where p and q are the polynomials and x is the smallest unit to use. Example:

```
triggcd(sin(x),1+cos(x),x/2);

     x
cos(---)
     2

triggcd(sin(x),1+cos(x),x);

1
```

See also the ASSIST package (chapter 23).

Chapter 86

WU: Wu algorithm for poly systems

Russell Bradford
School of Mathematical Sciences, University of Bath,
Bath, BA2 7AY, England
e–mail: rjb@maths.bath.ac.uk

The interface:

```
wu( {x^2+y^2+z^2-r^2, x*y+z^2-1, x*y*z-x^2-y^2-z+1}, {x,y,z});
```

calls `wu` with the named polynomials, and with the variable ordering `x` > `y` > `z`. In this example, `r` is a parameter.

The result is

```
    2    3    2
{{{r  + z  - z  - 1,

    2  2    2      2    4    2  2    2
   r *y  + r *z + r  - y  - y *z  + z  - z - 2,

          2
   x*y + z  - 1},

  y},

    6  4      6  2    6      4  7      4  6      4  5      4  4
```

```
{{r *z  - 2*r *z  + r  + 3*r *z  - 3*r *z  - 6*r *z  + 3*r *z  + 3*

   4  3      4  2      4      2  10      2  9      2  8      2  7
  r *z  + 3*r *z  - 3*r  + 3*r *z   - 6*r *z  - 3*r *z  + 6*r *z  +

      2  6      2  5      2  4      2  3      2    13      12    11
   3*r *z  + 6*r *z  - 6*r *z  - 6*r *z  + 3*r  + z   - 3*z   + z

        10    9      8      7    6      4      3    2
   + 2*z   + z  + 2*z  - 6*z  - z  + 2*z  + 3*z  - z  - 1,

   2   2    3    2
  y *(r  + z  - z  - 1),

         2
  x*y + z  - 1},

     2    3    2
 y*(r  + z  - z  - 1)}}
```

namely, a list of pairs of characteristic sets and initials for the characteristic sets.

Thus, the first pair above has the characteristic set

$$r^2 + z^3 - z^2 - 1, r^2y^2 + r^2z + r^2 - y^4 - y^2z^2 + z^2 - z - 2, xy + z^2 - 1$$

and initial y.

According to Wu's theorem, the set of roots of the original polynomials is the union of the sets of roots of the characteristic sets, with the additional constraints that the corresponding initial is non-zero. Thus, for the first pair above, we find the roots of $\{r^2 + z^3 - z^2 - 1, \dots\}$ under the constraint that $y \neq 0$. These roots, together with the roots of the other characteristic set (under the constraint of $y(r^2 + z^3 - z^2 - 1) \neq 0$), comprise all the roots of the original set.

Chapter 87

XCOLOR: Calculation of the color factor in non-abelian gauge field theories

A. Kryukov
Institute for Nuclear Physics, Moscow State University
119899, Moscow, Russia
e–mail: kryukov@npi.msu.su

XCOLOR calculates the colour factor in non-abelian gauge field theories. It provides two commands and two operators.

`SUdim` integer

Sets the order of the SU group. The default value is 3.

`SpTT` expression

Sets the normalisation coefficient A in the equation $Sp(T_i T_j) = A\Delta(i,j)$. The default value is 1/2.

`QG`(inQuark, outQuark, Gluon)

Describes the quark-gluon vertex. The parameters may be any identifiers. The first and second of then must be in- and out- quarks correspondingly. Third one is a gluon.

`G3`(Gluon1, Gluon2, Gluon3)

Describes the three-gluon vertex. The parameters may be any identifiers. The order of gluons must be clockwise.

In terms of QG and G3 operators one can input a diagram in "color" space as a product of these operators. For example

```
          e1
       ----->---
      /         \
     /           \
    |      e2     |
 v1*.............*v2
    |             |
     \           /
      \    e3   /
       -----<---
```

where `--->---` is a quark and .... is a gluon.

The related REDUCE expression is `QG(e3,e1,e2)*QG(e1,e3,e2)`.

Chapter 88

XIDEAL: Gröbner for exterior algebra

David Hartley
GMD, Institute I1, Schloss Birlinghoven
D–53757 St. Augustin, Germany
e–mail: David.Hartley@gmd.de

and
Philip A. Tuckey
Max Planck Institute for Physics
Foehringer Ring 6
D–80805 Munich, Germany
e–mail: pht@iws170.mppmu.mpg.de

XIDEAL extends the Gröbner base method to exterior algebras.

XIDEAL constructs Gröbner bases for solving the left ideal membership problem: Gröbner left ideal bases or GLIBs. For graded ideals, where each form is homogeneous in degree, the distinction between left and right ideals vanishes. Furthermore, if the generating forms are all homogeneous, then the Gröbner bases for the non-graded and graded ideals are identical. In this case, XIDEAL is able to save time by truncating the Gröbner basis at some maximum degree if desired. XIDEAL uses the EXCALC package (chapter 39).

88.1 Operators

XIDEAL

XIDEAL calculates a Gröbner left ideal basis in an exterior algebra. The syntax is

```
XIDEAL(S:list of forms[,R:integer]):list of forms.
```

XIDEAL calculates the Gröbner left ideal basis for the left ideal generated by S using graded lexicographical ordering based on the current kernel ordering. The resulting list can be used for subsequent reductions with XMODULOP as long as the kernel ordering is not changed. If the set of generators S is graded, an optional parameter R can be given, and XIDEAL produces a truncated basis suitable for reducing exterior forms of degree less than or equal to R in the left ideal. This can save time and space with large expressions, but the result cannot be used for exterior forms of degree greater than R. See also the switches XSTATS and XFULLREDUCTION.

XMODULO

XMODULO reduces exterior forms to their (unique) normal forms modulo a left ideal. The syntax is

```
XMODULO(F:form, S:list of forms):form
```

or

```
XMODULO(F:list of forms, S:list of forms):list of forms.
```

An alternative infix syntax is also available:

```
F XMODULO S.
```

XMODULO(F,S) first calculates a Gröbner basis for the left ideal generated by S, and then reduces F. F may be either a single exterior form, or a list of forms, and S is a list of forms. If F is a list of forms, each element is reduced, and any which vanish are deleted from the result. If this operator is used more than once, and S does not change between calls, then the Gröbner basis is not recalculated. If the set of generators S is graded, then a truncated Gröbner basis is calculated using the degree of `F` (or the maximal degree in `F`).

XMODULOP

`XMODULOP` reduces exterior forms to their (not necessarily unique) normal forms modulo a set of exterior polynomials. The syntax is

```
XMODULOP(F:form, S:list of forms):form
```

or

```
XMODULOP(F:list of forms, S:list of forms):list of forms.
```

An alternative infix syntax is also available:

```
F XMODULOP S.
```

`XMODULOP(F,S)` reduces `F` with respect to the set of exterior polynomials `S`, which is not necessarily a Gröbner basis. `F` may be either a single exterior form, or a list of forms, and `S` is a list of forms. This operator can be used in conjunction with `XIDEAL` to produce the same effect as `XMODULO`: for a single form `F` in an ideal generated by the graded set `S`, `F XMODULO S` is equivalent to `F XMODULOP XIDEAL(S,EXDEGREE F)`.

88.2 Switches

XFULLREDUCE

`ON XFULLREDUCE` allows `XIDEAL` and `XMODULO` to calculate reduced (but not necessarily normed) Gröbner bases, which speeds up subsequent reductions, and guarantees a unique form (up to scaling) for the Gröbner basis. `OFF XFULLREDUCE` turns of this feature, which may speed up calculation of the Gröbner basis. `XFULLREDUCE` is `ON` by default.

XSTATS

`ON XSTATS` produces counting and timing information. As `XIDEAL` is running, a hash mark (`#`) is printed for each form taken from the input list, followed by a sequences of carets (^) and dollar signs ($). Each caret represents a new basis element obtained by a simple wedge product, and each dollar sign represents a new basis element obtained from an S-polynomial. At the end, a table is printed summarising the calculation. `XSTATS` is `OFF` by default.

88.3 Examples

Suppose EXCALC and XIDEAL have been loaded, the switches are at their default settings, and the following exterior variables have been declared:

```
pform x=0,y=0,z=0,t=0,f(i)=1,h=0,hx=0,ht=0;
```

In a commutative polynomial ring, a single polynomial is its own Gröbner basis. This is no longer true for exterior algebras because of the presence of zero divisors, and can lead to some surprising reductions:

```
xideal {d x^d y - d z^d t};

     {d T^d Z + d X^d Y,

      d X^d Y^d Z,

      d T^d X^d Y}

f(3)^f(4)^f(5)^f(6)
  xmodulo {f(1)^f(2) + f(3)^f(4) + f(5)^f(6)};

     0
```

The heat equation, $h_{xx} = h_t$ can be represented by the following exterior differential system.

```
S := {d h - ht*d t - hx*d x,
      d ht^d t + d hx^d x,
      d hx^d t - ht*d x^d t};
```

`XMODULO` can be used to check that the exterior differential system is closed under exterior differentiation.

```
d S xmodulo S;

     {}
```

Non-graded left and right ideals are no longer the same:

```
d t^(d z+d x^d y) xmodulo {d z+d x^d y};

     0

(d z+d x^d y)^d t xmodulo {d z+d x^d y};

      - 2*d t^d z
```

Higher order forms can now reduce lower order ones:

```
d x xmodulo {d y^d z + d x,d x^d y + d z};

     0
```

Any form containing a 0-form term generates the whole ideal:

```
xideal {1 + f(1) + f(1)^f(2) + f(2)^f(3)^f(4)};

     {1}
```

Chapter 89

ZEILBERG: A package for indefinite and definite summation

Wolfram Koepf and Gregor Stölting
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany
e–mail: Koepf@zib.de

The ZEILBERG package provides an implementation of the Gosper and Zeilberger algorithms for indefinite, and definite summation of hypergeometric terms, respectively, with extensions for ratios of products of powers, factorials, Γ function terms, binomial coefficients, and shifted factorials that are rational-linear in their arguments.

89.1 The GOSPER summation operator

The `gosper` operator is an implementation of the Gosper algorithm.

- `gosper(a,k)` determines a closed form antidifference. If it does not return a closed form solution, then a closed form solution does not exist.

- `gosper(a,k,m,n)` determines

$$\sum_{k=m}^{n} a_k$$

  using Gosper's algorithm. This is only successful if Gosper's algorithm applies.

Example:

```
gosper((-1)^(k+1)*(4*k+1)*factorial(2*k)/
   (factorial(k)*4^k*(2*k-1)*factorial(k+1)),k);

              k
      - ( - 1) *factorial(2*k)
------------------------------------
  2*k
 2   *factorial(k + 1)*factorial(k)

gosper(binomial(k,n),k);

 (k + 1)*binomial(k,n)
-----------------------
         n + 1
```

89.2 EXTENDED_GOSPER operator

The `extended_gosper` operator is an implementation of an extended version of Gosper's algorithm.

- `extended_gosper(a,k)` determines an antidifference g_k of a_k whenever there is a number m such that $h_k - h_{k-m} = a_k$, and h_k is an *m-fold hypergeometric term*, i. e.

$$h_k/h_{k-m} \quad \text{is a rational function with respect to } k.$$

  If it does not return a solution, then such a solution does not exist.

- `extended_gosper(a,k,m)` determines an *m-fold antidifference* h_k of a_k, i. e. $h_k - h_{k-m} = a_k$, if it is an m-fold hypergeometric term.

Examples:

```
extended_gosper(binomial(k/2,n),k);

                   k                          k - 1
 (k + 2)*binomial(---,n) + (k + 1)*binomial(-------,n)
                   2                            2
-------------------------------------------------------
                       2*(n + 1)

extended_gosper(k*factorial(k/7),k,7);

                   k
(k + 7)*factorial(---)
                   7
```

89.3 SUMRECURSION operator

The `sumrecursion` operator is an implementation of the (fast) Zeilberger algorithm.

- `sumrecursion(f,k,n)` determines a holonomic recurrence equation for
  $$\texttt{sum(n)} = \sum_{k=-\infty}^{\infty} f(n,k)$$
  with respect to n. The resulting expression equals zero.

- `sumrecursion(f,k,n,j)` searches for a holonomic recurrence equation of order j.Note that if j is too large, the recurrence equation may not be unique, and only one particular solution is returned.

```
sumrecursion(binomial(n,k),k,n);

2*sum(n - 1) - sum(n)
```

89.4 HYPERRECURSION operator

If a recursion for a generalised hypergeometric function is to be established, one can use

- `hyperrecursion(upper,lower,x,n)` determines a holonomic recurrence equation with respect to n for

$$ {}_pF_q \left(\begin{array}{cccc|} a_1, & a_2, & \cdots, & a_p \\ b_1, & b_2, & \cdots, & b_q \end{array} \, x \right), $$

  where `upper`$= \{a_1, a_2, \ldots, a_p\}$ is the list of upper parameters, and `lower`$= \{b_1, b_2, \ldots, b_q\}$ is the list of lower parameters depending on n.

- `hyperrecursion(upper,lower,x,n,j)` ($j \in \mathbb{N}$) searches only for a holonomic recurrence equation of order j. This operator does not automatically use `extended_sumrecursion`.

```
hyperrecursion({-n,b},{c},1,n);

(b - c - n + 1)*sum(n - 1) + (c + n - 1)*sum(n)
```

If a hypergeometric expression is given in hypergeometric notation, then the use of `hyperrecursion` is more natural than the use of `sumrecursion`.

Moreover the REDUCE operator

- `hyperterm(upper,lower,x,k)` yields the hypergeometric term

$$ \frac{(a_1)_k \cdot (a_2)_k \cdots (a_p)_k}{(b_1)_k \cdot (b_2)_k \cdots (b_q)_k \, k!} x^k $$

  with upper parameters `upper`$= \{a_1, a_2, \ldots, a_p\}$, and lower parameters `lower`$= \{b_1, b_2, \ldots, b_q\}$

in connection with hypergeometric terms.

89.5 HYPERSUM operator

With the operator `hypersum`, hypergeometric sums are directly evaluated in closed form whenever the extended Zeilberger algorithm leads to a recurrence equation containing only two terms:

- `hypersum(upper,lower,x,n)` determines a closed form representation for

${}_pF_q\left(\begin{array}{cccc} a_1, & a_2, & \cdots, & a_p \\ b_1, & b_2, & \cdots, & b_q \end{array}\middle|\, x\right)$, where `upper`$= \{a_1, a_2, \ldots, a_p\}$ is the list of upper parameters, and `lower`$= \{b_1, b_2, \ldots, b_q\}$ is the list of lower parameters depending on n. The result is given as a hypergeometric term with respect to n.

If the result is a list of length m, we call it *m-fold symmetric*, which is to be interpreted as follows: Its j^{th} part is the solution valid for all n of the form $n = mk + j - 1$ ($k \in \mathbb{N}_0$). In particular, if the resulting list contains two terms, then the first part is the solution for even n, and the second part is the solution for odd n.

```
hypersum({a,1+a/2,c,d,-n},{a/2,1+a-c,1+a-d,1+a+n},1,n);

 pochhammer(a - c - d + 1,n)*pochhammer(a + 1,n)
-------------------------------------------------
 pochhammer(a - c + 1,n)*pochhammer(a - d + 1,n)

hypersum({a,1+a/2,d,-n},{a/2,1+a-d,1+a+n},-1,n);

  pochhammer(a + 1,n)
-------------------------
 pochhammer(a - d + 1,n)
```

Note that the operator `togamma` converts expressions given in factorial-Γ-binomial-Pochhammer notation into a pure Γ function representation:

```
togamma(hypersum({a,1+a/2,d,-n},{a/2,1+a-d,1+a+n},-1,n));

 gamma(a - d + 1)*gamma(a + n + 1)
-----------------------------------
 gamma(a - d + n + 1)*gamma(a + 1)
```

89.6 SUMTOHYPER operator

With the operator `sumtohyper`, sums given in factorial-Γ-binomial-Pochhammer notation are converted into hypergeometric notation.

- `sumtohyper(f,k)` determines the hypergeometric representation of $\sum\limits_{k=-\infty}^{\infty} f_k$, *i.e.* its output is `c*hypergeometric(upper,lower,x)`, corresponding to the representation

$$\sum_{k=-\infty}^{\infty} f_k = c \cdot {}_pF_q \left(\begin{array}{cccc} a_1, & a_2, & \cdots, & a_p \\ b_1, & b_2, & \cdots, & b_q \end{array} \middle| x \right) ,$$

  where `upper`$= \{a_1, a_2, \ldots, a_p\}$ and `lower`$= \{b_1, b_2, \ldots, b_q\}$ are the lists of upper and lower parameters.

Examples:

```
sumtohyper(binomial(n,k)^3,k);

hypergeometric({ - n, - n, - n},{1,1},-1)
```

89.7 Simplification Operators

For the decision that an expression a_k is a hypergeometric term, it is necessary to find out whether or not a_k/a_{k-1} is a rational function with respect to k. For the purpose to decide whether or not an expression involving powers, factorials, Γ function terms, binomial coefficients, and Pochhammer symbols is a hypergeometric term, the following simplification operators can be used:

- `simplify_gamma(f)` simplifies an expression `f` involving only rational, powers and Γ function terms.
- `simplify_combinatorial(f)` simplifies an expression `f` involving powers, factorials, Γ function terms, binomial coefficients, and Pochhammer symbols by converting factorials, binomial coefficients, and Pochhammer symbols into Γ function terms, and applying `simplify_gamma` to its result. If the output is not rational, it is given in terms of Γ functions. If factorials are preferred use
- `gammatofactorial` (rule) converting Γ function terms into factorials using $\Gamma(x) \rightarrow (x-1)!$.
- `simplify_gamma2(f)` uses the duplication formula of the Γ function to simplify f.

- `simplify_gamman(f,n)` uses the multiplication formula of the Γ function to simplify f.

The use of `simplify_combinatorial(f)` is a safe way to decide the rationality for any ratio of products of powers, factorials, Γ function terms, binomial coefficients, and Pochhammer symbols.

Example:

```
simplify_gamma2(gamma(2*n)/gamma(n));

  2*n         2*n + 1
 2   *gamma(---------)
                2
-----------------------
      2*sqrt(pi)
```

Chapter 90

ZTRANS: Z-transform package

Wolfram Koepf and Lisa Temme
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustraše 7
D–14195 Berlin–Dahlem, Germany
e–mail: Koepf@zib.de

The Z-Transform of a sequence $\{f_n\}$ is the discrete analogue of the Laplace Transform, and

$$\mathcal{Z}\{f_n\} = F(z) = \sum_{n=0}^{\infty} f_n z^{-n} \; .$$

This series converges in the region outside the circle $|z| = |z_0| = \limsup_{n\to\infty} \sqrt[n]{|f_n|}$. In the same way that a Laplace Transform can be used to solve differential equations, so Z-Transforms can be used to solve difference equations.

SYNTAX: `ztrans(`f_n`, n, z)` where f_n is an expression, and n,z are identifiers.

This package can compute the Z-Transforms of the following list of f_n, and certain combinations thereof.

1	$e^{\alpha n}$	$\frac{1}{(n+k)}$
$\frac{1}{n!}$	$\frac{1}{(2n)!}$	$\frac{1}{(2n+1)!}$
$\frac{\sin(\beta n)}{n!}$	$\sin(\alpha n + \phi)$	$e^{\alpha n}\sin(\beta n)$
$\frac{\cos(\beta n)}{n!}$	$\cos(\alpha n + \phi)$	$e^{\alpha n}\cos(\beta n)$
$\frac{\sin(\beta(n+1))}{n+1}$	$\sinh(\alpha n + \phi)$	$\frac{\cos(\beta(n+1))}{n+1}$
$\cosh(\alpha n + \phi)$	$\binom{n+k}{m}$	

Other Combinations

Linearity	$\mathcal{Z}\{af_n + bg_n\} = a\mathcal{Z}\{f_n\} + b\mathcal{Z}\{g_n\}$
Multiplication by n	$\mathcal{Z}\{n^k \cdot f_n\} = -z\frac{d}{dz}\left(\mathcal{Z}\{n^{k-1} \cdot f_n, n, z\}\right)$
Multiplication by λ^n	$\mathcal{Z}\{\lambda^n \cdot f_n\} = F\left(\frac{z}{\lambda}\right)$
Shift Equation	$\mathcal{Z}\{f_{n+k}\} = z^k\left(F(z) - \sum_{j=0}^{k-1} f_j z^{-j}\right)$
Symbolic Sums	$\mathcal{Z}\left\{\sum_{k=0}^{n} f_k\right\} = \frac{z}{z-1} \cdot \mathcal{Z}\{f_n\}$
	$\mathcal{Z}\left\{\sum_{k=p}^{n+q} f_k\right\}$ combination of the above

where $k,\lambda \in \mathbf{N}-\{0\}$; and a,b are variables or fractions; and $p,q \in \mathbf{Z}$ or are functions of n; and α, β and ϕ are angles in radians.

The calculation of the Laurent coefficients of a regular function results in the following inverse formula for the Z-Transform:

If $F(z)$ is a regular function in the region $|z| > \rho$ then $\exists$ a sequence $\{f_n\}$

with $\mathcal{Z}\{f_n\} = F(z)$ given by

$$f_n = \frac{1}{2\pi i} \oint F(z) z^{n-1} dz$$

SYNTAX: `invztrans(`$F(z)$`, z, n)` where $F(z)$ is an expression, and z,n are identifiers.

This package can compute the Inverse Z-Transforms of any rational function, whose denominator can be factored over **Q**, in addition to the following list of $F(z)$.

$\sin\left(\frac{\sin(\beta)}{z}\right) e^{\left(\frac{\cos(\beta)}{z}\right)}$	$\cos\left(\frac{\sin(\beta)}{z}\right) e^{\left(\frac{\cos(\beta)}{z}\right)}$
$\sqrt{\frac{z}{A}} \sin\left(\sqrt{\frac{z}{A}}\right)$	$\cos\left(\sqrt{\frac{z}{A}}\right)$
$\sqrt{\frac{z}{A}} \sinh\left(\sqrt{\frac{z}{A}}\right)$	$\cosh\left(\sqrt{\frac{z}{A}}\right)$
$z \ \log\left(\frac{z}{\sqrt{z^2-Az+B}}\right)$	$z \ \log\left(\frac{\sqrt{z^2+Az+B}}{z}\right)$
$\arctan\left(\frac{\sin(\beta)}{z+\cos(\beta)}\right)$	

here k,$\lambda \in \mathbf{N}-\{0\}$ and A,B are fractions or variables ($B > 0$) and α,β, & ϕ are angles in radians.

Examples:

```
ztrans(sum(1/factorial(k),k,0,n),n,z);

  1/z
 e   *z
--------
 z - 1

invztrans(z/((z-a)*(z-b)),z,n);

  n    n
 a  - b
---------
```

```
a - b
```

Part III

Standard Lisp Report

Chapter 91

The Standard Lisp Report

Jed Marti
A. C. Hearn
M. L. Griss
C. Griss

91.1 Introduction

Although the programming language LISP was first formulated in 1960 [16], a widely accepted standard has never appeared. As a result, various dialects of LISP were produced [2, 9, 15, 19, 17, 18] in some cases several on the same machine! Consequently, a user often faces considerable difficulty in moving programs from one system to another. In addition, it is difficult to write and use programs which depend on the structure of the source code such as translators, editors and cross-reference programs.

In 1969, a model for such a standard was produced [12] as part of a general effort to make a large LISP based algebraic manipulation program, REDUCE [11], as portable as possible. The goal of this work was to define a uniform subset of LISP 1.5 and its variants so that programs written in this subset could run on any reasonable LISP system.

In the intervening years, two deficiencies in the approach taken in Ref. [12] have emerged. First in order to be as general as possible, the specific semantics and values of several key functions were left undefined. Consequently, programs built on this subset could not make any assumptions about the form of the values of such functions. The second deficiency related to the proposed method of implementation of this language. The model considered in effect two versions of LISP on any given machine, namely Standard LISP and the LISP of the host machine (which we shall refer to as Target LISP). This meant that if any definition was stored in interpretive form, it would vary from implementation to implementation, and consequently one could not write programs in Standard LISP which needed to assume any knowledge about the structure of such forms. This deficiency became apparent during recent work on the development of a portable compiler for LISP [10]. Clearly a compiler has to know precisely the structure of its source code; we concluded that the appropriate source was Standard LISP and not Target LISP.

With these thoughts in mind we decided to attempt again a definition of Standard LISP. However, our approach this time is more aggressive. In this document we define a standard for a reasonably large subset of LISP with as precise as possible a statement about the semantics of each function. Secondly, we now require that the target machine interpreter be modified or written to support this standard, rather than mapping Standard LISP onto Target LISP as previously.

We have spent countless hours in discussion over many of the definitions given in this report. We have also drawn on the help and advice of a lot of friends whose names are given in the Acknowledgements. Wherever possible, we have used the definition of a function as given in the LISP 1.5 Programmer's Manual [16] and have only deviated where we felt it desirable in the light of LISP programming experience since that time. In particular, we have given considerable thought to the question of variable bindings and the definition of the evaluator functions EVAL and APPLY. We have also abandoned the previous definition of LISP arrays in favor of the more accepted idea of a vector which most modern LISP systems support. These are the places where we have strayed furthest from the conventional definitions, but we feel that the consistency which results from our approach is worth the redefinition.

We have avoided entirely in this report problems which arise from environment passing, such as those represented by the FUNARG problem. We do not necessarily exclude these considerations from our standard, but in this report have decided to avoid the controversy which they create. The semantic differences between compiled and interpreted functions is the topic of another paper [10]. Only functions which affect the compiler in a general way make reference to it.

This document is not intended as an introduction to LISP rather it is assumed that the reader is already familiar with some version. The document is thus intended as an arbiter of the syntax and semantics of Standard LISP. However, since it is not intended as an implementation description, we deliberately leave unspecified many of the details on which an actual implementation depends. For example, while we assume the existence of a symbol table for atoms (the "object list" in LISP terminology), we do not specify its structure, since conventional LISP programming does not require this information. Our ultimate goal, however, is to remedy this by defining an interpreter for Standard LISP which is sufficiently complete that its implementation on any given computer will be straightforward and precise. At that time, we shall produce an implementation level specification for Standard LISP which will extend the description of the primitive functions defined herein by introducing a new set of lower level primitive functions in which the structure of the symbol table, heap and so on may be defined.

The plan of this chapter is as follows. In Section 91.2 we describe the various data types used in Standard LISP. In Section 91.3, a description of all Standard LISP functions is presented, organized by type. These functions are defined in an RLISP syntax which is easier to read than LISP S-expressions. Section 91.4 describes global variables which control the operation of Standard LISP.

91.2 Preliminaries

91.2.1 Primitive Data Types

integer Integers are also called "fixed" numbers. The magnitude of an integer is unrestricted. Integers in the LISP input stream are recognized by the grammar:

<digit> ::= 0|1|2|3|4|5|6|7|8|9
<unsigned-integer> ::= *<digit>*|*<unsigned-integer><digit>*
<integer> ::= *<unsigned-integer>* |
        +*<unsigned-integer>* |
        —*<unsigned-integer>*

floating - Any floating point number. The precision of floating point numbers is determined solely by the implementation. In BNF floating point numbers are recognized by the grammar:

<base> ::= *<unsigned-integer>*.|.*<unsigned-integer>*|
<unsigned-integer>.*<unsigned-integer>*
<unsigned-floating> ::= *<base>*|
*<base>*E*<unsigned-integer>*|
*<base>*E-*<unsigned-integer>*|
*<base>*E+*<unsigned-integer>*
<floating> ::= *<unsigned-floating>*|
+*<unsigned-floating>*|-*<unsigned-floating>*

id An identifier is a string of characters which may have the following items associated with it.

print name The characters of the identifier.

flags An identifier may be tagged with a flag. Access is by the FLAG, REMFLAG, and FLAGP functions defined in section 91.3.4 on page 629.

properties An identifier may have an indicator-value pair associated with it. Access is by the PUT, GET, and REMPROP functions defined in section 91.3.4 on page 629.

values/functions An identifier may have a value associated with it. Access to values is by SET and SETQ defined in section 91.3.6 on page 633. The method by which the value is attached to the identifier is known as the binding type, being one of LOCAL, GLOBAL, or FLUID. Access to the binding type is by the GLOBAL, GLOBALP, FLUID, FLUIDP, and UNFLUID functions.

An identifier may have a function or macro associated with it. Access is by the PUTD, GETD, and REMD functions (see "Function Definition", section 91.3.5, on page 631). An identifier may not have both a function and a value associated with it.

OBLIST entry An identifier may be entered and removed from a structure called the OBLIST. Its presence on the OBLIST does not directly affect the other properties. Access to the OBLIST is by the INTERN, REMOB, and READ functions.

The maximum length of a Standard LISP identifier is 24 characters (excluding occurrences of the escape character !) but an implementation may allow more. Special characters (digits in the first position and punctuation) must be prefixed with an escape character, an ! in Standard LISP. In BNF identifiers are recognized by the grammar:

<*special-character*> ::= !<*any-character*>
<*alphabetic*> ::=
A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<*lead-character*> ::= <*special-character*>|<*alphabetic*>
<*regular-character*> ::= <*lead-character*>|<*digit*>
<*last-part*> ::= <*regular-character*> |
<*last-part*><*regular-character*>
<*id*> ::= <*lead-character*>|<*lead-character*><*last-part*>

Note: Using lower case letters in identifiers may cause portability problems. Lower case letters are automatically converted to upper case when the !*RAISE flag is T.

string A set of characters enclosed in double quotes as in "THIS IS A STRING". A quote is included by doubling it as in "HE SAID, ""LISP""". The maximum size of strings is 80 characters but an implementation may allow more. Strings are not part of the OBLIST and are considered constants like numbers, vectors, and function-pointers.

dotted-pair A primitive structure which has a left and right part. A notation called *dot-notation* is used for dotted pairs and takes the form:

(<*left-part*> . <*right-part*>)

The <*left-part*> is known as the CAR portion and the <*right-part*> as the CDR portion. The left and right parts may be of any type. Spaces are used to resolve ambiguity with floating point numbers.

vector A primitive uniform structure in which an integer index is used to access random values in the structure. The individual elements of a vector may be of any type. Access to vectors is restricted to functions defined in “Vectors” section 91.3.9 on page 639. A notation for vectors, *vector-notation*, has the elements of a vector surrounded by square brackets[1]

<*elements*> ::= <*any*>|<*any*> <*elements*>
<*vector*> ::= [<*elements*>]

function-pointer An implementation may have functions which deal with specific data types other than those listed. The use of these entities is to be avoided with the exception of a restricted use of the function-pointer, an access method to compiled EXPRs and FEXPRs. A particular function-pointer must remain valid throughout execution. Systems which change the location of a function must use either an indirect reference or change all occurrences of the associated value. There are two classes of use of function-pointers, those which are supported by Standard LISP but are not well defined, and those which are well defined.

Not well defined Function pointers may be displayed by the print functions or expanded by EXPLODE. The value appears in the convention of the implementation site. The value is not defined in Standard LISP. Function pointers may be created by COMPRESS in the format used for printing but the value used is not defined in Standard LISP. Function pointers may be created by functions which deal with compiled function loading. Again, the values created are not well defined in Standard LISP.

Well defined The function pointer associated with an EXPR or FEXPR may be retrieved by GETD and is valid as long as Standard LISP is in execution. Function pointers may be stored using PUTD, PUT, SETQ and the like or by being bound to variables. Function pointers may be checked for equivalence by EQ. The value may be checked for being a function pointer by the CODEP function.

[1] Vector elements are not separated by commas as in the published version of this document.

91.2.2 Classes of Primitive Data Types

The classes of primitive types are a notational convenience for describing the properties of functions.

boolean The set of global variables {T,NIL}, or their respective values, {T, NIL}.

extra-boolean Any value in the system. Anything that is not NIL has the boolean interpretation T.

ftype The class of definable function types. The set of ids {EXPR, FEXPR, MACRO}.

number The set of {integer, floating}.

constant The set of {integer, floating, string, vector, function-pointer}. Constants evaluate to themselves (see the definition of EVAL in "The Interpreter", section 91.3.14 on page 653).

any The set of {integer, floating, string, id, dotted-pair, vector, function-pointer}. An S-expression is another term for any. All Standard LISP entities have some value unless an ERROR occurs during evaluation or the function causes transfer of control (such as GO and RETURN).

atom The set {any}-{dotted-pair}.

91.2.3 Structures

Structures are entities created out of the primitive types by the use of dotted-pairs. Lists are structures very commonly required as actual parameters to functions. Where a list of homogeneous entities is required by a function this class will be denoted by <**xxx**-*list*> where ***xxx*** is the name of a class of primitives or structures. Thus a list of ids is an *id-list*, a list of integers an *integer-list* and so on.

list A list is recursively defined as NIL or the dotted-pair (any . list). A special notation called *list-notation* is used to represent lists. List-notation eliminates extra parentheses and dots. The list (a . (b . (c . NIL))) in list notation is (a b c). List-notation and dot-notation may be mixed as in (a b . c) or (a (b . c) d) which are (a . (b . c)) and (a . ((b . c) . (d . NIL))). In BNF lists are recognized by the grammar:

<left-part> ::= (| *<left-part>* *<any>*
<list> ::= *<left-part>*) | *<left-part>* . *<any>*)

Note: () is an alternate input representation of NIL.

alist An association list; each element of the list is a dotted-pair, the CAR part being a key associated with the value in the CDR part.

cond-form A cond-form is a list of 2 element lists of the form:

(**ANTECEDENT**:*any* **CONSEQUENT**:*any*)

The first element will henceforth be known as the antecedent and the second as the consequent. The antecedent must have a value. The consequent may have a value or an occurrence of GO or RETURN as described in the "Program Feature Functions", section 91.3.7 on page 635.

lambda A LAMBDA expression which must have the form (in list notation): (LAMBDA parameters body). "parameters" is a list of formal parameters for "body" an S-expression to be evaluated. The semantics of the evaluation are defined with the EVAL function (see "The Interpreter", section 91.3.14 on page 653).

function A LAMBDA expression or a function-pointer to a function. A function is always evaluated as an EVAL, SPREAD form.

91.2.4 Function Descriptions

Each function is provided with a prototypical header line. Each formal parameter is given a name and suffixed with its allowed type. Lower case, italic tokens are names of classes and upper case, bold face, tokens are parameter names referred to in the definition. The type of the value returned by the function (if any) is suffixed to the parameter list. If it is not commonly used the parameter type may be a specific set enclosed in brackets {...}. For example:

`PUTD`(**FNAME**:*id*, **TYPE**:*ftype*, **BODY**:{*lambda, function-pointer*}):*id*

PUTD is a function with three parameters. The parameter FNAME is an id to be the name of the function being defined. TYPE is the type of the function being defined and BODY is a lambda expression or a function-pointer. PUTD returns the name of the function being defined.

Functions which accept formal parameter lists of arbitrary length have the type class and parameter enclosed in square brackets indicating that zero or more occurrences of that argument are permitted. For example:

`AND`([**U**:*any*]):*extra-boolean*

AND is a function which accepts zero or more arguments which may be of any type.

91.2.5 Function Types

EVAL type functions are those which are invoked with evaluated arguments. NOEVAL functions are invoked with unevaluated arguments. SPREAD type functions have their arguments passed in one-to-one correspondence with their formal parameters. NOSPREAD functions receive their arguments as a single list. EVAL, SPREAD functions are associated with EXPRs and NOEVAL, NOSPREAD functions with FEXPRs. EVAL, NOSPREAD and NOEVAL, SPREAD functions can be simulated using NOEVAL, NOSPREAD functions or MACROs.

EVAL, SPREAD type functions may have a maximum of 15 parameters. There is no limit on the number of parameters a NOEVAL, NOSPREAD function or MACRO may have.

In the context of the description of an EVAL, SPREAD function, then we speak of the formal parameters we mean their actual values. However, in a NOEVAL, NOSPREAD function it is the unevaluated actual parameters.

A third function type, the MACRO, implements functions which create S-expressions based on actual parameters. When a macro invocation is encountered, the body of the macro, a lambda expression, is invoked as a NOEVAL, NOSPREAD function with the macro's invocation bound as a list to the macros single formal parameter. When the macro has been evaluated the resulting S-expression is reevaluated. The description of the EVAL and EXPAND functions provide precise details.

91.2.6 Error and Warning Messages

Many functions detect errors. The description of such functions will include these error conditions and suggested formats for display of the generated error messages. A call on the ERROR function is implied but the error number is not specified by Standard LISP. In some cases a warning message is sufficient. To distinguish between errors and warnings, errors are prefixed with five asterisks and warnings with only three.

Primitive functions check arguments that must be of a certain primitive type for being of that type and display an error message if the argument is not correct. The type mismatch error always takes the form:

```
***** PARAMETER not TYPE for FN
```

Here PARAMETER is the unacceptable actual parameter, TYPE is the type that PARAMETER was supposed to be. FN is the name of the function that detected the error.

91.2.7 Comments

The character % signals the start of a comment, text to be ignored during parsing. A comment is terminated by the end of the line it is on. The function READCH must be able to read a comment one character at a time. Comments are transparent to the function READ. % may occur as a character in identifiers by preceding it with the escape character !.

91.3 Functions

91.3.1 Elementary Predicates

Functions in this section return T when the condition defined is met and NIL when it is not. Defined are type checking functions and elementary comparisons.

`ATOM`(**U**:*any*):*boolean* *eval, spread*

Returns T if U is not a pair.

```
EXPR PROCEDURE ATOM(U);
   NULL PAIRP U;
```

`CODEP`(**U**:`any`):*boolean* *eval, spread*

Returns T if U is a function-pointer.

`CONSTANTP`(**U**:*any*):*boolean* *eval, spread*

Returns T if U is a constant (a number, string, function-pointer, or vector).

```
EXPR PROCEDURE CONSTANTP(U);
   NULL OR(PAIRP U, IDP U);
```

`EQ`(**U**:*any*, **V**:*any*):*boolean* *eval, spread*

Returns T if U points to the same object as V. EQ is <u>not</u> a reliable comparison between numeric arguments.

`EQN`(**U**:*any*, **V**:*any*):*boolean* *eval, spread*

Returns T if U and V are EQ or if U and V are numbers and have the same value and type.

`EQUAL`(**U**:*any*, **V**:*any*):*boolean* *eval, spread*

Returns T if U and V are the same. Dotted-pairs are compared recursively to the bottom levels of their trees. Vectors must have identical dimensions and EQUAL values in all positions. Strings must have identical characters. Function pointers must have EQ values. Other atoms must be EQN equal.

`FIXP`(**U**:*any*):*boolean* *eval, spread*

Returns T if U is an integer (a fixed number).

`FLOATP`(**U**:*any*):*boolean* *eval, spread*

Returns T if U is a floating point number.

`IDP`(**U**:*any*):*boolean* *eval, spread*

Returns T if U is an id.

`MINUSP`(**U**:*any*):*boolean* *eval, spread*

Returns T if U is a number and less than 0. If U is not a number or is a positive number, NIL is returned.

```
EXPR PROCEDURE MINUSP(U);
   IF NUMBERP U THEN LESSP(U, 0) ELSE NIL;
```

`NULL`(**U**:*any*):*boolean* *eval, spread*

Returns T if U is NIL.

```
EXPR PROCEDURE NULL(U);
   U EQ NIL;
```

`NUMBERP`(**U**:*any*):*boolean* *eval, spread*

Returns T if U is a number (integer or floating).

```
EXPR PROCEDURE NUMBERP(U);
   IF OR(FIXP U, FLOATP U) THEN T ELSE NIL;
```

`ONEP`(**U**:*any*):*boolean* *eval, spread.*

Returns T if U is a number and has the value 1 or 1.0. Returns NIL otherwise. [a]

```
EXPR PROCEDURE ONEP(U);
   OR(EQN(U, 1), EQN(U, 1.0));
```

[a]The definition in the published report is incorrect as it does not return T for **U** of 1.0.

`PAIRP`(**U**:*any*):*boolean* *eval, spread*

Returns T if U is a dotted-pair.

`STRINGP`(**U**:*any*):*boolean* *eval, spread*

Returns T if U is a string.

`VECTORP`(**U**:*any*):*boolean* *eval, spread*

Returns T if U is a vector.

`ZEROP`(**U**:*any*):*boolean* *eval, spread.*

Returns T if U is a number and has the value 0 or 0.0. Returns NIL otherwise.[a]

```
EXPR PROCEDURE ZEROP(U);
   OR(EQN(U, 0), EQN(U, 0.0));
```

[a]The definition in the published report is incorrect as it does not return T for **U** of 0.0.

91.3.2 Functions on Dotted-Pairs

The following are elementary functions on dotted-pairs. All functions in this section which require dotted-pairs as parameters detect a type mismatch error if the actual parameter is not a dotted-pair.

`CAR`(**U**:*dotted-pair*):*any* *eval, spread*

CAR(CONS(a, b)) $\rightarrow$ a. The left part of U is returned. The type mismatch error occurs if U is not a dotted-pair.

`CDR`(**U**:*dotted-pair*):*any* *eval, spread*

CDR(CONS(a, b)) $\rightarrow$ b. The right part of U is returned. The type mismatch error occurs if U is not a dotted-pair.

The composites of CAR and CDR are supported up to 4 levels, namely:

CAAAAR	CAAAR	CAAR
CAAADR	CAADR	CADR
CAADAR	CADAR	CDAR
CAADDR	CADDR	CDDR
CADAAR	CDAAR	
CADADR	CDADR	
CADDAR	CDDAR	
CADDDR	CDDDR	
CDAAAR		
CDAADR		
CDADAR		
CDADDR		
CDDAAR		
CDDADR		
CDDDAR		
CDDDDR		

`CONS`(**U**:*any*, **V**:*any*):*dotted-pair* *eval, spread*

Returns a dotted-pair which is not EQ to anything and has U as its CAR part and V as its CDR part.

`LIST`([**U**:*any*]):*list* *noeval, nospread, or macro*

A list of the evaluation of each element of U is returned. The order of evaluation need not be first to last as the following definition implies.[a]

```
FEXPR PROCEDURE LIST(U);
   EVLIS U;
```

[a] The published report's definition implies a specific ordering.

`RPLACA`(**U**:*dotted-pair*, **V**:*any*):*dotted-pair* *eval, spread*

The CAR portion of the dotted-pair U is replaced by V. If dotted-pair U is (a . b) then (V . b) is returned. The type mismatch error occurs if U is not a dotted-pair.

`RPLACD`(**U**:*dotted-pair*, **V**:*any*):*dotted-pair* *eval, spread*

The CDR portion of the dotted-pair U is replaced by V. If dotted-pair U is (a . b) then (a . V) is returned. The type mismatch error occurs if U is not a dotted-pair.

91.3.3 Identifiers

The following functions deal with identifiers and the OBLIST, the structure of which is not defined. The function of the OBLIST is to provide a symbol table for identifiers created during input. Identifiers created by READ which have the same characters will therefore refer to the same object (see the EQ function in "Elementary Predicates", section 91.3.1 on page 622).

`COMPRESS`(**U**:*id-list*):{*atom-vector*} *eval, spread*

U is a list of single character identifiers which is built into a Standard LISP entity and returned. Recognized are numbers, strings, and identifiers with the escape character prefixing special characters. The formats of these items appear in "Primitive Data Types" section 91.2.1 on page 615. Identifiers are not interned on the OBLIST. Function pointers may be compressed but this is an undefined use. If an entity cannot be parsed out of U or characters are left over after parsing an error occurs:

```
***** Poorly formed atom in COMPRESS
```

`EXPLODE`(**U**:{*atom*}-{*vector*}):*id-list* *eval, spread*

Returned is a list of interned characters representing the characters to print of the value of U. The primitive data types have these formats:

integer Leading zeroes are suppressed and a minus sign prefixes the digits if the integer is negative.

floating The value appears in the format [-]0.nn...nnE[-]mm if the magnitude of the number is too large or small to display in [-]nnnn.nnnn format. The crossover point is determined by the implementation.

id The characters of the print name of the identifier are produced with special characters prefixed with the escape character.

string The characters of the string are produced surrounded by double quotes "...".

function-pointer The value of the function-pointer is created as a list of characters conforming to the conventions of the system site.

The type mismatch error occurs if U is not a number, identifier, string, or function-pointer.

`GENSYM`():*identifier* *eval, spread*

Creates an identifier which is not interned on the OBLIST and consequently not EQ to anything else.

`INTERN`(**U**:{*id,string*}):*id* *eval, spread*

INTERN searches the OBLIST for an identifier with the same print name as U and returns the identifier on the OBLIST if a match is found. Any properties and global values associated with U may be lost. If U does not match any entry, a new one is created and returned. If U has more than the maximum number of characters permitted by the implementation (the minimum number is 24) an error occurs:

```
***** Too many characters to INTERN
```

`REMOB`(**U**:*id*):*id* *eval, spread*

If U is present on the OBLIST it is removed. This does not affect U having properties, flags, functions and the like. U is returned.

91.3.4 Property List Functions

With each id in the system is a "property list", a set of entities which are associated with the id for fast access. These entities are called "flags" if their use gives the id a single valued property, and "properties" if the id is to have a multivalued attribute: an indicator with a property.

Flags and indicators may clash, consequently care should be taken to avoid this occurrence. Flagging X with an id which already is an indicator for X may result in that indicator and associated property being lost. Likewise, adding an indicator which is the same id as a flag may result in the flag being destroyed.

`FLAG`(**U**:*id-list*, **V**:*id*):*NIL* *eval, spread*

U is a list of ids which are flagged with V. The effect of FLAG is that FLAGP will have the value T for those ids of U which were flagged. Both V and all the elements of U must be identifiers or the type mismatch error occurs.

`FLAGP`(**U**:*any*, **V**:*any*):*boolean* *eval, spread*

Returns T if U has been previously flagged with V, else NIL. Returns NIL if either U or V is not an id.

`GET`(**U**:*any*, **IND**:*any*):*any* *eval, spread*

Returns the property associated with indicator IND from the property list of U. If U does not have indicator IND, NIL is returned. GET cannot be used to access functions (use GETD instead).

`PUT`(**U**:*id*, **IND**:*id*, **PROP**:*any*):*any* *eval, spread*

The indicator IND with the property PROP is placed on the property list of the id U. If the action of PUT occurs, the value of PROP is returned. If either of U and IND are not ids the type mismatch error will occur and no property will be placed. PUT cannot be used to define functions (use PUTD instead).

`REMFLAG`(**U**:*any-list*, **V**:*id*):*NIL* *eval, spread*

Removes the flag V from the property list of each member of the list U. Both V and all the elements of U must be ids or the type mismatch error will occur.

`REMPROP`(**U**:*any*, **IND**:*any*):*any* *eval, spread*

Removes the property with indicator IND from the property list of U. Returns the removed property or NIL if there was no such indicator.

91.3.5 Function Definition

Functions in Standard LISP are global entities. To avoid function-variable naming clashes no variable may have the same name as a function.

`DE`(**FNAME**:*id*, **PARAMS**:*id-list*, **FN**:*any*):*id* *noeval, nospread*

The function FN with the formal parameter list PARAMS is added to the set of defined functions with the name FNAME. Any previous definitions of the function are lost. The function created is of type EXPR. If the !*COMP variable is non-NIL, the EXPR is first compiled. The name of the defined function is returned.

```
FEXPR PROCEDURE DE(U);
   PUTD(CAR U, 'EXPR, LIST('LAMBDA, CADR U, CADDR U));
```

`DF`(**FNAME**:*id*, **PARAM**:*id-list*, **FN**:*any*):*id* *noeval, nospread*

The function FN with formal parameter PARAM is added to the set of defined functions with the name FNAME. Any previous definitions of the function are lost. The function created is of type FEXPR. If the !*COMP variable is T the FEXPR is first compiled. The name of the defined function is returned.

```
FEXPR PROCEDURE DF(U);
   PUTD(CAR U, 'FEXPR, LIST('LAMBDA, CADR U, CADDR U));
```

`DM`(**MNAME**:*id*, **PARAM**:*id-list*, **FN**:*any*):*id* *noeval, nospread*

The macro FN with the formal parameter PARAM is added to the set of defined functions with the name MNAME. Any previous definitions of the function are overwritten. The function created is of type MACRO. The name of the macro is returned.

```
FEXPR PROCEDURE DM(U);
   PUTD(CAR U, 'MACRO, LIST('LAMBDA, CADR U, CADDR U));
```

`GETD`(**FNAME**:*any*):{NIL, *dotted-pair*} *eval, spread*

If FNAME is not the name of a defined function, NIL is returned. If FNAME is a defined function then the dotted-pair

(**TYPE**:*ftype* . **DEF**:{*function-pointer, lambda*})

is returned.

`PUTD`(**FNAME**:*id*, **TYPE**:*ftype*, **BODY**:*function*):*id* *eval, spread*

Creates a function with name FNAME and definition BODY of type TYPE. If PUTD succeeds the name of the defined function is returned. The effect of PUTD is that GETD will return a dotted-pair with the functions type and definition. Likewise the GLOBALP predicate will return T when queried with the function name.
If the function FNAME has already been declared as a GLOBAL or FLUID variable the error:

```
***** FNAME is a non-local variable
```

occurs and the function will not be defined. If function FNAME already exists a warning message will appear:

```
*** FNAME redefined
```

The function defined by PUTD will be compiled before definition if the !*COMP global variable is non-NIL.

`REMD`(**FNAME**:*id*):{NIL, *dotted-pair*} *eval, spread*

Removes the function named FNAME from the set of defined functions. Returns the (ftype . function) dotted-pair or NIL as does GETD. The global/function attribute of FNAME is removed and the name may be used subsequently as a variable.

91.3.6 Variables and Bindings

A variable is a place holder for a Standard LISP entity which is said to be bound to the variable. The scope of a variable is the range over which the variable has a defined value. There are three different binding mechanisms in Standard LISP.

Local Binding This type of binding occurs only in compiled functions. Local variables occur as formal parameters in lambda expressions and as PROG form variables. The binding occurs when a lambda expression is evaluated or when a PROG form is executed. The scope of a local variable is the body of the function in which it is defined.

Global Binding Only one binding of a global variable exists at any time allowing direct access to the value bound to the variable. The scope of a global variable is universal. Variables declared GLOBAL may not appear as parameters in lambda expressions or as PROG form variables. A variable must be declared GLOBAL prior to its use as a global variable since the default type for undeclared variables is FLUID.

Fluid Binding Fluid variables are global in scope but may occur as formal parameters or PROG form variables. In interpreted functions all formal parameters and PROG form variables are considered to have fluid binding until changed to local binding by compilation. When fluid variables are used as parameters they are rebound in such a way that the previous binding may be restored. All references to fluid variables are to the currently active binding.

`FLUID`(**IDLIST**:*id-list*):**NIL** *eval, spread*

The ids in IDLIST are declared as FLUID type variables (ids not previously declared are initialized to NIL). Variables in IDLIST already declared FLUID are ignored. Changing a variable's type from GLOBAL to FLUID is not permissible and results in the error:

```
***** ID cannot be changed to FLUID
```

`FLUIDP`(**U**:*any*):*boolean* *eval, spread*

If U has been declared FLUID (by declaration only) T is returned, otherwise NIL is returned.

`GLOBAL`(**IDLIST**:*id-list*):**NIL** *eval, spread*

The ids of IDLIST are declared global type variables. If an id has not been declared previously it is initialized to NIL. Variables already declared GLOBAL are ignored. Changing a variables type from FLUID to GLOBAL is not permissible and results in the error:

```
***** ID cannot be changed to GLOBAL
```

`GLOBALP`(**U**:*any*):*boolean* *eval, spread*

If U has been declared GLOBAL or is the name of a defined function, T is returned, else NIL is returned.

`SET`(**EXP**:*id*, **VALUE**:*any*):*any* *eval, spread*

EXP must be an identifier or a type mismatch error occurs. The effect of SET is replacement of the item bound to the identifier by VALUE. If the identifier is not a local variable or has not been declared GLOBAL it is automatically declared FLUID with the resulting warning message:

```
*** EXP declared FLUID
```

EXP must not evaluate to T or NIL or an error occurs:

```
***** Cannot change T or NIL
```

`SETQ`(**VARIABLE**:*id*, **VALUE**:*any*):*any* *noeval, nospread*

If VARIABLE is not local or GLOBAL it is by default declared FLUID and the warning message:

```
*** VARIABLE declared FLUID
```

appears. The value of the current binding of VARIABLE is replaced by the value of VALUE. VARIABLE must not be T or NIL or an error occurs:

```
***** Cannot change T or NIL
```

```
MACRO PROCEDURE SETQ(X);
   LIST('SET, LIST('QUOTE, CADR X), CADDR X);
```

`UNFLUID`(**IDLIST**:*id-list*):*NIL* *eval, spread*

The variables in IDLIST that have been declared as FLUID variables are no longer considered as fluid variables. Others are ignored. This affects only compiled functions as free variables in interpreted functions are automatically considered fluid [10].

91.3.7 Program Feature Functions

These functions provide for explicit control sequencing, and the definition of blocks altering the scope of local variables.

`GO`(**LABEL**:*id*) *noeval, nospread*

GO alters the normal flow of control within a PROG function. The next statement of a PROG function to be evaluated is immediately preceded by LABEL. A GO may only appear in the following situations:

1. At the top level of a PROG referencing a label which also appears at the top level of the same PROG.
2. As the consequent of a COND item of a COND appearing on the top level of a PROG.
3. As the consequent of a COND item which appears as the consequent of a COND item to any level.
4. As the last statement of a PROGN which appears at the top level of a PROG or in a PROGN appearing in the consequent of a COND to any level subject to the restrictions of 2 and 3.
5. As the last statement of a PROGN within a PROGN or as the consequent of a COND in a PROGN to any level subject to the restrictions of 2, 3 and 4.

If LABEL does not appear at the top level of the PROG in which the GO appears, an error occurs:

```
***** LABEL is not a known label
```

If the GO has been placed in a position not defined by rules 1-5, another error is detected:

```
***** Illegal use of GO to LABEL
```

`PROG`(**VARS**:*id-list*, [**PROGRAM**:{*id, any*}]):*any* *noeval, nospread*

VARS is a list of ids which are considered fluid when the PROG is interpreted and local when compiled (see "Variables and Bindings", section 91.3.6 on page 633). The PROGs variables are allocated space when the PROG form is invoked and are deallocated when the PROG is exited. PROG variables are initialized to NIL. The PROGRAM is a set of expressions to be evaluated in order of their appearance in the PROG function. Identifiers appearing in the top level of the PROGRAM are labels which can be referenced by GO. The value returned by the PROG function is determined by a RETURN function or NIL if the PROG "falls through".

`PROGN`([**U**:*any*]):*any* *noeval, nospread*

U is a set of expressions which are executed sequentially. The value returned is the value of the last expression.

`PROG2`(A:any, B:any)*any* *eval, spread*

Returns the value of B.

```
EXPR PROCEDURE PROG2(A, B);
   B;
```

`RETURN`(**U**:*any*) *eval, spread*

Within a PROG, RETURN terminates the evaluation of a PROG and returns U as the value of the PROG. The restrictions on the placement of RETURN are exactly those of GO. Improper placement of RETURN results in the error:

```
***** Illegal use of RETURN
```

91.3.8 Error Handling

`ERROR`(**NUMBER**:*integer*, **MESSAGE**:*any*) *eval, spread*

NUMBER and MESSAGE are passed back to a surrounding ERRORSET (the Standard LISP reader has an ERRORSET). MESSAGE is placed in the global variable EMSG!* and the error number becomes the value of the surrounding ERRORSET. FLUID variables and local bindings are unbound to return to the environment of the ERRORSET. Global variables are not affected by the process.

`ERRORSET`(**U**:*any*, **MSGP**:*boolean*, **TR**:*boolean*):*any* *eval, spread*

If an error occurs during the evaluation of U, the value of NUMBER from the ERROR call is returned as the value of ERRORSET. In addition, if the value of MSGP is non-NIL, the MESSAGE from the ERROR call is displayed upon both the standard output device and the currently selected output device unless the standard output device is not open. The message appears prefixed with 5 asterisks. The MESSAGE list is displayed without top level parentheses. The MESSAGE from the ERROR call will be available in the global variable EMSG!*. The exact format of error messages generated by Standard LISP functions described in this document are not fixed and should not be relied upon to be in any particular form. Likewise, error numbers generated by Standard LISP functions are implementation dependent.

If no error occurs during the evaluation of U, the value of (LIST (EVAL U)) is returned.

If an error has been signaled and the value of TR is non-NIL a traceback sequence will be initiated on the selected output device. The traceback will display information such as unbindings of FLUID variables, argument lists and so on in an implementation dependent format.

91.3.9 Vectors

Vectors are structured entities in which random elements may be accessed with an integer index. A vector has a single dimension. Its maximum size is determined by the implementation and available space. A suggested input "vector notation" is defined in "Classes of Primitive Data Types", section 91.2.2 on page 619 and output with EXPLODE, "Identifiers" section 91.3.3 on page 627.

`GETV`(**V**:*vector*, **INDEX**:*integer*):*any* *eval, spread*

Returns the value stored at position INDEX of the vector V. The type mismatch error may occur. An error occurs if the INDEX does not lie within 0... UPBV(V) inclusive:

```
***** INDEX subscript is out of range
```

`MKVECT`(**UPLIM**:*integer*):*vector* *eval, spread*

Defines and allocates space for a vector with UPLIM+1 elements accessed as 0... UPLIM. Each element is initialized to NIL. An error will occur if UPLIM is < 0 or there is not enough space for a vector of this size:

```
***** A vector of size UPLIM cannot be allocated
```

`PUTV`(**V**:*vector*, **INDEX**:*integer*, **VALUE**:*any*):*any* *eval, spread*

Stores VALUE into the vector V at position INDEX. VALUE is returned. The type mismatch error may occur. If INDEX does not lie in 0... UPBV(V) an error occurs:

```
***** INDEX subscript is out of range
```

`UPBV`(**U**:*any*):NIL,*integer* *eval, spread*

Returns the upper limit of U if U is a vector, or NIL if it is not.

91.3.10 Boolean Functions and Conditionals

`AND`([**U**:*any*]):*extra-boolean* *noeval, nospread*

AND evaluates each U until a value of NIL is found or the end of the list is encountered. If a non-NIL value is the last value it is returned, or NIL is returned.

```
FEXPR PROCEDURE AND(U);
BEGIN
   IF NULL U THEN RETURN NIL;
LOOP: IF NULL CDR U THEN RETURN EVAL CAR U
         ELSE IF NULL EVAL CAR U THEN RETURN NIL;
     U := CDR U;
     GO LOOP
END;
```

`COND`([**U**:*cond-form*]):*any* *noeval, nospread*

The antecedents of all U's are evaluated in order of their appearance until a non-NIL value is encountered. The consequent of the selected U is evaluated and becomes the value of the COND. The consequent may also contain the special functions GO and RETURN subject to the restraints given for these functions in "Program Feature Functions", section 91.3.7 on page 635. In these cases COND does not have a defined value, but rather an effect. If no antecedent is non-NIL the value of COND is NIL. An error is detected if a U is improperly formed:

```
***** Improper cond-form as argument of COND
```

`NOT`(**U**:*any*):*boolean* *eval, spread*

If U is NIL, return T else return NIL (same as function NULL).

```
EXPR PROCEDURE NOT(U);
   U EQ NIL;
```

`OR`([**U**:*any*]):*extra-boolean* *noeval, nospread*

U is any number of expressions which are evaluated in order of their appearance. When one is found to be non-NIL it is returned as the value of OR. If all are NIL, NIL is returned.

```
FEXPR PROCEDURE OR(U);
BEGIN SCALAR X;
LOOP: IF NULL U THEN RETURN NIL
         ELSE IF (X := EVAL CAR U) THEN RETURN X;
     U := CDR U;
     GO LOOP
END;
```

91.3.11 Arithmetic Functions

Conversions between numeric types are provided explicitly by the FIX and FLOAT functions and implicitly by any multi-parameter arithmetic function which receives mixed types of arguments. A conversion from fixed to floating point numbers may result in a loss of precision without a warning message being generated. Since integers may have a greater magnitude that that permitted for floating numbers, an error may be signaled when the attempted conversion cannot be done. Because the magnitude of integers is unlimited the conversion of a floating point number to a fixed number is always possible, the only loss of precision being the digits to the right of the decimal point which are truncated. If a function receives mixed types of arguments the general rule will have the fixed numbers converted to floating before arithmetic operations are performed. In all cases an error occurs if the parameter to an arithmetic function is not a number:

```
***** XXX parameter to FUNCTION is not a number
```

XXX is the value of the parameter at fault and FUNCTION is the name of the function that detected the error. Exceptions to the rule are noted where they occur.

`ABS`(**U**:*number*):*number* *eval, spread*

Returns the absolute value of its argument.

```
EXPR PROCEDURE ABS(U);
   IF LESSP(U, 0) THEN MINUS(U) ELSE U;
```

`ADD1`(**U**:*number*):*number* *eval, spread*

Returns the value of U plus 1 of the same type as U (fixed or floating).

```
EXPR PROCEDURE ADD1(U);
   PLUS2(U, 1);
```

`DIFFERENCE`(**U**:*number*, **V**:*number*):*number* *eval, spread*

The value U - V is returned.

`DIVIDE`(**U**:*number*, **V**:*number*):*dotted-pair* *eval, spread*

The dotted-pair (quotient . remainder) is returned. The quotient part is computed the same as by QUOTIENT and the remainder the same as by REMAINDER. An error occurs if division by zero is attempted:

```
***** Attempt to divide by 0 in DIVIDE
```

```
EXPR PROCEDURE DIVIDE(U, V);
   (QUOTIENT(U, V) . REMAINDER(U, V));
```

`EXPT`(**U**:*number*, **V**:*integer*):*number* *eval, spread*

Returns U raised to the V power. A floating point U to an integer power V does not have V changed to a floating number before exponentiation.

`FIX`(**U**:*number*):*integer* *eval, spread*

Returns an integer which corresponds to the truncated value of U. The result of conversion must retain all significant portions of U. If U is an integer it is returned unchanged.

`FLOAT`(**U**:*number*):*floating* *eval, spread*

The floating point number corresponding to the value of the argument U is returned. Some of the least significant digits of an integer may be lost do to the implementation of floating point numbers. FLOAT of a floating point number returns the number unchanged. If U is too large to represent in floating point an error occurs:

```
***** Argument to FLOAT is too large
```

`GREATERP`(**U**:*number*, **V**:*number*):*boolean* *eval, spread*

Returns T if U is strictly greater than V, otherwise returns NIL.

`LESSP`(**U**:*number*, **V**:*number*):*boolean* *eval, spread*

Returns T if U is strictly less than V, otherwise returns NIL.

`MAX`([**U**:*number*]):*number* *noeval, nospread, or macro*

Returns the largest of the values in U. If two or more values are the same the first is returned.

```
MACRO PROCEDURE MAX(U);
   EXPAND(CDR U, 'MAX2);
```

`MAX2`(**U**:*number*, **V**:*number*):*number* *eval, spread*

Returns the larger of U and V. If U and V are the same value U is returned (U and V might be of different types).

```
EXPR PROCEDURE MAX2(U, V);
   IF LESSP(U, V) THEN V ELSE U;
```

`MIN`([**U**:*number*]):*number* *noeval, nospread, or macro*

Returns the smallest of the values in U. If two or more values are the same the first of these is returned.

```
MACRO PROCEDURE MIN(U);
   EXPAND(CDR U, 'MIN2);
```

`MIN2`(**U**:*number*, **V**:*number*):*number* *eval, spread*

Returns the smaller of its arguments. If U and V are the same value, U is returned (U and V might be of different types).

```
EXPR PROCEDURE MIN2(U, V);
   IF GREATERP(U, V) THEN V ELSE U;
```

`MINUS`(**U**:*number*):*number* *eval, spread*

Returns -U.

```
EXPR PROCEDURE MINUS(U);
   DIFFERENCE(0, U);
```

`PLUS`([**U**:*number*]):*number* *noeval, nospread, or macro*

Forms the sum of all its arguments.

```
MACRO PROCEDURE PLUS(U);
   EXPAND(CDR U, 'PLUS2);
```

`PLUS2`(**U**:*number*, **V**:*number*):*number* *eval, spread*

Returns the sum of U and V.

`QUOTIENT`(**U**:*number*, **V**:*number*):*number* *eval, spread*

The quotient of U divided by V is returned. Division of two positive or two negative integers is conventional. When both U and V are integers and exactly one of them is negative the value returned is the negative truncation of the absolute value of U divided by the absolute value of V. An error occurs if division by zero is attempted:

```
***** Attempt to divide by 0 in QUOTIENT
```

`REMAINDER`(**U**:*number*, **V**:*number*):*number* *eval, spread*

If both U and V are integers the result is the integer remainder of U divided by V. If either parameter is floating point, the result is the difference between U and V*(U/V) all in floating point. If either number is negative the remainder is negative. If both are positive or both are negative the remainder is positive. An error occurs if V is zero:

```
***** Attempt to divide by 0 in REMAINDER
```

```
EXPR PROCEDURE REMAINDER(U, V);
   DIFFERENCE(U, TIMES2(QUOTIENT(U, V), V));
```

`SUB1`(**U**:*number*):*number* *eval, spread*

Returns the value of U less 1. If U is a FLOAT type number, the value returned is U less 1.0.

```
EXPR PROCEDURE SUB1(U);
   DIFFERENCE(U, 1);
```

`TIMES`([**U**:*number*]):*number* *noeval, nospread, or macro*

Returns the product of all its arguments.

```
MACRO PROCEDURE TIMES(U);
   EXPAND(CDR U, 'TIMES2);
```

`TIMES2`(**U**:*number*, **V**:*number*):*number* *eval, spread*

Returns the product of U and V.

91.3.12 MAP Composite Functions

`MAP`(**X**:*list*, F**N**:*function*):*any* *eval, spread*

Applies FN to successive CDR segments of X. NIL is returned.

```
EXPR PROCEDURE MAP(X, FN);
   WHILE X DO << FN X; X := CDR X >>;
```

`MAPC`(X:list, FN:function):*any* *eval, spread*

FN is applied to successive CAR segments of list X. NIL is returned.

```
EXPR PROCEDURE MAPC(X, FN);
   WHILE X DO << FN CAR X; X := CDR X >>;
```

`MAPCAN`(X:list, FN:function):*any* *eval, spread*

A concatenated list of FN applied to successive CAR elements of X is returned.

```
EXPR PROCEDURE MAPCAN(X, FN);
   IFNULL X THEN NIL
     ELSE NCONC(FN CAR X, MAPCAN(CDR X, FN));
```

`MAPCAR`(X:list, FN:function):*any* *eval, spread*

Returned is a constructed list of FN applied to each CAR of list X.

```
EXPR PROCEDURE MAPCAR(X, FN);
   IFNULL X THEN NIL
     ELSE FN CAR X . MAPCAR(CDR X, FN);
```

`MAPCON`(X:list, FN:function):*any* *eval, spread*

Returned is a concatenated list of FN applied to successive CDR segments of X.

```
EXPR PROCEDURE MAPCON(X, FN);
   IFNULL X THEN NIL
     ELSE NCONC(FN X, MAPCON(CDR X, FN));
```

`MAPLIST`(X:list, FN:function):*any* *eval, spread*

Returns a constructed list of FN applied to successive CDR segments of X.

```
EXPR PROCEDURE MAPLIST(X, FN);
   IFNULL X THEN NIL
     ELSE FN X . MAPLIST(CDR X, FN);
```

91.3.13 Composite Functions

`APPEND`(**U**:*list*, **V**:*list*):*list* *eval, spread*

Returns a constructed list in which the last element of U is followed by the first element of V. The list U is copied, V is not.

```
EXPR PROCEDURE APPEND(U, V);
   IFNULL U THEN V
     ELSE CAR U . APPEND(CDR U, V);
```

`ASSOC`(**U**:*any*, **V**:*alist*):{*dotted-pair*, NIL} *eval, spread*

If U occurs as the CAR portion of an element of the alist V, the dotted-pair in which U occurred is returned, else NIL is returned. ASSOC might not detect a poorly formed alist so an invalid construction may be detected by CAR or CDR.

```
EXPR PROCEDURE ASSOC(U, V);
   IF NULL V THEN NIL
      ELSE IF ATOM CAR V THEN
           ERROR(000, LIST(V, "is a poorly formed alist"))
      ELSE IF U = CAAR V THEN CAR V
      ELSE ASSOC(U, CDR V);
```

`DEFLIST`(**U**:*dlist*, **IND**:*id*):*list* *eval, spread*

A "dlist" is a list in which each element is a two element list: (ID:id PROP:any). Each ID in U has the indicator IND with property PROP placed on its property list by the PUT function. The value of DEFLIST is a list of the first elements of each two element list. Like PUT, DEFLIST may not be used to define functions.

```
EXPR PROCEDURE DEFLIST(U, IND);
   IF NULL U THEN NIL
     ELSE << PUT(CAAR U, IND, CADAR U);
              CAAR U >> .  DEFLIST(CDR U, IND);
```

`DELETE`(**U**:*any*, **V**:*list*):*list* *eval, spread*

Returns V with the first top level occurrence of U removed from it.

```
EXPR PROCEDURE DELETE(U, V);
   IF NULL V THEN NIL
     ELSE IF CAR V = U THEN CDR V
     ELSE CAR V . DELETE(U, CDR V);
```

`DIGIT`(**U**:*any*):*boolean* *eval, spread*

Returns T if U is a digit, otherwise NIL.

```
EXPR PROCEDURE DIGIT(U);
   IF MEMQ(U, '(!0 !1 !2 !3 !4 !5 !6 !7 !8 !9))
     THEN T ELSE NIL;
```

`LENGTH`(**X**:*any*):*integer* *eval, spread*

The top level length of the list X is returned.

```
EXPR PROCEDURE LENGTH(X);
   IF ATOM X THEN 0
     ELSE PLUS(1, LENGTH CDR X);
```

`LITER`(**U**:*any*):*boolean* *eval, spread*

Returns T if U is a character of the alphabet, NIL otherwise.[a]

```
EXPR PROCEDURE LITER(U);
   IF MEMQ(U, '(!A !B !C !D !E !F !G !H !I !J !K !L !M
                !N !O !P !Q !R !S !T !U !V !W !X !Y !Z
                !a !b !c !d !e !f !g !h !i !j !k !l !m
                !n !o !p !q !r !s !t !u !v !w !x !y !z))
      THEN T ELSE NIL;
```

[a]The published report omits escape characters. These are required for both upper and lower case as some systems default to lower.

`MEMBER`(**A**:*any*, **B**:*list*):*extra-boolean* *eval, spread*

Returns NIL if A is not a member of list B, returns the remainder of B whose first element is A.

```
EXPR PROCEDURE MEMBER(A, B);
   IF NULL B THEN NIL
     ELSE IF A = CAR B THEN B
     ELSE MEMBER(A, CDR B);
```

`MEMQ`(**A**:*any*, **B**:*list*):*extra-boolean* *eval, spread*

Same as MEMBER but an EQ check is used for comparison.

```
EXPR PROCEDURE MEMQ(A, B);
   IF NULL B THEN NIL
      ELSE IF A EQ CAR B THEN B
      ELSE MEMQ(A, CDR B);
```

`NCONC`(**U**:*list*, **V**:*list*):*list* *eval, spread*

Concatenates V to U without copying U. The last CDR of U is modified to point to V.

```
EXPR PROCEDURE NCONC(U, V);
BEGIN SCALAR W;
     IF NULL U THEN RETURN V;
     W := U;
     WHILE CDR W DO W := CDR W;
     RPLACD(W, V);
     RETURN U
END;
```

`PAIR`(**U**:*list*, **V**:*list*):*alist* *eval, spread*

U and V are lists which must have an identical number of elements. If not, an error occurs (the 000 used in the ERROR call is arbitrary and need not be adhered to). Returned is a list where each element is a dotted-pair, the CAR of the pair being from U, and the CDR the corresponding element from V.

```
EXPR PROCEDURE PAIR(U, V);
   IF AND(U, V) THEN (CAR U . CAR V) . PAIR(CDR U, CDR V)
     ELSE IF OR(U, V) THEN ERROR(000,
         "Different length lists in PAIR")
     ELSE NIL;
```

`REVERSE`(**U**:*list*):*list* *eval, spread*

Returns a copy of the top level of U in reverse order.

```
EXPR PROCEDURE REVERSE(U);
BEGIN SCALAR W;
     WHILE U DO << W := CAR U . W;
                  U := CDR U >>;
     RETURN W
END;
```

`SASSOC`(**U**:*any*, **V**:*alist*, **FN**:*function*):*any* *eval, spread*

Searches the alist V for an occurrence of U. If U is not in the alist the evaluation of function FN is returned.

```
EXPR PROCEDURE SASSOC(U, V, FN);
   IF NULL V THEN FN()
     ELSE IF U = CAAR V THEN CAR V
     ELSE SASSOC(U, CDR V, FN);
```

`SUBLIS`(**X**:*alist*, **Y**:*any*):*any* *eval, spread*

The value returned is the result of substituting the CDR of each element of the alist X for every occurrence of the CAR part of that element in Y.

```
EXPR PROCEDURE SUBLIS(X, Y);
  IF NULL X THEN Y
     ELSE BEGIN SCALAR U;
                U := ASSOC(Y, X);
                RETURN IF U THEN CDR U
                       ELSE IF ATOM Y THEN Y
                       ELSE SUBLIS(X, CAR Y) .
                            SUBLIS(X, CDR Y)
                END;
```

`SUBST`(**U**:*any*, **V**:*any*, **W**:*any*):*any* *eval, spread*

The value returned is the result of substituting U for all occurrences of V in W.

```
EXPR PROCEDURE SUBST(U, V, W);
   IF NULL W THEN NIL
     ELSE IF V = W THEN U
     ELSE IF ATOM W THEN W
     ELSE SUBST(U, V, CAR W) . SUBST(U, V, CDR W);
```

91.3.14 The Interpreter

APPLY(**FN**:{*id,function*}, **ARGS**:*any-list*):*any* *eval, spread*

APPLY returns the value of FN with actual parameters ARGS. The actual parameters in ARGS are already in the form required for binding to the formal parameters of FN. Implementation specific portions described in English are enclosed in boxes.

```
EXPR PROCEDURE APPLY(FN, ARGS);
BEGIN SCALAR DEFN;
    IF CODEP FN THEN RETURN
       | Spread the actual parameters in ARGS       |
       | following the conventions:  for calling    |
       | functions, transfer to the entry point     | ;
       | of the function, and return the value      |
       | returned by the function.                  |
    IF IDP FN THEN RETURN
       IF NULL(DEFN := GETD FN) THEN
          ERROR(000, LIST(FN, "is an undefined function"))
       ELSE IF CAR DEFN EQ 'EXPR THEN
          APPLY(CDR DEFN, ARGS)
       ELSE ERROR(000,
          LIST(FN, "cannot be evaluated by APPLY"));
    IF OR(ATOM FN, NOT(CAR FN EQ 'LAMBDA)) THEN
       ERROR(000,
       LIST(FN, "cannot be evaluated by APPLY"));
    RETURN
       | Bind the actual parameters in ARGS to      |
       | the formal parameters of the lambda        |
       | expression.  If the two lists are not      |
       | of equal length then ERROR(000, "Number    |
       | of parameters do not match"); The value    |
       | returned is EVAL CADDR FN.                 |
END;
```

`EVAL`(**U**:*any*):*any* *eval, spread*

The value of the expression U is computed. Error numbers are arbitrary. Portions of EVAL involving machine specific coding are expressed in English enclosed in boxes.

```
EXPR PROCEDURE EVAL(U);
BEGIN SCALAR FN;
     IF CONSTANTP U THEN RETURN U;
     IF IDP U THEN RETURN
        U is an id.  Return the value most
        currently bound to U or if there
        is no such binding:  ERROR(000,
        LIST("Unbound:", U));
     IF PAIRP CAR U THEN RETURN
        IF CAAR U EQ 'LAMBDA THEN APPLY(CAR U, EVLIS CDR U)
        ELSE ERROR(000, LIST(CAR U,
                   "improperly formed LAMBDA expression"))
        ELSE IF CODEP CAR U THEN
                   RETURN APPLY(CAR U, EVLIS CDR U);
     FN := GETD CAR U;
     IF NULL FN THEN
        ERROR(000, LIST(CAR U, "is an undefined function"))
     ELSE IF CAR FN EQ 'EXPR THEN
        RETURN APPLY(CDR FN, EVLIS CDR U)
     ELSE IF CAR FN EQ 'FEXPR THEN
        RETURN APPLY(CDR FN, LIST CDR U)
     ELSE IF CAR FN EQ 'MACRO THEN
        RETURN EVAL APPLY(CDR FN, LIST U)
END;
```

`EVLIS`(**U**:*any-list*):*any-list* *eval, spread*

EVLIS returns a list of the evaluation of each element of U.

```
EXPR PROCEDURE EVLIS(U);
   IF NULL U THEN NIL
     ELSE EVAL CAR U . EVLIS CDR U;
```

`EXPAND`(**L**:*list*, **FN**:*function*):*list* *eval, spread*

FN is a defined function of two arguments to be used in the expansion of a MACRO. EXPAND returns a list in the form:

(FN L_0 (FN L_1 ... (FN L_{n-1} L_n) ...))

where n is the number of elements in L, L_i is the ith element of L.

```
EXPR PROCEDURE EXPAND(L,FN);
   IF NULL CDR L THEN CAR L
     ELSE LIST(FN, CAR L, EXPAND(CDR L, FN));
```

`FUNCTION`(**FN**:*function*):*function* *noeval, nospread*

The function FN is to be passed to another function. If FN is to have side effects its free variables must be fluid or global. FUNCTION is like QUOTE but its argument may be affected by compilation. We do not consider FUNARGs in this report.

`QUOTE`(U:any):*any* *noeval, nospread*

Stops evaluation and returns U unevaluated.

```
FEXPR PROCEDURE QUOTE(U);
    CAR U;
```

91.3.15 Input and Output

The user normally communicates with Standard LISP through "standard devices". The default devices are selected in accordance with the conventions of the implementation site. Other input and output devices or files may be selected for reading and writing using the functions described herein.

`CLOSE`(**FILEHANDLE**:*any*):*any* *eval, spread*

Closes the file with the internal name FILEHANDLE writing any necessary end of file marks and such. The value of FILEHANDLE is that returned by the corresponding OPEN. The value returned is the value of FILEHANDLE. An error occurs if the file can not be closed.

```
***** FILEHANDLE could not be closed
```

`EJECT`():NIL *eval, spread*

Skip to the top of the next output page. Automatic EJECTs are executed by the print functions when the length set by the PAGE-LENGTH function is exceeded.

`LINELENGTH`(**LEN**:{*integer*, NIL}):*integer* *eval, spread*

If LEN is an integer the maximum line length to be printed before the print functions initiate an automatic TERPRI is set to the value LEN. No initial Standard LISP line length is assumed. The previous line length is returned except when LEN is NIL. This special case returns the current line length and does not cause it to be reset. An error occurs if the requested line length is too large for the currently selected output file or LEN is negative or zero.

```
***** LEN is an invalid line length
```

`LPOSN`():*integer* *eval, spread*

Returns the number of lines printed on the current page. At the top of a page, 0 is returned.

`OPEN`(**FILE**:*any*, **HOW**:*id*):*any* *eval, spread*

Open the file with the system dependent name FILE for output if HOW is EQ to OUTPUT, or input if HOW is EQ to INPUT. If the file is opened successfully, a value which is internally associated with the file is returned. This value must be saved for use by RDS and WRS. An error occurs if HOW is something other than INPUT or OUTPUT or the file can't be opened.

```
***** HOW is not option for OPEN
 ***** FILE could not be opened
```

`PAGELENGTH`(**LEN**:{*integer*, NIL}):*integer* *eval, spread*

Sets the vertical length (in lines) of an output page. Automatic page EJECTs are executed by the print functions when this length is reached. The initial vertical length is implementation specific. The previous page length is returned. If LEN is 0, no automatic page ejects will occur.

`POSN`():*integer* *eval, spread*

Returns the number of characters in the output buffer. When the buffer is empty, 0 is returned.

`PRINC`(**U**:*id*):*id* *eval, spread*

U must be a single character id such as produced by EXPLODE or read by READCH or the value of !$EOL!$. The effect is the character U displayed upon the currently selected output device. The value of !$EOL!$ causes termination of the current line like a call to TERPRI.

`PRINT`(**U**:*any*):*any* *eval, spread*

Displays U in READ readable format and terminates the print line. The value of U is returned.

```
EXPR PROCEDURE PRINT(U);
     << PRIN1 U; TERPRI(); U >>;
```

`PRIN1`(**U**:*any*):*any* *eval, spread*

U is displayed in a READ readable form. The format of display is the result of EXPLODE expansion; special characters are prefixed with the escape character !, and strings are enclosed in "...". Lists are displayed in list-notation and vectors in vector-notation.

`PRIN2`(**U**:*any*):*any* *eval, spread*

U is displayed upon the currently selected print device but output is not READ readable. The value of U is returned. Items are displayed as described in the EXPLODE function with the exceptions that the escape character does not prefix special characters and strings are not enclosed in "...". Lists are displayed in list-notation and vectors in vector-notation. The value of U is returned.

`RDS`(**FILEHANDLE**:*any*):*any* *eval, spread*

Input from the currently selected input file is suspended and further input comes from the file named. FILEHANDLE is a system dependent internal name which is a value returned by OPEN. If FILEHANDLE is NIL the standard input device is selected. When end of file is reached on a non-standard input device, the standard input device is reselected. When end of file occurs on the standard input device the Standard LISP reader terminates. RDS returns the internal name of the previously selected input file.

```
***** FILEHANDLE could not be selected for input
```

`READ()`:*any*

The next expression from the file currently selected for input. Valid input forms are: vector-notation, dot-notation, list-notation, numbers, function-pointers, strings, and identifiers with escape characters. Identifiers are interned onW the OBLIST (see the INTERN function in "Identifiers", section 91.3.3 on page 627). READ returns the value of !$EOF!$ when the end of the currently selected input file is reached.

`READCH()`:*id*

Returns the next interned character from the file currently selected for input. Two special cases occur. If all the characters in an input record have been read, the value of !$EOL!$ is returned. If the file selected for input has all been read the value of !$EOF!$ is returned. Comments delimited by % and end-of-line are not transparent to READCH.

`TERPRI()`:**NIL**

The current print line is terminated.

`WRS`(**FILEHANDLE**:*any*):*any* *eval, spread*

Output to the currently active output file is suspended and further output is directed to the file named. FILEHANDLE is an internal name which is returned by OPEN. The file named must have been opened for output. If FILEHANDLE is NIL the standard output device is selected. WRS returns the internal name of the previously selected output file.

```
***** FILEHANDLE could not be selected for output
```

91.3.16 LISP Reader

An EVAL read loop has been chosen to drive a Standard LISP system to provide a continuity in functional syntax. Choices of messages and the amount of extra information displayed are decisions left to the implementor.

```
EXPR PROCEDURE STANDARD!-LISP();
BEGIN SCALAR VALUE;
     RDS NIL; WRS NIL;
     PRIN2 "Standard LISP"; TERPRI();
     WHILE T DO
        << PRIN2 "EVAL:"; TERPRI();
            VALUE := ERRORSET(QUOTE EVAL READ(), T, T);
            IF NOT ATOM VALUE THEN PRINT CAR VALUE;
            TERPRI() >>;
END;
```

`QUIT()`
Causes termination of the LISP reader and control to be transferred to the operating system.

91.4 System GLOBAL Variables

These variables provide global control of the LISP system, or implement values which are constant throughout execution.[2]

***COMP** = NIL *global*
The value of !*COMP controls whether or not PUTD compiles the function defined in its arguments before defining it. If !*COMP is NIL the function is defined as an xEXPR. If !*COMP is something else the function is first compiled. Compilation will produce certain changes in the semantics of functions particularly FLUID type access.

[2]The published document does not specify that all these are GLOBAL.

EMSG* = NIL *global*

Will contain the MESSAGE generated by the last ERROR call (see "Error Handling" section 91.3.8 on page 638).

EOF = <*an uninterned identifier*> *global*

The value of !$EOF!$ is returned by all input functions when the end of the currently selected input file is reached.

EOL = <*an uninterned identifier*> *global*

The value of !$EOL!$ is returned by READCH when it reaches the end of a logical input record. Likewise PRINC will terminate its current line (like a call to TERPRI) when !$EOL!$ is its argument.

***GC** = NIL *global*

!*GC controls the printing of garbage collector messages. If NIL no indication of garbage collection may occur. If non-NIL various system dependent messages may be displayed.

NIL = NIL *global*

NIL is a special global variable. It is protected from being modified by SET or SETQ.

***RAISE** = NIL *global*

If !*RAISE is non-NIL all characters input through Standard LISP input/output functions will be raised to upper case. If !*RAISE is NIL characters will be input as is.

T = T *global*

T is a special global variable. It is protected from being modified by SET or SETQ.

91.5 The Extended Syntax

Whenever it is possible to define Standard LISP functions in LISP the text of the function will appear in an extended syntax. These definitions are supplied as an aid to understanding the behavior of functions and not as a strict implementation guide. A formal scheme for the translation of extended syntax to Standard LISP is presented to eliminate misinterpretation of the definitions.

91.5.1 Definition

The goal of the transformation scheme is to produce a PUTD invocation which has the function translated from the extended syntax as its actual parameter. A rule has a name in brackets <...> by which it is known and is defined by what follows the meta symbol ::=. Each rule of the set consists of one or more "alternatives" separated by the | meta symbol, being the different ways in which the rule will be matched by source text. Each alternative is composed of a "recognizer" and a "generator" separated by the $\Longrightarrow$ meta symbol. The recognizer is a concatenation of any of three different forms. 1) Terminals - Upper case lexemes and punctuation which is not part of the meta syntax represent items which must appear as is in the source text for the rule to succeed. 2) Rules - Lower case lexemes enclosed in <...> are names of other rules. The source text is matched if the named rule succeeds. 3) Primitives - Lower case singletons not in brackets are names of primitives or primitive classes of Standard LISP. The syntax and semantics of the primitives are given in Part I.

The recognizer portion of the following rule matches an extended syntax procedure:

<*function*> ::= ftype PROCEDURE id (<*id list*>);
<*statement*>; $\Longrightarrow$

A function is recognized as an "ftype" (one of the tokens EXPR, FEXPR, etc.) followed by the keyword PROCEDURE, followed by an "id" (the name of the function), followed by an <*id list*> (the formal parameter names) enclosed in parentheses. A semicolon terminates the title line. The body of the function is a <*statement*> followed by a semicolon. For example:

```
EXPR PROCEDURE NULL(X); EQ(X, NIL);
```

satisfies the recognizer, causes the generator to be activated and the rule to be matched successfully.

The generator is a template into which generated items are substituted. The three syntactic entities have corresponding meanings when they appear in the generator portion. 1) Terminals - These lexemes are copied as is to the generated text. 2) Rules - If a rule has succeeded in the recognizer section then the value of the rule is the result of the generator portion of that rule. 3) Primitives - When primitives are matched the primitive lexeme replaces its occurrence in the generator.

If more than one occurrence of an item would cause ambiguity in the generator portion this entity appears with a bracketed subscript. Thus:

<*conditional*> ::=
IF <*expression*> THEN <*statement*$_1$>
ELSE <*statement*$_2$> ...

has occurrences of two different <*statement*>s. The generator portion uses the subscripted entities to reference the proper generated value.

The <*function*> rule appears in its entirety as:

<*function*> ::= ftype PROCEDURE id (<*id list*>);<*statement*>; $\Longrightarrow$
(PUTD (QUOTE id)
(QUOTE ftype)
(QUOTE (LAMBDA (<*id list*>) <*statement*>)))

If the recognizer succeeds (as it would in the case of the NULL procedure example) the generator returns:

```
(PUTD (QUOTE NULL) (QUOTE EXPR) (QUOTE (LAMBDA (X) (EQ X NIL))))
```

The identifier in the template is replaced by the procedure name NULL, <*id list*> by the single formal parameter X, the <*statement*> by (EQ X NIL) which is the result of the <*statement*> generator. EXPR replaces ftype, the type of the defined procedure.

91.5.2 The Extended Syntax Rules

<*function*> ::= ftype **PROCEDURE** id (<*id list*>); <*statement*>; $\Longrightarrow$
(PUTD (QUOTE id)
(QUOTE ftype)
(QUOTE (LAMBDA (<*id list*>) <*statement*>)))

<*id list*> ::= id $\Longrightarrow$ id |
id, <*id list*> $\Longrightarrow$ id <*id list*> |
$\Longrightarrow$ NIL
<*statement*> ::= <*expression*> $\Longrightarrow$ <*expression*> |
<*proper statement*> $\Longrightarrow$ <*proper statement*>

<*proper statement*> ::=
<*assignment statement*> $\Longrightarrow$ <*assignment statement*> |
<*conditional statement*> $\Longrightarrow$ <*conditional statement*> |
<*while statement*> $\Longrightarrow$ <*while statement*> |
<*compound statement*> $\Longrightarrow$ <*compound statement*>

<*assignment statement*> ::= id := <*expression*> $\Longrightarrow$
(SETQ id <*expression*>)

<*conditional statement*> ::=
IF <*expression*> **THEN** <*statement*$_1$> **ELSE** <*statement*$_2$> $\Longrightarrow$
(COND (<*expression*> <*statement*$_1$>)(T <*statement*$_2$>)) |
IF <*expression*> **THEN** <*statement*> $\Longrightarrow$
(COND (<*expression*> <*statement*>))

<*while statement*> ::= **WHILE** <*expression*> **DO** <*statement*> $\Longrightarrow$
(PROG NIL
LBL (COND ((NULL <*expression*>) (RETURN NIL)))
<*statement*>
(GO LBL))

<*compound statement*> ::=
BEGIN SCALAR <*id list*>; <*program list*> **END** $\Longrightarrow$
(PROG (<*id list*>) <*program list*>) |
BEGIN <*program list*> **END** $\Longrightarrow$
(PROG NIL <*program list*>) |
<< <*statement list*> >> $\Longrightarrow$ (PROGN <*statement list*>)

<*program list*> ::= <*full statement*> $\Longrightarrow$ <*full statement*> |
<*full statement*> <*program list*> $\Longrightarrow$
<*full statement*> <*program list*>

<*full statement*> ::= <*statement*> $\Longrightarrow$ <*statement*> | id: $\Longrightarrow$ id

<*statement list*> ::= <*statement*> $\Longrightarrow$ <*statement*> |
<*statement*>; <*statement list*> $\Longrightarrow$
<*statement*> <*statement list*>

<*expression*> ::=
<*expression*$_1$> **.** <*expression*$_2$> $\Longrightarrow$
(CONS <*expression*$_1$> <*expression*$_2$> |
<*expression*$_1$> **=** <*expression*$_2$> $\Longrightarrow$
(EQUAL <*expression*$_1$> <*expression*$_2$>) |
<*expression*$_1$> **EQ** <*expression*$_2$> $\Longrightarrow$
(EQ <*expression*$_1$> <*expression*$_2$>) |
'<*expression*> $\Longrightarrow$ (QUOTE <*expression*>) |
function <*expression*> $\Longrightarrow$ (function <*expression*>) |
function(<*argument list*>) $\Longrightarrow$ (function <*argument list*>) |
number $\Longrightarrow$ number |
id $\Longrightarrow$ id

<*argument list*> ::= () $\Longrightarrow$ |
<*expression*> $\Longrightarrow$ <*expression*> |
<*expression*>, <*argument list*> $\Longrightarrow$ <*expression*> <*argument list*>

Notice the three infix operators . EQ and = which are translated into calls on CONS, EQ, and EQUAL respectively. Note also that a call on a function which has no formal parameters must have () as an argument list. The QUOTE function is abbreviated by '.

Part IV

Appendix

Appendix A

Reserved Identifiers

We list here all identifiers that are normally reserved in REDUCE including names of commands, operators and switches initially in the system. Excluded are words that are reserved in specific implementations of the system.

Commands	ALGEBRAIC ANTISYMMETRIC ARRAY BYE CLEAR CLEARRULES COMMENT CONT DECOMPOSE DEFINE DEPEND DISPLAY ED EDITDEF END EVEN FACTOR FOR FORALL FOREACH GO GOTO IF IN INDEX INFIX INPUT INTEGER KORDER LET LINEAR LISP LISTARGP LOAD LOAD_PACKAGE MASS MATCH MATRIX MSHELL NODEPEND NONCOM NONZERO NOSPUR ODD OFF ON OPERATOR ORDER OUT PAUSE PRECEDENCE PRINT_PRECISION PROCEDURE QUIT REAL REMFAC REMIND RETRY RETURN SAVEAS SCALAR SETMOD SHARE SHOWTIME SHUT SPUR SYMBOLIC SYMMETRIC VECDIM VECTOR WEIGHT WRITE WTLEVEL
Boolean Operators	EVENP FIXP FREEOF NUMBERP ORDP PRIMEP
Infix Operators	:= = >= > <= < => + * / ^ ** . WHERE SETQ OR AND MEMBER MEMQ EQUAL NEQ EQ GEQ GREATERP LEQ LESSP PLUS DIFFERENCE MINUS TIMES QUOTIENT EXPT CONS

Numerical Operators	ABS ACOS ACOSH ACOT ACOTH ACSC ACSCH ASEC ASECH ASIN ASINH ATAN ATANH ATAN2 COS COSH COT COTH CSC CSCH EXP FACTORIAL FIX FLOOR HYPOT LN LOG LOGB LOG10 NEXTPRIME ROUND SEC SECH SIN SINH SQRT TAN TANH
Prefix Operators	APPEND ARGLENGTH CEILING COEFF COEFFN COFACTOR CONJ DEG DEN DET DF DILOG EI EPS ERF FACTORIZE FIRST GCD G IMPART INT INTERPOL LCM LCOF LENGTH LHS LINELENGTH LTERM MAINVAR MAT MATEIGEN MAX MIN MKID NULLSPACE NUM PART PF PRECISION RANDOM RANDOM_NEW_SEED RANK REDERR REDUCT REMAINDER REPART REST RESULTANT REVERSE RHS SECOND SET SHOWRULES SIGN SOLVE STRUCTR SUB SUM THIRD TP TRACE VARNAME
Reserved Variables	CARD_NO E EVAL_MODE FORT_WIDTH HIGH_POW I INFINITY K!* LOW_POW NIL PI ROOT_MULTIPLICITY T
Switches	ADJPREC ALGINT ALLBRANCH ALLFAC BFSPACE COMBINEEXPT COMBINELOGS COMP COMPLEX CRAMER CREF DEFN DEMO DIV ECHO ERRCONT EVALLHSEQP EXP EXPANDLOGS EZGCD FACTOR FORT FULLROOTS GCD IFACTOR INT INTSTR LCM LIST LISTARGS MCD MODULAR MSG MULTIPLICITIES NAT NERO NOSPLIT OUTPUT PERIOD PRECISE PRET PRI RAT RATARG RATIONAL RATIONALIZE RATPRI REVPRI RLISP88 ROUNDALL ROUNDBF ROUNDED SAVESTRUCTR SOLVESINGULAR TIME TRA TRFAC TRIGFORM TRINT
Other Reserved Ids	BEGIN DO EXPR FEXPR INPUT LAMBDA LISP MACRO PRODUCT REPEAT SMACRO SUM UNTIL WHEN WHILE WS

Bibliography

[1] George A. Baker(Jr.) and Peter Graves-Morris. *Padé Approximants, Part I: Basic Theory.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

[2] Computation Center. *LISP Reference Manual, CDC-6000.* The University of Texas at Austin.

[3] S.-C. Chou. Proving elementary geometry theorems using Wu's algorithm. In *Contemp. Math.*, volume 19, pages 243 – 286. AMS, Providence, Rhode Island, 1984.

[4] S.-C. Chou. *Mechanical geometry theorem proving.* Reidel, Dortrecht, 1988.

[5] S.-C. Chou. Automated reasoning in geometries using the characteristic set method and Gröbner basis method. In *Proc. ISSAC-90*, pages 255–260. ACM Press, 1990.

[6] Andreas Dolzmann and Thomas Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, June 1997.

[7] Andreas Dolzmann and Thomas Sturm. Simplification of quantifier-free formulae over ordered fields. *Journal of Symbolic Computation*, 24(2):209–231, August 1997.

[8] Andreas Dolzmann and Thomas Sturm. *Redlog User Manual.* FMI, Universität Passau, D-94030 Passau, Germany, April 1999. Edition 2.0 for Version 2.0.

[9] Stanford Center for Information Processing. *LISP/360 Reference Manual.* Stanford University.

[10] M. L. Griss and A. C. Hearn. A portable LISP compiler. *Software—Practice and Experience*, 11:541–605, June 1981.

[11] A. C. Hearn. REDUCE user's manual: Version 3.3. Publication CP78 (Rev 1/88), RAND, 1988.

[12] A. C. Hearn, P. K. Kuo, and D. R. Yennie. Radiative corrections to an electron-positron scattering experiment. *Phys. Rev.*, 187:2088–2096, 1969.

[13] Wolfram Koepf. REDUCE package for the indefinite and definite summation. *SIGSAM Bulletin*, 29(1):14–30, January 1995.

[14] T. H. Koornwinder. On Zeilberger's algorithm and its q-analogue: a rigorous description. *J. of Comput. and Appl. Math.*, 48:91–111, 1993.

[15] *MACLISP Reference Manual*, March 1976.

[16] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmers Manual*. The M.I.T. Press, Cambridge, Massachusettes, 1965.

[17] Mats Nordstrom, Erik Sandewall, and Diz Breslow. *LISP F1: A FORTRAN Implementation of LISP 1.5*. Uppsala University, Department of Computer Sciences.

[18] Lynn H. Quam and Whitfield Diffie. *Stanford LISP 1.6 Manual*. Stanford Artificial Intelligence Laboratory, operating note 28.7 edition.

[19] Warren Teitelman. *INTERLISP Reference Manual*. XEROX, Palo Alto Research Centers, 3333 Coyote Road, Palo Alto, California 94304, 1978.

[20] Volker Weispfenning. Comprehensive Gröbner bases. *Journal of Symbolic Computation*, 14:1–29, July 1992.

[21] W.-T. Wu. On the decision problem and the mechanization of theorem-proving in elementary geometry. In *Contemp. Math.*, volume 19, pages 213 – 234. AMS, Providence, Rhode Island, 1984.

[22] W.-T. Wu. Some recent advances in mechanical theorem proving of geometry. In *Contemp. Math.*, volume 19, pages 235 – 241. AMS, Providence, Rhode Island, 1984.

[23] W.-T. Wu. *Mechanical Theorem Proving in Geometries*. Number 1 in Texts and Monographs in Symbolic Computation. Springer, Wien, 1994.

Index