Computer Algebra

Reference Manual

REDUCE

Version July 2010

LaTeX'ed by $\boxed{\mathcal{SCIOS}}$

REDUCE is an interactive system for general algebraic computations of interest to mathematicians, scientists and engineers. It has been produced by a collaborative effort involving many contributors.

REDUCE traces its origins to work begun by Anthony Hearn in 1963 and continued ever since. The first distribution occurred in 1968. Since that time, over a hundred people have been involved in various ways in its development.

A small number of these people have made sustained contributions to the REDUCE core and associated packages over many years, namely John Fitch, Herbert Melenk, Winfried Neun, Arthur Norman and Eberhard Schrüfer.

Others who have contributed to either documentation or packages for REDUCE include John Abbott, Paul Abbott, Victor Adamchik, Werner Antweiler, Alan Barnes, Andreas Bernig, Yuri A. Blinkov, Harald Boeing, W.N. Borst, F. Brackx, Russell Bradford, Andreas Brand, Fran Burstall, Chris Cannam, Hubert Caprasse, D. Constales, Caroline Cotter, James Davenport, Michael Dewar, C. Dicrescenzo, Andreas Dolzmann, Alain Dresse, Ladislav Drska, James Eastwood, Bruce A. Florman, Kerry Gaskell, Karin Gatermann, Barbara L. Gates, R. Gebauer, Vladimir Gerdt, John Gottschalk, Hans-Gert Graebe, Martin Griss, A.G. Grozin, David Harper, John Harper, Steve Harrington, David Hartley, M.C. van Heerwaarden, Friedrich Hehl, Daniel Hobbs, Joachim Hollman, B.J.A. Hulshof, J.A. van Hulzen, V. Ilyin, N. Ito, F. Kako, Stan Kameny, C. Kazasov, Nancy Kirkwood, K. Kishimoto, Wolfram Koepf, H. Kredel, A.P. Kryukov, Neil Langmead, A. Lasaruk, D. Lewien, Richard Liska, Ruediger Loos, Malcolm MacCallum, Norman MacDonald, Jed Marti, Kevin McIsaac, H. Meyer, H. Michael Moeller, Mary Ann Moore, Shuichi Moritsugu, Donald Morrison, Alain Moussiaux, C.J. Neerdaels, K. Onaga, Julian Padget, Gerhard Rayna, Matt Rebbeck, Francoise Richard, F. Richard-Jung, A.J. Roberts, A.Ya. Rodionov, T. Sasaki, Carsten Schoebel, Franziska Schoebel, Rainer Schoepf, Fritz Schwarz, Andreas Seidl, James Sherring, Luis Alvarez Sobreviela, D. Stauffer, Gregor Stoelting, David R. Stoutemyer, Stephen Scowcroft, Yves Siret, M. Spiridonova, H. Steeb, Andreas Strotmann, Thomas Sturm, Takeyuki Takahashi, A. Taranov, Lisa Temme, Walter Tietze, V. Tomov, Evelyne Tournier, Arrigo Triulzi, R.W. Tucker, Philip A Tuckey, Gokturk Ucoluk, J. Ueberberg, J.B. van Veelen, Mathias Warns, Volker Winkelmann, Thomas Wolf, Francis Wright, K. Yamamoto, J.G. Zabolitzky, Alexey Yu. Zharkov and Vadim V. Zhytnikov.

and many others . . .

Contents

1 Preface

This manual describes the REDUCE symbolic mathematics system. REDUCE has two modes of operation: the algebraic mode, which deals with polynomials and mathematical functions in a simple procedural syntax, and the symbolic mode, which allows Lisp-like syntax and operations. The commands, declarations, switches and operators available in algebraic-mode REDUCE are arranged in this manual in alphabetical order. Symbols are listed before the letter A.

Following the general alphabetical reference section is a similar reference section for the High-Energy Physics operators. After that, you can find several cross-reference sections. The first section contains lists of reserved words and an Instant Function Cross-Reference. Next you will find brief explanations of the common REDUCE error messages. The next section is organized by type into Commands, Declarations, Operators, Switches and Variables, with a brief listing for each operation.

For a general introduction to using algebraic-mode REDUCE, see the *REDUCE User's Guide*, which also contains information on symbolic mode. The *The Standard Lisp Report* is a technical reference on REDUCE's Lisp language.

The following symbols are used to describe syntax in this manual:

`This font means you must type an item exactly as you see it.`

This font indicates a descriptive name for a type of REDUCE expression. You may choose any REDUCE expression of the appropriate type.

{} Braces surround an item or set of items that may be followed by an asterisk or plus. Do not type the braces.

*** An italic asterisk indicates that the preceding item may be repeated zero or more times. Do not type the asterisk. It does not indicate multiplication.

+ An italic plus indicates that the preceding item must appear once, and may be repeated one or more times. Do not type the plus. It does not indicate addition.

&option(...) *&option* indicates that the parameters that follow are optional. *&options* indicates that options are available and explained in the text below the command line. *&option(s)* is not to be typed.

The switch settings for REDUCE in the examples in this manual are assumed to be the default settings, unless specifically given otherwise. See the cross-reference section *Switches* in the back of this volume.

The examples in this manual should exactly reproduce the results you get by typing in the statements given. Any non-default switch settings are shown. Be sure that the variables or operators used have no prior definition by using the `clear` command. The numbered line prompts have generally been left out. You can find executable files of all the examples shown here in your `$reduce/refex` directory, named alphabetically. If you are working your way through this manual, you can run the examples as you go by starting a new REDUCE session, and entering the command, for example:

```
in "reduce/refex/a-ex";
```

There are numerous pauses in the files so that you can enter your own examples and commands. If you change any switch settings or assign values to variables in one of the pauses, make sure to restore everything to its original state before you continue the file (see the entry under `CLEAR` if you need help in clearing variables and operators).

REDUCE converts all input to upper case, and all its responses are in upper case. You can type input in upper case, lower case, or mixed, as you wish. In the examples, the input is lower case, and REDUCE's responses are shown in upper case. This protocol makes it easy to distinguish input from results. You can tell whether you are in algebraic or symbolic mode by looking at the numbered prompt statement REDUCE gives you: the algebraic prompt contains a colon (`:`), while the symbolic prompt contains an asterisk (`*`).

2 Concepts

2.1 IDENTIFIER

IDENTIFIER Type

Identifiers in REDUCE consist of one or more alphanumeric characters, of which the first must be alphabetical. The maximum number of characters allowed is system dependent, but is usually over 100. However, printing is simplified if they are kept under 25 characters.

You can also use special characters in your identifiers, but each must be preceded by an exclamation point ! as an escape character. Useful special characters are # $ % ^ & * - + = ? < > ~ | / ! and the space. Note that the use of the exclamation point as a special character requires a second exclamation point as an escape character. The underscore _ is special in this regard. It must be preceded by an escape character in the first position in an identifier, but is treated like a normal letter within an identifier.

Other characters, such as ( ) # ; ‘ ’ " can also be used if preceded by a !, but as they have special meanings to the Lisp reader it is best to avoid them to avoid confusion.

Many system identifiers have * before or after their names, or - between words. If you accidentally pick one of these names for your own identifier, it could have disastrous effects. For this reason it is wise not to include * or - anywhere in your identifiers.

You will notice that REDUCE does not use the escape characters when it prints identifiers containing special characters; however, you still must use them when you refer to these identifiers. Be careful when editing statements containing escaped special characters to treat the character and its escape as an inseparable pair.

Identifiers are used for variable names, labels for `go to` statements, and names of arrays, matrices, operators, and procedures. Once an identifier is used as a matrix, array, scalar or operator identifier, it may not be used again as a matrix, array or operator. An operator or array identifier may later be used as a scalar without problems, but a matrix identifier cannot be used as a scalar. All procedures are entered into the system as operators, so the name of a procedure may not be used as a matrix, array, or operator identifier either.

2.2 KERNEL

KERNEL Type

A `kernel` is a form that cannot be modified further by the REDUCE canonical simplifier. Scalar variables are always kernels. The other important class of kernels are operators with their arguments. Some examples should help clarify this concept:

Expression	Kernel?
`x`	Yes
varname	Yes
`cos(a)`	Yes
`log(sin(x**2))`	Yes
`a*b`	No
`(x+y)**4`	No
matrix identifier	No

Many REDUCE operators expect kernels among their arguments. Error messages result from attempts to use non-kernel expressions for these arguments.

2.3 STRING

STRING — Type

A `string` is any collection of characters enclosed in double quotation marks ("). It may be used as an argument for a variety of commands and operators, such as `in`, `rederr` and `write`.

Examples

```
write "this is a string";      ⇒   this is a string

write a, " ", b, " ",c,"!";    ⇒   A B C!
```

3 Variables

3.1 assumptions

ASSUMPTIONS — Variable

After solving a linear or polynomial equation system with parameters, the variable **`assumptions`** contains a list of side relations for the parameters. The solution is valid only as long as none of these expression is zero.

Examples

```
solve({a*x-b*y+x,y-c},{x,y});
                                      b*c
                        ⇒     {{x=-------,y=c}}
                                     a + 1
assumptions;            ⇒     {a + 1}
```

3.2 CARD_NO

CARD_NO Variable

card_no sets the total number of cards allowed in a Fortran output statement when fort is on. Default is 20.

Examples

```
on fort;

card_no := 4;       ⇒   CARD_NO=4.

z := (x + y)**15;   ⇒

             ANS1=5005.*X**6*Y**9+3003.*X**5*Y**10+1365.*X**4*Y**
            . 11+455.*X**3*Y**12+105.*X**2*Y**13+15.*X*Y**14+Y**15
             Z=X**15+15.*X**14*Y+105.*X**13*Y**2+455.*X**12*Y**3+
            . 1365.*X**11*Y**4+3003.*X**10*Y**5+5005.*X**9*Y**6+
            . 6435.*X**8*Y**7+6435.*X**7*Y**8+ANS1
```

Comments

Twenty total cards means 19 continuation cards. You may set it for more if your Fortran system allows more. Expressions are broken apart in a Fortran-compatible way if they extend for more than card_no continuation cards.

3.3 E

E Constant

The constant `e` is reserved for use as the base of the natural logarithm. Its value is approximately 2.71828284590, which REDUCE gives to the current decimal precision when the switch `rounded` is on.

Comments

`e` may be used as an iterative variable in a `for` statement, or as a local variable or a `procedure`. If `e` is defined as a local variable inside the procedure, the normal definition as the base of the natural logarithm would be suspended inside the procedure.

3.4 EVAL_MODE

EVAL_MODE Variable

The system variable `eval_mode` contains the current mode, either `algebraic` or `symbolic`.

Examples

`EVAL_MODE;` $\Rightarrow$ `ALGEBRAIC`

Comments

Some commands do not behave the same way in algebraic and symbolic modes.

3.5 FORT_WIDTH

FORT_WIDTH Variable

The `fort_width` variable sets the number of characters in a line of Fortran-compatible output produced when the `fort` switch is on. Default is 70.

Examples

```
fort_width := 30;   ⇒   FORT_WIDTH := 30

on fort;

df(sin(x**3*y),x);  ⇒         ANS=3.*COS(X
                            . **3*Y)*X**2*
                            . Y
```

Comments

`fort_width` includes the usually blank characters at the beginning of the card. As you may notice above, it is conservative and makes the lines even shorter than it was told.

3.6 HIGH_POW

HIGH_POW — Variable

The variable `high_pow` is set by `coeff` to the highest power of the variable of interest in the given expression. You can access this variable for use in further computation or display.

Examples

```
coeff((x+1)^5*(x*(y+3)^2)^2,x);

                         ⇒

       {0,
        0,

         4        3        2
        Y  + 12*Y  + 54*Y  + 108*Y + 81,

            4        3        2
        5*(Y  + 12*Y  + 54*Y  + 108*Y + 81),

             4        3        2
        10*(Y  + 12*Y  + 54*Y  + 108*Y + 81),

             4        3        2
        10*(Y  + 12*Y  + 54*Y  + 108*Y + 81),

            4        3        2
        5*(Y  + 12*Y  + 54*Y  + 108*Y + 81),

         4        3        2
        Y  + 12*Y  + 54*Y  + 108*Y + 81}
high_pow;                ⇒   7
```

3.7 I

I Constant

REDUCE knows `i` is the square root of -1, and that $i^2 = -1$.

Examples

`(a + b*i)*(c + d*i);`	⇒	`A*C + A*D*I + B*C*I - B*D`
`i**2;`	⇒	`-1`

Comments

`i` cannot be used as an identifier. It is all right to use `i` as an index variable in a `for` loop, or as a local (`scalar`) variable inside a `begin...end` block, but it loses its definition as the square root of -1 inside the block in that case.

Only the simplest properties of i are known by REDUCE unless the switch `complex` is turned on, which implements full complex arithmetic in factoring, simplification, and functional values. `complex` is ordinarily off.

3.8 INFINITY

INFINITY — Constant

The name `infinity` is used to represent the infinite positive number. However, at the present time, arithmetic in terms of this operator reflects finite arithmetic, rather than true operations on infinity.

3.9 LOW_POW

LOW_POW Variable

The variable `low_pow` is set by `coeff` to the lowest power of the variable of interest in the given expression. You can access this variable for use in further computation or display.

Examples

```
                              6
coeff((x+2*y)**6,y);     ⇒   {X ,

                                 5
                              12*X ,

                                 4
                              60*X ,

                                  3
                              160*X ,

                                  2
                              240*X ,
                              192*X,
                              64}
low_pow;                 ⇒   0

coeff(x**2*(x*sin(y) + 1),x);

                         ⇒   {0,0,1,SIN(Y)}

low_pow;                 ⇒   2
```

3.10 NIL

NIL — Constant

`nil` represents the truth value *false* in symbolic mode, and is a synonym for 0 in algebraic mode. It cannot be used for any other purpose, even inside procedures or `for` loops.

3.11 PI

PI Constant

The identifier `pi` is reserved for use as the circular constant. Its value is given by 3.14159265358..., which REDUCE gives to the current decimal precision when REDUCE is in a floating-point mode.

Comments

`pi` may be used as a looping variable in a `for` statement, or as a local variable in a `procedure`. Its value in such cases will be taken from the local environment.

3.12 requirements

REQUIREMENTS — Variable

After an attempt to solve an inconsistent equation system with parameters, the variable `requirements` contains a list of expressions. These expressions define a set of conditions implicitly equated with zero. Any solution to this system defines a setting for the parameters sufficient to make the original system consistent.

Examples

```
solve({x-a,x-y,y-1},{x,y});  ⇒  {}

requirements;                ⇒  {a - 1}
```

3.13 ROOT_MULTIPLICITIES

ROOT_MULTIPLICITIES Variable

The `root_multiplicities` variable is set to the list of the multiplicities of the roots of an equation by the `solve` operator.

Comments

`solve` returns its solutions in a list. The multiplicities of each solution are put in the corresponding locations of the list `root_multiplicities`.

3.14 T

T Constant

The constant `t` stands for the truth value *true*. It cannot be used as a scalar variable in a `block`, as a looping variable in a `for` statement or as an `operator` name.

4 Syntax

4.1 semicolon

; **Command**

The semicolon is a statement delimiter, indicating results are to be printed when used in interactive mode.

Examples

```
                         2
(x+1)**2;          ⇒    X  + 2*X + 1

df(x**2 + 1,x);    ⇒    2*X
```

Comments

Entering a Return without a semicolon or dollar sign results in a prompt on the following line. A semicolon or dollar sign can be added at this point to execute the statement. In interactive mode, a statement that is ended with a semicolon and Return has its results printed on the screen.

Inside a group statement `<<...>>` or a `begin...end` block, a semicolon or dollar sign separates individual REDUCE statements. Since results are not printed from a block without a specific `return` statement, there is no difference between using the semicolon or dollar sign. In a group statement, the last value produced is the value returned by the group statement. Thus, if a semicolon or dollar sign is placed between the last statement and the ending brackets, the group statement returns the value 0 or *nil*, rather than the value of the last statement.

4.2 dollar

$ Command

The dollar sign is a statement delimiter, indicating results are not to be printed when used in interactive mode.

Examples

```
(x+1)**2$  ⇒

        The workspace is set to x^2 + 2x + 1
        but nothing shows on the screen
                      2
ws;        ⇒    X  + 2*X + 1
```

Comments

Entering a Return without a semicolon or dollar sign results in a prompt on the following line. A semicolon or dollar sign can be added at this point to execute the statement. In interactive mode, a statement that ends with a dollar sign `$` and a Return is executed, but the results not printed.

Inside a `group` statement `<<`...`>>` or a `begin`...`end` `block`, a semicolon or dollar sign separates individual REDUCE statements. Since results are not printed from a `block` without a specific `return` statement, there is no difference between using the semicolon or dollar sign.

In a group statement, the last value produced is the value returned by the group statement. Thus, if a semicolon or dollar sign is placed between the last statement and the ending brackets, the group statement returns the value 0 or *nil*, rather than the value of the last statement.

4.3 percent

% Command

The percent sign is used to precede comments; everything from a percent to the end of the line is ignored.

Examples

```
df(x**3 + y,x);% This is a comment  Return
```

```
                 2
        ⇒   3*X
```

```
int(3*x**2,x) %This is a comment;  Return
```

A prompt is given, waiting for the semicolon that was not detected in the comment

Comments

Statement delimiters `;` and `$` are not detected between a percent sign and the end of the line.

4.4 dot

. **Operator**

The . (dot) infix binary operator adds a new item to the beginning of an existing `list`. In high energy physics expressions, it can also be used to represent the scalar product of two Lorentz four-vectors.

item . *list*

item can be any REDUCE scalar expression, including a list; *list* must be a `list` to avoid producing an error message. The dot operator is right associative.

Examples

```
liss := a . {};             ⇒   LISS := {A}

liss := b . liss;           ⇒   LISS := {B,A}

newliss := liss . liss;     ⇒   NEWLISS := {{B,A},B,A}

firstlis := a . b . {c};    ⇒   FIRSTLIS := {A,B,C}

secondlis := x . y . {z};   ⇒   SECONDLIS := {X,Y,Z}

for i := 1:3 sum part(firstlis,i)*part(secondlis,i);

                            ⇒   A*X + B*Y + C*Z
```

4.5 assign

:= Operator

The `:=` is the assignment operator, assigning the value on the right-hand side to the identifier or other valid expression on the left-hand side.

restricted_expression `:=` *expression*

restricted_expression is ordinarily a single identifier, though simple expressions may be used (see Comments below). *expression* is any valid REDUCE expression. If *expression* is a `matrix` identifier, then *restricted_expression* can be a matrix identifier (redimensioned if necessary) which has each element set to the corresponding elements of the identifier on the right-hand side.

Examples

```
                                        2
a := x**2 + 1;                 ⇒   A := X  + 1
                                    2
a;                             ⇒   X  + 1

first := second := third;      ⇒   FIRST := SECOND := THIRD

first;                         ⇒   THIRD

second;                        ⇒   THIRD

b := for i := 1:5 product i;

                               ⇒   B := 120

b;                             ⇒   120

w + (c := x + 3) + z;          ⇒   W + X + Z + 3

c;                             ⇒   X + 3

y + b := c;                    ⇒   Y + B := C

y;                             ⇒   - (B - C)
```

Comments

The assignment operator is right associative, as shown in the second and third examples. A string of such assignments has all but the last item set to the value of the last item. Embedding an assignment statement in another expression has the side effect of making the assignment, as well as causing the given replacement in the expression.

Assignments of values to expressions rather than simple identifiers (such as in the last example above) can also be done, subject to the following remarks:

(i) If the left-hand side is an identifier, an operator, or a power, the substitution rule is added to the rule table.

(ii) If the operators `- + /` appear on the left-hand side, all but the first term of the expression is moved to the right-hand side.

(iii) If the operator `*` appears on the left-hand side, any constant terms are moved to the right-hand side, but the symbolic factors remain.

Assignment is valid for `array` elements, but not for entire arrays. The assignment operator can also be used to attach functionality to operators.

A recursive construction such as `a := a + b` is allowed, but when `a` is referenced again, the process of resubstitution continues until the expression stack overflows (you get an error message). Recursive assignments can be done safely inside controlled loop expressions, such as `for`... or `repeat`...`until`.

4.6 equalsign

= **Operator**

The = operator is a prefix or infix equality comparison operator.

=(*expression* , *expression*) or *expression* = *expression*

expression can be any REDUCE scalar expression.

Examples

```
a := 4;                   ⇒   A := 4

if =(a,10) then write "yes" else write "no";

                          ⇒   no

b := c;                   ⇒   B := C

if b = c then write "yes" else write "no";

                          ⇒   yes

on rounded;

if 4.0 = 4 then write "yes" else write "no";

                          ⇒   yes
```

Comments

This logical equality operator can only be used inside a conditional statement, such as `if`...`then`...`else` or `repeat`...`until`. In other places the equal sign establishes an algebraic object of type `equation`.

4.7 replace

REPLACE Operator

The following sign is used: =>

The `=>` operator is a binary operator used in `rule` lists to denote replacements.

Examples

```
operator f;

let f(x) => x^2;

f(x);
```

$\Rightarrow$ x^2

4.8 plussign

$+$ **Operator**

The + operator is a prefix or infix n-ary addition operator.

expression { +*expression*}+

or +(*expression*{, *expression*}+)

expression may be any valid REDUCE expression.

Examples

```
x**4 + 4*x**2 + 17*x + 1;  ⇒   X^4 + 4*X^2 + 17*X + 1

14 + 15 + x;               ⇒   X + 29

+(1,2,3,4,5);              ⇒   15
```

Comments

+ is also valid as an addition operator for `matrix` variables that are of the same dimensions and for `equation`s.

4.9 minussign

- Operator

The `-` operator is a prefix or infix binary subtraction operator, as well as the unary minus operator.

expression `-` *expression* or `-`(*expression*, *expression*)

expression may be any valid REDUCE expression.

Examples

`15 - 4;`	⇒	`11`
`x*(-5);`	⇒	`- 5*X`
`a - b - 15;`	⇒	`A - B - 15`
`-(a,4);`	⇒	`A - 4`

Comments

The subtraction operator is left associative, so that a - b - c is equivalent to (a - b) - c, as shown in the third example. The subtraction operator is also valid with `matrix` expressions of the correct dimensions and with `equation`s.

4.10 asterisk

***** **Operator**

The * operator is a prefix or infix n-ary multiplication operator.

expression { * *expression*}+

or *(*expression*{, *expression*}+)

expression may be any valid REDUCE expression.

Examples

```
15*3;               =>   45

24*x*yvalue*2;      =>   48*X*YVALUE

*(6,x);             =>   6*X

on rounded;
                            3
3*1.5*x*x*x;        =>   4.5*X

off rounded;
                          2
2x**2;              =>   2*X
```

Comments

REDUCE assumes you are using an implicit multiplication operator when an identifier is preceded by a number, as shown in the last line above. Since no valid identifiers can begin with numbers, there is no ambiguity in making this assumption.

The multiplication operator is also valid with `matrix` expressions of the proper dimensions: matrices A and B can be multiplied if A is $n \times m$ and B is $m \times p$. Matrices and `equation`s can also be multiplied by scalars: the result is as if each element was multiplied by the scalar.

4.11 slash

/   Operator

The / operator is a prefix or infix binary division operator or prefix unary `recip`rocal operator.

expression/*expression* *or* /*expression*

or /(*expression*, *expression*)

expression may be any valid REDUCE expression.

Examples

```
20/5;        ⇒   4

                 50
100/6;       ⇒   ----
                 3

                 8
16/2/x;      ⇒   ---
                 X

                 1
/b;          ⇒   ---
                 B

                 Y
/(y,5);      ⇒   ---
                 5

on rounded;

35/4;        ⇒   8.75

/20;         ⇒   0.05
```

Comments

The division operator is left associative, so that `a/b/c` is equivalent to `(a/b)/c`. The division operator is also valid with square `matrix` expressions of the same dimensions: With A and B both $n \times n$ matrices and B invertible, A/B is given by $A \times B^{-1}$. Division of a matrix by a scalar is defined, with the results being the division of each element of the matrix by the scalar. Division of a scalar by a matrix is defined if the matrix is invertible, and has the effect of multiplying the scalar by the inverse of the matrix. When / is used as a reciprocal operator for a matrix, the inverse of the matrix is returned if it exists.

4.12 power

**

Operator

The ** operator is a prefix or infix binary exponentiation operator.

expression ***expression* or **(*expression*, *expression*)

expression may be any valid REDUCE expression.

Examples

```
                 15
x**15;       ⇒  X
                 Y*Z
x**y**z;     ⇒  X

                  Z
                 Y
x**(y**z);   ⇒  X
                 4
**(y,4);     ⇒  Y

on rounded;

2**pi;       ⇒  8.82497782708
```

Comments

The exponentiation operator is left associative, so that `a**b**c` is equivalent to `(a**b)**c`, as shown in the second example. Note that this is *not* `a**(b**c)`, which would be right associative.

When `nat` is on (the default), REDUCE output produces raised exponents, as shown. The symbol `^`, which is the upper-case 6 on most keyboards, may be used in the place of `**`.

A square `matrix` may also be raised to positive and negative powers with the exponentiation operator (negative powers require the matrix to be invertible). Scalar expressions and `equation`s may be raised to fractional and floating-point powers.

4.13 caret

---

^ **Operator**

The ^ operator is a prefix or infix binary exponentiation operator. It is equivalent to `power` or **.

expression ^ *expression* or ^(*expression*, *expression*)

expression may be any valid REDUCE expression.

Examples

```
               15
x^15;      ⇒  X

               Y*Z
x^y^z;     ⇒  X

                Z
               Y
x^(y^z);   ⇒  X

               4
^(y,4);    ⇒  Y

on rounded;

2^pi;      ⇒  8.82497782708
```

Comments

The exponentiation operator is left associative, so that `a^b^c` is equivalent to `(a^b)^c`, as shown in the second example. Note that this is *not* `a^(b^c)`, which would be right associative.

When `nat` is on (the default), REDUCE output produces raised exponents, as shown.

A square `matrix` may also be raised to positive and negative powers with the exponentiation operator (negative powers require the matrix to be invertible). Scalar expressions and `equation`s may be raised to fractional and floating-point powers.

4.14 geqsign

GEQ — Operator

The following sign is used: $>=$

`>=` is an infix binary comparison operator, which returns *true* if its first argument is greater than or equal to its second argument.

expression `>=` *expression*

expression must evaluate to an integer or floating-point number.

Examples

```
if (3 >= 2) then yes;   ⇒  yes

a := 15;                ⇒  A := 15

if a >= 20 then big else small;

                        ⇒  small
```

Comments

The binary comparison operators can only be used for comparisons between numbers or variables that evaluate to numbers. The truth values returned by such a comparison can only be used inside programming constructs, such as `if`...`then`...`else` or `repeat`...`until` or `while`...`do`.

4.15 greater

GREATER Operator

The following sign is used: $>$

The `>` is an infix binary comparison operator that returns *true* if its first argument is strictly greater than its second.

expression `>` *expression*

expression must evaluate to a number, e.g., integer, rational or floating point number.

Examples

```
on rounded;

if 3.0 > 3 then write "different" else write "same";

                        ⇒   same

off rounded;

a := 20;                ⇒   A := 20

if a > 20 then write "bigger" else write "not bigger";

                        ⇒   not bigger
```

Comments

The binary comparison operators can only be used for comparisons between numbers or variables that evaluate to numbers. The truth values returned by such a comparison can only be used inside programming constructs, such as `if`...`then`...`else` or `repeat`...`until` or `while`...`do`.

4.16 leqsign

LEQ Operator

The following sign is used: <=

`<=` is an infix binary comparison operator that returns *true* if its first argument is less than or equal to its second argument.

expression `<=` *expression*

expression must evaluate to a number, e.g., integer, rational or floating point number.

Examples

`a := 10;`	⇒	`A := 10`
`if a <= 10 then true;`	⇒	`true`

Comments

The binary comparison operators can only be used for comparisons between numbers or variables that evaluate to numbers. The truth values returned by such a comparison can only be used inside programming constructs, such as `if`...`then`...`else` or `repeat`...`until` or `while`...`do`.

4.17 less

LESS Operator

The following sign is used: $<$

< is an infix binary logical comparison operator that returns *true* if its first argument is strictly less than its second argument.

expression < *expression*

expression must evaluate to a number, e.g., integer, rational or floating point number.

Examples

```
f := -3;                      ⇒   F := -3

if f < -3 then write "yes" else write "no";

                              ⇒   no
```

Comments

The binary comparison operators can only be used for comparisons between numbers or variables that evaluate to numbers. The truth values returned by such a comparison can only be used inside programming constructs, such as `if`...`then`...`else` or `repeat`...`until` or `while`...`do`.

4.18 tilde

~ Operator

The ~ is used as a unary prefix operator in the left-hand sides of `rule`s to mark `free variable`s. A double tilde marks an optional `free variable`.

4.19 group

GROUP Command

The following signs are used: << and >>

The `<<...>>` command is a group statement, used to group statements together where REDUCE expects a single statement.

`<<`*statement*{; *statement* `or` $*statement*}*`>>`

statement may be any valid REDUCE statement or expression.

Examples

```
a := 2;                    ⇒   A := 2

if a < 5 then <<b := a + 10; write b>>;

                           ⇒   12

<<d := c/15; f := d + 3; f**2>>;
                                 2
                                C  + 90*C + 202
                           ⇒   -----------------
                                      225
```

Comments

The value returned from a group statement is the value of the last individual statement executed inside it. Note that when a semicolon is placed between the last statement and the closing brackets, 0 or *nil* is returned. Group statements are often used in the consequence portions of `if`...`then`, `repeat`...`until`, and `while`...`do` clauses. They may also be used in interactive operation to execute several statements at one time. Statements inside the group statement are separated by semicolons or dollar signs.

4.20 AND

AND Operator

The `and` binary logical operator returns *true* if both of its arguments are *true*.

logical_expression `and` *logical_expression*

logical_expression must evaluate to *true* or *nil*.

Examples

```
a := 12;                   ⇒   A := 12

if numberp a and a < 15 then write a**2 else write "no";

                           ⇒   144

clear a;

if numberp a and a < 15 then write a**2 else write "no";

                           ⇒   no
```

Comments

Logical operators can only be used inside conditional statements, such as `while...do` or `if...then...else`. `and` examines each of its arguments in order, and quits, returning *nil*, on finding an argument that is not *true*. An error results if it is used in other contexts.

`and` is left associative: `x and y and z` is equivalent to `(x and y) and z`.

4.21 BEGIN

BEGIN — Command

`begin` is used to start a `block` statement, which is closed with `end`.

`begin` *statement*{`;` *statement*}∗ `end`

statement is any valid REDUCE statement.

Examples

```
begin for i := 1:3 do write i end;

                         ⇒   1
                             2
                             3
begin scalar n;n:=1;b:=for i:=1:4 product(x-i);return n end;

                         ⇒   1
                              4       3       2
b;                       ⇒   X  - 10*X  + 35*X  - 50*X + 24
```

Comments

A `begin`...`end` block can do actions (such as `write`), but does not return a value unless instructed to by a `return` statement, which must be the last statement executed in the block. It is unnecessary to insert a semicolon before the `end`.

Local variables, if any, are declared in the first statement immediately after `begin`, and may be defined as `scalar, integer,` or `real`. `array` variables declared within a `begin`...`end` block are global in every case, and `let` statements have global effects. A `let` statement involving a formal parameter affects the calling parameter that corresponds to it. `let` statements involving local variables make global assignments, overwriting outside variables by the same name or creating them if they do not exist. You can use this feature to affect global variables from procedures, but be careful that you do not do it inadvertently.

4.22 block

BLOCK Command

A `block` is a sequence of statements enclosed by commands `begin` and `end`.

`begin` *statement*{`;` *statement*}$*$ `end`

For more details see `begin`.

4.23 COMMENT

COMMENT Command

Beginning with the word `comment`, all text until the next statement terminator (`;` or `$`) is ignored.

Examples

```
x := a**2 comment--a is the velocity of the particle;;
```

```
                                         2
                          =>    X := A
```

Comments

Note that the first semicolon ends the comment and the second one terminates the original REDUCE statement.

Multiple-line comments are often needed in interactive files. The `comment` command allows a normal-looking text to accompany the REDUCE statements in the file.

4.24 CONS

CONS Operator

The `cons` operator adds a new element to the beginning of a `list`. Its operation is identical to the symbol `dot` (dot). It can be used infix or prefix.

`cons`(*item*, *list*) or *item* `cons` *list*

item can be any REDUCE scalar expression, including a list; *list* must be a list.

Examples

```
liss := cons(a,{b});        ⇒   {A,B}

liss := c cons liss;        ⇒   {C,A,B}

newliss := for each y in liss collect cons(y,list x);

                            ⇒   NEWLISS := {{C,X},{A,X},{B,X}}

for each y in newliss sum (first y)*(second y);

                            ⇒   X*(A + B + C)
```

Comments

If you want to use `cons` to put together two elements into a new list, you must make the second one into a list with curly brackets or the `list` command. You can also start with an empty list created by {}.

The `cons` operator is right associative: `a cons b cons c` is valid if `c` is a list; `b` need not be a list. The list produced is {`a,b,c`}.

4.25 END

END Command

The command `end` has two main uses:

(i) as the ending of a `begin`...`end block`; and

(ii) to end input from a file.

Comments

In a `begin`...`end block`, there need not be a delimiter (`;` or `$`) before the `end`, though there must be one after it, or a right bracket matching an earlier left bracket.

Files to be read into REDUCE should end with `end;`, which must be preceded by a semicolon (usually the last character of the previous line). The additional semicolon avoids problems with mistakes in the files. If you have suspended file operation by answering `n` to a `pause` command, you are still, technically speaking, "in" the file. Use `end` to exit the file.

An `end` at the top level of a program is ignored.

4.26 EQUATION

EQUATION Type

An `equation` is an expression where two algebraic expressions are connected by the (infix) operator `equal` or by `=`. For access to the components of an `equation` the operators `lhs`, `rhs` or `part` can be used. The evaluation of the left-hand side of an `equation` is controlled by the switch `evallhseqp`, while the right-hand side is evaluated unconditionally. When an `equation` is part of a logical expression, e.g. in a `if` or `while` statement, the equation is evaluated by subtracting both sides can comparing the result with zero.

Equations occur in many contexts, e.g. as arguments of the `sub` operator and in the arguments and the results of the operator `solve`. An equation can be member of a `list` and you may assign an equation to a variable. Elementary arithmetic is supported for equations: if `evallhseqp` is on, you may add and subtract equations, and you can combine an equation with a scalar expression by addition, subtraction, multiplication, division and raise an equation to a power.

Examples

```
on evallhseqp;

u:=x+y=1$

v:=2x-y=0$

2*u-v;          ⇒   - 3*y=-2
                      2
ws/3;           ⇒   y=---
                      3
```

Important: the equation must occur in the leftmost term of such an expression. For other operations, e.g. taking function values of both sides, use the `map` operator.

4.27 FIRST

FIRST Operator

The `first` operator returns the first element of a `list`.

`first`(*list*) or `first` *list*

list must be a non-empty list to avoid an error message.

Examples

```
alist := {a,b,c,d};          ⇒   ALIST := {A,B,C,D}

first alist;                 ⇒   A

blist := {x,y,{ww,aa,qq},z};

                             ⇒   BLIST := {X,Y,{WW,AA,QQ},Z}

first third blist;           ⇒   WW
```

4.28 FOR

FOR — Command

The `for` command is used for iterative loops. There are many possible forms it can take.

$$\texttt{for}\left\{\begin{array}{l} var\ \texttt{:=}\ start\ \texttt{:}\ stop \\ var\ \texttt{:=}\ start\ \texttt{step}\ inc\ \texttt{until}\ stop \\ \texttt{each}\ var\ \texttt{in}\ list\end{array}\right\}\left\{\begin{array}{l}\texttt{collect}\\ \texttt{do}\\ \texttt{join}\\ \texttt{product}\\ \texttt{sum}\end{array}\right\} expression$$

var can be any valid REDUCE identifier except `t` or `nil`, *inc*, *start* and *stop* can be any expression that evaluates to a positive or negative integer. *list* must be a valid `list` structure. The action taken must be one of the actions shown above, each of which is followed by a single REDUCE expression, statement or a `group` (`<<`...`>>`) or `block` (`begin`...`end`) statement.

Examples

```
for i := 1:10 sum i;        ⇒   55

for a := -2 step 3 until 6 product a;

                            ⇒   -8

a := 3;                     ⇒   A := 3

for iter := 4:a do write iter;

m := 0;                     ⇒   M := 0

for s := 10 step -1 until 3 do <<d := 10*s;m := m + d>>;

m;                          ⇒   520

for each x in {q,r,s} sum x**2;
                                  2     2     2
                            ⇒   Q  + R  + S
                                       1     1     1
for i := 1:4 collect 1/i;   ⇒   {1,---,---,---}
                                       2     3     4

for i := 1:3 join list solve(x**2 + i*x + 1,x);
```

⇒

```
            SQRT(3)*I + 1
{{X= ----------------,
                2

            SQRT(3)*I - 1
  X= ----------------}
                2

 {X=-1},

             SQRT(5) + 3      SQRT(5) - 3
 {X= - -------------,X=-------------}}
                2                2
```

Comments

The behavior of each of the five action words follows:

Action Word Behavior		
Keyword	Argument Type	Action
do	statement, command, group or block	Evaluates its argument once for each iteration of the loop, not saving results
collect	expression, statement, command, group, block, list	Evaluates its argument once for each iteration of the loop, storing the results in a list which is returned by the `for` statement when done
join	list or an operator which produces a list	Evaluates its argument once for each iteration of the loop, appending the elements in each individual result list onto the overall result list
product	expression, statement, command, `group` or `block`	Evaluates its argument once for each iteration of the loop, multiplying the results together and returning the overall product
sum	expression, statement, command, group or block	Evaluates its argument once for each iteration of the loop, adding the results together and returning the overall sum

For number-driven `for` statements, if the ending limit is smaller than the beginning limit (larger in the case of negative steps) the action statement is not executed at all. The iterative variable is local to the `for` statement, and does not affect the value of an identifier with the same name. For list-driven `for` statements, if the list is empty, the action statement is not executed, but no error occurs.

You can use nested `for` statements, with the inner `for` statement after the action keyword. You must make sure that your inner statement returns an expression that the outer statement can handle.

4.29 FOREACH

FOREACH Command

`foreach` is a synonym for the `for each` variant of the `for` construct. It is designed to iterate down a list, and an error will occur if a list is not used. The use of `for each` is preferred to `foreach`.

`foreach` *variable* in *list action expression*
where *action* ::= `do | product | sum | collect | join`

Examples

```
foreach x in {q,r,s} sum x**2;
```

$$\Rightarrow \quad Q^2 + R^2 + S^2$$

4.30 GEQ

GEQ Operator

The `geq` operator is a binary infix or prefix logical operator. It returns true if its first argument is greater than or equal to its second argument. As an infix operator it is identical with `>=`.

`geq`(*expression*, *expression*) or *expression* `geq` *expression*

expression can be any valid REDUCE expression that evaluates to a number.

Examples

```
a := 20;                  ⇒   A := 20

if geq(a,25) then write "big" else write "small";
                          ⇒   small

if a geq 20 then write "big" else write "small";
                          ⇒   big

if (a geq 18) then write "big" else write "small";
                          ⇒   big
```

Comments

Logical operators can only be used in conditional statements such as `if`...`then`...`else` or `repeat`...`until`.

4.31 GOTO

GOTO Command

Inside a `begin`...`end block`, `goto`, or preferably, `go to`, transfers flow of control to a labeled statement.

`go to` *labeled_statement* or `goto` *labeled_statement*

labeled_statement is of the form *label* :*statement*

Examples

```
procedure dumb(a);
   begin scalar q;
      go to lab;
      q := df(a**2 - sin(a),a);
      write q;
 lab: return a
   end;
```

	⇒	`DUMB`
`dumb(17);`	⇒	`17`

Comments

`go to` can only be used inside a `begin`...`end block`, and inside the block only statements at the top level can be labeled, not ones inside `<<`...`>>`, `while`...`do`, etc.

4.32 GREATERP

GREATERP Operator

The `greaterp` logical operator returns true if its first argument is strictly greater than its second argument. As an infix operator it is identical with `>`.

`greaterp`(*expression*, *expression*) or *expression* `greaterp` *expression*

expression can be any valid REDUCE expression that evaluates to a number.

Examples

```
a := 20;                 ⇒   A := 20

if greaterp(a,25) then write "big" else write "small";

                         ⇒   small

if a greaterp 20 then write "big" else write "small";

                         ⇒   small

if (a greaterp 18) then write "big" else write "small";

                         ⇒   big
```

Comments

Logical operators can only be used in conditional statements such as `if...then...else` or `repeat...while`.

4.33 IF

IF Command

The `if` command is a conditional statement that executes a statement if a condition is true, and optionally another statement if it is not.

`if` *condition* `then` *statement* &option(`else` *statement*)

condition must be a logical or comparison operator that evaluates to a `boolean value`. *statement* must be a single REDUCE statement or a `group` (`<<`...`>>`) or `block` (`begin`...`end`) statement.

Examples

```
if x = 5 then a := b+c else a := d+f;
                                ⇒   D + F
x := 9;                         ⇒   X := 9
if numberp x and x<20 then y := sqrt(x) else write "illegal";
                                ⇒   3
clear x;
if numberp x and x<20 then y := sqrt(x) else write "illegal";
                                ⇒   illegal
x := 12;                        ⇒   X := 12
a := if x < 5 then 100 else 150;
                                ⇒   A := 150
b := u**(if x < 10 then 2);     ⇒   B := 1
bb := u**(if x > 10 then 2);
                                         2
                                ⇒   BB := U
```

Comments

An `if` statement may be used inside an assignment statement and sets its value depending on the conditions, or used anywhere else an expression would be valid, as shown in the last example. If there is no `else` clause, the value is 0 if a number is expected, and nothing otherwise.

The `else` clause may be left out if no action is to be taken if the condition is false.

The condition may be a compound conditional statement using `and` or `or`. If a non-conditional statement, such as a constant, is used by accident, it is assumed to have value *true*.

Be sure to use `group` or `block` statements after `then` or `else`.

The `if` operator is right associative. The following constructions are examples:

(1) `if` *condition* `then if` *condition* `then` *action* `else` *action*

which is equivalent to

`if` *condition* `then` (`if` *condition* `then` *action* `else` *action*);

(2) `if` *condition* `then` *action* `else if` *condition* `then` *action* `else` *action*

which is equivalent to

`if` *condition* `then` *action* `else`
(`if` *condition* `then` *action* `else` *action*).

4.34 LIST

LIST Operator

The `list` operator constructs a list from its arguments.

`list`(*item*{, *item*}∗) or `list`() to construct an empty list.

item can be any REDUCE scalar expression, including another list. Left and right curly brackets can also be used instead of the operator `list` to construct a list.

Examples

```
liss := list(c,b,c,{xx,yy},3x**2+7x+3,df(sin(2*x),x));

                        ⇒

                                   2
       LISS := {C,B,C,{XX,YY},3*X  + 7*X + 3,2*COS(2*X)}

length liss;            ⇒   6

liss := {c,b,c,{xx,yy},3x**2+7x+3,df(sin(2*x),x)};

                        ⇒

                                   2
       LISS := {C,B,C,{XX,YY},3*X  + 7*X + 3,2*COS(2*X)}

emptylis := list();     ⇒   EMPTYLIS := {}

a . emptylis;           ⇒   {A}
```

Comments

Lists are ordered, hierarchical structures. The elements stay where you put them, and only change position in the list if you specifically change them. Lists can have nested sublists to any (reasonable) level. The `part` operator can be used to access elements anywhere within a list hierarchy. The `length` operator counts the number of top-level elements of its list argument; elements that are themselves lists still only count as one element.

4.35 OR

OR — Operator

The `or` binary logical operator returns *true* if either one or both of its arguments is *true*.

logical expression `or` *logical expression*

logical expression must evaluate to *true* or *nil*.

Examples

```
a := 10;                  ⇒   A := 10

if a<0 or a>140 then write "not a valid human age" else
   write "age = ",a;

                          ⇒   age = 10

a := 200;                 ⇒   A := 200

if a < 0 or a > 140 then write "not a valid human age";

                          ⇒   not a valid human age
```

Comments

The `or` operator is left associative: `x or y or z` is equivalent to `(x or y) or z`.

Logical operators can only be used in conditional expressions, such as `if`...`then`...`else` and `while`...`do`. `or` evaluates its arguments in order and quits, returning *true*, on finding the first *true* statement.

4.36 PROCEDURE

PROCEDURE Command

The `procedure` command allows you to define a mathematical operation as a function with arguments.

&*option* `procedure` *identifier* (*arg*{, *arg*}+); *body*

The *option* may be `algebraic` or `symbolic`, indicating the mode under which the procedure is executed, or `real` or `integer`, indicating the type of answer expected. The default is algebraic. Real or integer procedures are subtypes of algebraic procedures; type-checking is done on the results of integer procedures, but not on real procedures (in the current REDUCE release). *identifier* may be any valid REDUCE identifier that is not already a procedure name, operator, `array` or `matrix`. *arg* is a formal parameter that may be any valid REDUCE identifier. *body* is a single statement (a `group` or `block` statement may be used) with the desired activities in it.

Examples

```
procedure fac(n);
   if not (fixp(n) and n>=0)
     then rederr "Choose nonneg. integer only"
    else for i := 0:n-1 product i+1;

                        ⇒   FAC

fac(0);                 ⇒   1

fac(5);                 ⇒   120

fac(-5);                ⇒   ***** choose nonneg. integer only
```

Comments

Procedures are automatically declared as operators upon definition. When REDUCE has parsed the procedure definition and successfully converted it to a form for its own use, it prints the name of the procedure. Procedure definitions cannot be nested. Procedures can call other procedures, or can recursively call themselves. Procedure identifiers can be cleared as you would clear an operator. Unlike `let` statements, new definitions under the same procedure name replace the previous definitions completely.

Be careful not to use the name of a system operator for your own procedure. REDUCE may or may not give you a warning message. If you redefine a system operator in your own procedure, the original function of the system operator is lost for the remainder of the REDUCE session.

Procedures may have none, one, or more than one parameter. A REDUCE parameter is a formal parameter only; the use of x as a parameter in a `procedure` definition has no connection with a value of x in the REDUCE session, and the results of calling a procedure have no effect on the value of x. If a procedure is *called* with x as a parameter, the current value of x is used as specified in the computation, but is not changed outside the procedure. Making an assignment statement by `:=` with a formal parameter on the left-hand side only changes the value of the calling parameter within the procedure.

Using a `let` statement inside a procedure always changes the value globally: a `let` with a formal parameter makes the change to the calling parameter. `let` statements cannot be made on local variables inside `begin`...`end blocks`. When `clear` statements are used on formal parameters, the calling variables associated with them are cleared globally too. The use of `let` or `clear` statements inside procedures should be done with extreme caution.

Arrays and operators may be used as parameters to procedures. The body of the procedure can contain statements that appropriately manipulate these arguments. Changes are made to values of the calling arrays or operators. Simple expressions can also be used as arguments, in the place of scalar variables. Matrices may *not* be used as arguments to procedures.

A procedure that has no parameters is called by the procedure name, immediately followed by empty parentheses. The empty parentheses may be left out when writing a procedure with no parameters, but must appear in a call of the procedure. If this is a nuisance to you, use a `let` statement on the name of the procedure (i.e., `let noargs = noargs()`) after which you can call the procedure by just its name.

Procedures that have a single argument can leave out the parentheses around it both in the definition and procedure call. (You can use the parentheses if you wish.) Procedures with more than one argument must use parentheses, with the arguments separated by commas.

Procedures often have a `begin`...`end` block in them. Inside the block, local variables are declared using `scalar`, `real` or `integer` declarations. The declarations must be made immediately after the word `begin`, and if more than one type of declaration is made, they are separated by semicolons. REDUCE currently does no type checking on local variables; `real` and `integer` are treated just like `scalar`. Actions take place as specified in the statements inside the block statement. Any identifiers that are not formal parameters or local variables are treated as global variables, and activities involving these identifiers are global in effect.

If a return value is desired from a procedure call, a specific `return` command must be the last statement executed before exiting from the procedure. If no `return` is used, a procedure returns a zero or no value.

Procedures are often written in a file using an editor, then the file is input using the command `in`. This method allows easy changes in development, and also allows you to load the named procedures whenever you like, by loading the files that contain them.

4.37 REPEAT

REPEAT — Command

The `repeat` command causes repeated execution of a statement `until` the given condition is found to be true. The statement is always executed at least once.

`repeat` *statement* `until` *condition*

statement can be a single statement, `group` statement, or a `begin`...`end block`. *condition* must be a logical operator that evaluates to *true* or *nil*.

Examples

```
<<m := 4; repeat <<write 100*x*m;m := m-1>> until m = 0>>;
```

$\Rightarrow$
```
400*X
300*X
200*X
100*X
```

```
<<m := -1; repeat <<write m; m := m-1>> until m <= 0>>;
```

$\Rightarrow$
```
-1
```

Comments

`repeat` must always be followed by an `until` with a condition. Be careful not to generate an infinite loop with a condition that is never true. In the second example, if the condition had been `m = 0`, it would never have been true since `m` already had value -2 when the condition was first evaluated.

4.38 REST

REST **Operator**

The `rest` operator returns a `list` containing all but the first element of the list it is given.

`rest`(*list*) or `rest` *list*

list must be a non-empty list, but need not have more than one element.

Examples

```
alist := {a,b,c,d};         ⇒   ALIST := {A,B,C,D};
rest alist;                 ⇒   {B,C,D}
blist := {x,y,{aa,bb,cc},z};
                            ⇒   BLIST := {X,Y,{AA,BB,CC},Z}
second rest blist;          ⇒   {AA,BB,CC}
clist := {c};               ⇒   CLIST := C
rest clist;                 ⇒   {}
```

4.39 RETURN

RETURN Command

The `return` command causes a value to be returned from inside a `begin`...`end` `block`.

`begin` *statements* `return` *&option(expression)* `end`

statements can be any valid REDUCE statements. The value of *expression* is returned.

Examples

```
begin write "yes"; return a end;
                              ⇒  yes
                                 A

procedure dumb(a);
  begin if numberp(a) then return a else return 10 end;
                              ⇒  DUMB

dumb(x);                      ⇒  10

dumb(-5);                     ⇒  -5

procedure dumb2(a);
  begin c := a**2 + 2*a + 1; d := 17; c*d; return end;
                              ⇒  DUMB2

dumb2(4);

c;                            ⇒  25

d;                            ⇒  17
```

Comments

Note in `dumb2` above that the assignments were made as requested, but the product `c*d` cannot be accessed. Changing the procedure to read `return c*d` would remedy this problem.

The `return` statement is always the last statement executed before leaving the block. If `return` has no argument, the block is exited but no value is returned. A block statement does not need a `return` ; the statements inside terminate in their normal fashion without one. In that case no value is returned, although the specified actions inside the block take place.

The `return` command can be used inside `<<`...`>>` `group` statements and `if`...`then`...`else` commands that are inside `begin`...`end` `block`s. It is not valid in these constructions that are not inside a `begin`...`end` block. It is not valid inside `for`, `repeat`...`until` or `while`...`do` loops in any construction. To force early termination from loops, the `go to`(`goto`) command must be used. When you use nested block statements, a `return` from an inner block exits returning a value to the next-outermost block, rather than all the way to the outside.

4.40 REVERSE

REVERSE — Operator

The `reverse` operator returns a `list` that is the reverse of the list it is given.

`reverse`(*list*) or `reverse` *list*

list must be a `list`.

Examples

```
aa := {c,b,a,{x**2,z**3},y};
                                              2   3
                            =>   AA := {C,B,A,{X ,Z },Y}
                                    2   3
reverse aa;                 =>   {Y,{X ,Z },A,B,C}
                                        2   3
reverse(q . reverse aa);    =>   {C,B,A,{X ,Z },Y,Q}
```

Comments

`reverse` and `cons` can be used together to add a new element to the end of a list (`.` adds its new element to the beginning). The `reverse` operator uses a noticeable amount of system resources, especially if the list is long. If you are doing much heavy-duty list manipulation, you should probably design your algorithms to avoid much reversing of lists. A moderate amount of list reversing is no problem.

4.41 RULE

RULE Type

A `rule` is an instruction to replace an algebraic expression or a part of an expression by another one.

lhs =¿ *rhs* or *lhs* =¿ *rhs* `when` *cond*

lhs is an algebraic expression used as search pattern and *rhs* is an algebraic expression which replaces matches of *rhs*. `=>` is the operator `replace`.

lhs can contain `free variable`s which are symbols preceded by a tilde `~` in their leftmost position in *lhs*. A double tilde marks an `optional free variable`. If a rule has a `when` *cond* part it will fire only if the evaluation of *cond* has a result `true`. *cond* may contain references to free variables of *lhs*.

Rules can be collected in a `list` which then forms a `rule list`. `Rule lists` can be used to collect algebraic knowledge for a specific evaluation context.

`Rules` and `rule lists` are globally activated and deactivated by `let`, `forall`, `clearrules`. For a single evaluation they can be locally activate by `where`. The active rules for an operator can be visualized by `showrules`.

Examples

```
operator f,g,h;

let f(x) => x^2;

                                   2
f(x);                       ⇒    x

g_rules:={g(~n,~x)=>h(n/2,x) when evenp n,

g(~n,~x)=>h((1-n)/2,x) when not evenp n}

let g_rules;

g(3,x);                     ⇒    h(-1,x)
```

4.42 Free Variable

FREE VARIABLE Type

A variable preceded by a tilde is considered as `free variable` and stands for an arbitrary part in an algebraic form during pattern matching. Free variables occur in the left-hand sides of `rule`s, in the side relations for `compact` and in the first arguments of `map` and `select` calls. See `rule` for examples.

In rules also `optional free variable`s may occur.

4.43 Optional Free Variable

OPTIONAL FREE VARIABLE — Type

A variable preceded by a double tilde is considered as `optional free variable` and stands for an arbitrary part part in an algebraic form during pattern matching. In contrast to ordinary `free variable`s an operator pattern with an `optional free variable` matches also if the operand for the variable is missing. In such a case the variable is bound to a neutral value. Optional free variables can be used as

term in a sum: set to 0 if missing,

factor in a product: set to 1 if missing,

exponent: set to 1 if missing

Examples

```
sin(~~u + ~~n * pi) => sin(u) when evenp u;
```

$$\Rightarrow$$

Optional free variables are allowed only in the left-hand sides of `rule`s.

4.44 SECOND

SECOND Operator

The `second` operator returns the second element of a list.

`second`(*list*) or `second` *list*

list must be a list with at least two elements, to avoid an error message.

Examples

`alist := {a,b,c,d};`	⇒	`ALIST := {A,B,C,D}`
`second alist;`	⇒	`B`
`blist := {x,{aa,bb,cc},z};`	⇒	`BLIST := {X,{AA,BB,CC},Z}`
`second second blist;`	⇒	`BB`

4.45 SET

SET — Operator

The `set` operator is used for assignments when you want both sides of the assignment statement to be evaluated.

`set`(*restricted_expression*, *expression*)

expression can be any REDUCE expression; *restricted_expression* must be an identifier or an expression that evaluates to an identifier.

Examples

```
a := y;              =>   A := Y
                                2
set(a,sin(x^2));     =>   SIN(X )
                                2
a;                   =>   SIN(X )
                                2
y;                   =>   SIN(X )

a := b + c;          =>   A := B + C

set(a-c,z);          =>   Z

b;                   =>   Z
```

Comments

Using an `array` or `matrix` reference as the first argument to `set` has the result of setting the *contents* of the designated element to `set`'s second argument. You should be careful to avoid unwanted side effects when you use this facility.

4.46 SETQ

SETQ — Operator

The `setq` operator is an infix or prefix binary assignment operator. It is identical to `:=`.

> `setq`(*restricted_expression*, *expression*) or
> *restricted_expression* `setq` *expression*

restricted expression is ordinarily a single identifier, though simple expressions may be used (see Comments below). *expression* can be any valid REDUCE expression. If *expression* is a `matrix` identifier, then *restricted_expression* can be a matrix identifier (redimensioned if necessary), which has each element set to the corresponding elements of the identifier on the right-hand side.

Examples

`setq(b,6);`	⇒	`B := 6`
`c setq sin(x);`	⇒	`C := SIN(X)`
`w + setq(c,x+3) + z;`	⇒	`W + X + Z + 3`
`c;`	⇒	`X + 3`
`setq(a1 + a2,25);`	⇒	`A1 + A2 := 25`
`a1;`	⇒	`- (A2 - 25)`

Comments

Embedding a `setq` statement in an expression has the side effect of making the assignment, as shown in the third example above.

Assignments are generally done for identifiers, but may be done for simple expressions as well, subject to the following remarks:

(i) If the left-hand side is an identifier, an operator, or a power, the rule is added to the rule table.

(ii) If the operators `- + /` appear on the left-hand side, all but the first term of the expression is moved to the right-hand side.

(iii) If the operator `*` appears on the left-hand side, any constant terms are moved to the right-hand side, but the symbolic factors remain.

Be careful not to make a recursive `setq` assignment that is not controlled inside a loop statement. The process of resubstitution continues until you get a stack overflow message. `setq` can be used to attach functionality to operators, as the `:=` does.

4.47 THIRD

THIRD Operator

The `third` operator returns the third item of a `list`.

`third`(*list*) or `third` *list*

list must be a list containing at least three items to avoid an error message.

Examples

```
alist := {a,b,c,d};          ⇒   ALIST := {A,B,C,D}

third alist;                 ⇒   C

blist := {x,{aa,bb,cc},y,z};

                             ⇒   BLIST := {X,{AA,BB,CC},Y,Z};

third second blist;          ⇒   CC

third blist;                 ⇒   Y
```

4.48 WHEN

WHEN Operator

The `when` operator is used inside a `rule` to make the execution of the rule depend on a boolean condition which is evaluated at execution time. For the use see `rule`.

5 Arithmetic Operations

5.1 ARITHMETIC_OPERATIONS

ARITHMETIC_OPERATIONS Introduction

This section considers operations defined in REDUCE that concern numbers, or operators that can operate on numbers in addition, in most cases, to more general expressions.

5.2 ABS

ABS Operator

The `abs` operator returns the absolute value of its argument.

`abs`(*expression*)

expression can be any REDUCE scalar expression.

Examples

```
abs(-a);     ⇒  ABS(A)
abs(-5);     ⇒  5
a := -10;    ⇒  A := -10
abs(a);      ⇒  10
abs(-a);     ⇒  10
```

Comments

If the argument has had no numeric value assigned to it, such as an identifier or polynomial, `abs` returns an expression involving `abs` of its argument, doing as much simplification of the argument as it can, such as dropping any preceding minus sign.

5.3 ADJPREC

ADJPREC Switch

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.

Examples

```
on rounded;

1.23456789012345;       ⇒   1.23456789012

on adjprec;

1.23456789012345;

*** precision increased to 15

1.23456789012345        ⇒
```

5.4 ARG

ARG Operator

If `complex` and `rounded` are on, and *arg* evaluates to a complex number, `arg` returns the polar angle of *arg*, measured in radians. Otherwise an expression in *arg* is returned.

Examples

```
arg(3+4i)                ⇒    ARG(3 + 4*I)

on rounded, complex;

ws;                      ⇒    0.927295218002

arg a;                   ⇒    ARG(A)
```

5.5 CEILING

CEILING — Operator

`ceiling`(*expression*)

This operator returns the ceiling (i.e., the least integer greater than or equal to its argument) if its argument has a numerical value. For negative numbers, this is equivalent to `fix`. For non-numeric arguments, the value is an expression in the original operator.

Examples

`ceiling 3.4;`	⇒	`4`
`fix 3.4;`	⇒	`3`
`ceiling(-5.2);`	⇒	`-5`
`fix(-5.2);`	⇒	`-5`
`ceiling a;`	⇒	`CEILING(A)`

5.6 CHOOSE

CHOOSE — Operator

`choose`(m,m) returns the number of ways of choosing m objects from a collection of n distinct objects — in other words the binomial coefficient. If m and n are not positive integers, or $m > n$, the expression is returned unchanged. than or equal to

Examples

`choose(2,3);` $\Rightarrow$ `3`

`choose(3,2);` $\Rightarrow$ `CHOOSE(3,2)`

`choose(a,b);` $\Rightarrow$ `CHOOSE(A,B)`

5.7 DEG2DMS

DEG2DMS   Operator

`deg2dms`(*expression*)

In `rounded` mode, if *expression* is a real number, the operator `deg2dms` will interpret it as degrees, and convert it to a list containing the equivalent degrees, minutes and seconds. In all other cases, an expression in terms of the original operator is returned.

Examples

```
deg2dms 60;     ⇒   DEG2DMS(60)

on rounded;

ws;             ⇒   {60,0,0}

deg2dms 42.4;   ⇒   {42,23,60.0}

deg2dms a;      ⇒   DEG2DMS(A)
```

5.8 DEG2RAD

DEG2RAD Operator

`deg2rad`(*expression*)

In `rounded` mode, if *expression* is a real number, the operator `deg2rad` will interpret it as degrees, and convert it to the equivalent radians. In all other cases, an expression in terms of the original operator is returned.

Examples

`deg2rad 60;`	⇒	`DEG2RAD(60)`
`on rounded;`		
`ws;`	⇒	`1.0471975512`
`deg2rad a;`	⇒	`DEG2RAD(A)`

5.9 DIFFERENCE

DIFFERENCE Operator

The `difference` operator may be used as either an infix or prefix binary subtraction operator. It is identical to `-` as a binary operator.

`difference`(*expression*, *expression*) or

expression `difference` *expression* {`difference` *expression*}∗

expression can be a number or any other valid REDUCE expression. Matrix expressions are allowed if they are of the same dimensions.

Examples

```
difference(10,4);          ⇒   6

15 difference 5 difference 2;

                           ⇒   8

a difference b;            ⇒   A - B
```

Comments

The `difference` operator is left associative, as shown in the second example above.

5.10 DILOG

DILOG   Operator

The `dilog` operator is known to the differentiation and integration operators, but has numeric value attached only at `dilog(0)`. Dilog is defined by

$$dilog(x) = -\int \frac{log(x)\,dx}{x-1}$$

Examples

```
                                     2
                          2*LOG(X )*X
df(dilog(x**2),x);  =>  - -------------
                               2
                              X  - 1

int(dilog(x),x);    =>

       DILOG(X)*X - DILOG(X) + LOG(X)*X - X

                            2
                          PI
dilog(0);           =>   -----
                           6
```

5.11 DMS2DEG

DMS2DEG — Operator

`dms2deg`(*list*)

In `rounded` mode, if *list* is a list of three real numbers, the operator `dms2deg` will interpret the list as degrees, minutes and seconds and convert it to the equivalent degrees. In all other cases, an expression in terms of the original operator is returned.

Examples

`dms2deg {42,3,7};`	⇒	`DMS2DEG({42,3,7})`
`on rounded;`		
`ws;`	⇒	`42.0519444444`
`dms2deg a;`	⇒	`DMS2DEG(A)`

5.12 DMS2RAD

DMS2RAD Operator

`dms2rad`(*list*)

In `rounded` mode, if *list* is a list of three real numbers, the operator `dms2rad` will interpret the list as degrees, minutes and seconds and convert it to the equivalent radians. In all other cases, an expression in terms of the original operator is returned.

Examples

```
dms2rad {42,3,7};   ⇒   DMS2RAD({42,3,7})

on rounded;

ws;                 ⇒   0.733944887421

dms2rad a;          ⇒   DMS2RAD(A)
```

5.13 FACTORIAL

FACTORIAL — Operator

`factorial`(*expression*)

If the argument of `factorial` is a positive integer or zero, its factorial is returned. Otherwise the result is expressed in terms of the original operator. For more general operations, the `gamma` operator is available in the `Special Function Package`.

Examples

`factorial 4;`	⇒	24
`factorial 30 ;`	⇒	265252859812191058636308480000000
`factorial(a) ; FACTORIAL(A)`	⇒	

5.14 FIX

FIX Operator

`fix`(*expression*)

The operator `fix` returns the integer part of its argument, if that argument has a numerical value. For positive numbers, this is equivalent to `floor`, and, for negative numbers, `ceiling`. For non-numeric arguments, the value is an expression in the original operator.

Examples

`fix 3.4;`	$\Rightarrow$	`3`
`floor 3.4;`	$\Rightarrow$	`3`
`ceiling 3.4;`	$\Rightarrow$	`4`
`fix(-5.2);`	$\Rightarrow$	`-5`
`floor(-5.2);`	$\Rightarrow$	`-6`
`ceiling(-5.2);`	$\Rightarrow$	`-5`
`fix(a);`	$\Rightarrow$	`FIX(A)`

5.15 FIXP

FIXP Operator

The `fixp` logical operator returns true if its argument is an integer.

`fixp`(*expression*) or `fixp` *simple_expression*

expression can be any valid REDUCE expression, *simple_expression* must be a single identifier or begin with a prefix operator.

Examples

```
if fixp 1.5 then write "ok" else write "not";

                        ⇒   not

if fixp(a) then write "ok" else write "not";

                        ⇒   not

a := 15;                ⇒   A := 15

if fixp(a) then write "ok" else write "not";

                        ⇒   ok
```

Comments

Logical operators can only be used inside conditional expressions such as `if`...`then` or `while`...`do`.

5.16 FLOOR

FLOOR Operator

floor(*expression*)

This operator returns the floor (i.e., the greatest integer less than or equal to its argument) if its argument has a numerical value. For positive numbers, this is equivalent to `fix`. For non-numeric arguments, the value is an expression in the original operator.

Examples

`floor 3.4;`	$\Rightarrow$	`3`
`fix 3.4;`	$\Rightarrow$	`3`
`floor(-5.2);`	$\Rightarrow$	`-6`
`fix(-5.2);`	$\Rightarrow$	`-5`
`floor a;`	$\Rightarrow$	`FLOOR(A)`

5.17 EXPT

EXPT Operator

The `expt` operator is both an infix and prefix binary exponentiation operator. It is identical to `^` or `**`.

`expt`(*expression*, *expression*) or *expression* `expt` *expression*

Examples

```
                    B
a expt b;      ⇒   A
                    B
expt(a,b);     ⇒   A
                    4      3        2  2        3    4
(x+y) expt 4;  ⇒   X  + 4*X *Y + 6*X *Y  + 4*X*Y  + Y
```

Comments

Scalar expressions may be raised to fractional and floating-point powers. Square matrix expressions may be raised to positive powers, and also to negative powers if non-singular.

`expt` is left associative. In other words, `a expt b expt c` is equivalent to `a expt (b*c)`, not `a expt (b expt c)`, which would be right associative.

5.18 GCD

GCD Operator

The `gcd` operator returns the greatest common divisor of two polynomials.

`gcd`(*expression*, *expression*)

expression must be a polynomial (or integer), otherwise an error occurs.

Examples

```
gcd(2*x**2 - 2*y**2,4*x + 4*y);
                              ⇒   2*(X + Y)
gcd(sin(x),x**2 + 1);        ⇒   1
gcd(765,68);                 ⇒   17
```

Comments

The operator `gcd` described here provides an explicit means to find the gcd of two expressions. The switch `gcd` described below simplifies expressions by finding and canceling gcd's at every opportunity. When the switch `ezgcd` is also on, gcd's are figured using the EZ GCD algorithm, which is usually faster.

5.19 LN

LN Operator

`ln`(*expression*)

expression can be any valid scalar REDUCE expression.

The `ln` operator returns the natural logarithm of its argument. However, unlike `log`, there are no algebraic rules associated with it; it will only evaluate when `rounded` is on, and the argument is a real number.

Examples

```
ln(x);          ⇒   LN(X)

ln 4;           ⇒   LN(4)

ln(e);          ⇒   LN(E)

df(ln(x),x);    ⇒   DF(LN(X),X)

on rounded;

ln 4;           ⇒   1.38629436112

ln e;           ⇒   1
```

Comments

Because of the restricted algebraic properties of `ln`, users are advised to use `log` whenever possible.

5.20 LOG

LOG — Operator

The `log` operator returns the natural logarithm of its argument.

`log`(*expression*) or `log` *expression*

expression can be any valid scalar REDUCE expression.

Examples

```
log(x);        ⇒   LOG(X)

log 4;         ⇒   LOG(4)

log(e);        ⇒   1

on rounded;

log 4;         ⇒   1.38629436112
```

Comments

`log` returns a numeric value only when `rounded` is on. In that case, use of a negative argument for `log` results in an error message. No error is given on a negative argument when REDUCE is not in that mode.

5.21 LOGB

LOGB Operator

logb(*expression integer*)

expression can be any valid scalar REDUCE expression.

The `logb` operator returns the logarithm of its first argument using the second argument as base. However, unlike `log`, there are no algebraic rules associated with it; it will only evaluate when `rounded` is on, and the first argument is a real number.

Examples

```
logb(x,2);          ⇒  LOGB(X,2)

logb(4,3);          ⇒  LOGB(4,3)

logb(2,2);          ⇒  LOGB(2,2)

df(logb(x,3),x);    ⇒  DF(LOGB(X,3),X)

on rounded;

logb(4,3);          ⇒  1.26185950714

logb(2,2);          ⇒  1
```

5.22 MAX

MAX **Operator**

The operator `max` is an n-ary prefix operator, which returns the largest value in its arguments.

`max`(*expression*{, *expression*}∗)

expression must evaluate to a number. `max` of an empty list returns 0.

Examples

```
max(4,6,10,-1);           ⇒   10

<<a := 23;b := 2*a;c := 4**2;max(a,b,c)>>;

                          ⇒   46

max(-5,-10,-a);           ⇒   -5
```

5.23 MIN

MIN Operator

The operator `min` is an n-ary prefix operator, which returns the smallest value in its arguments.

`min`(*expression*{, *expression*}∗)

expression must evaluate to a number. `min` of an empty list returns 0.

Examples

```
min(-3,0,17,2);            ⇒   -3

&lt;&lt;a := 23;b := 2*a;c := 4**2;min(a,b,c)&gt;&gt;;

                           ⇒   16

min(5,10,a);               ⇒   5
```

5.24 MINUS

MINUS — Operator

The `minus` operator is a unary minus, returning the negative of its argument. It is equivalent to the unary `-`.

`minus`(*expression*)

expression may be any scalar REDUCE expression.

Examples

```
minus(a);          ⇒   - A

minus(-1);         ⇒   1

                          4       3       2
minus((x+1)**4);   ⇒   - (X  + 4*X  + 6*X  + 4*X + 1)
```

5.25 NEXTPRIME

NEXTPRIME Operator

`nextprime`(*expression*)

If the argument of `nextprime` is an integer, the least prime greater than that argument is returned. Otherwise, a type error results.

Examples

`nextprime 5001;`	⇒	`5003`
`nextprime(10^30);`	⇒	`1000000000000000000000000000057`
`nextprime a;`	⇒	`***** A invalid as integer`

5.26 NOCONVERT

NOCONVERT Switch

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.

Examples

```
on rounded;

1.0000000000001;  ⇒  1

on noconvert;

1.0000000000001;  ⇒  1.0
```

5.27 NORM

NORM Operator

norm(*expression*)

If `rounded` is on, and the argument is a real number, *norm* returns its absolute value. If `complex` is also on, *norm* returns the square root of the sum of squares of the real and imaginary parts of the argument. In all other cases, a result is returned in terms of the original operator.

Examples

```
norm (-2);    ⇒  NORM(-2)

on rounded;

ws;           ⇒  2.0

norm(3+4i);   ⇒  NORM(4*I+3)

on complex;

ws;           ⇒  5.0
```

5.28 PERM

PERM Operator

perm(*expression1*,*expression2*)

If *expression1* and *expression2* evaluate to positive integers, `perm` returns the number of permutations possible in selecting *expression1* objects from *expression2* objects. In other cases, an expression in the original operator is returned.

Examples

`perm(1,1);`	⇒	`1`
`perm(3,5);`	⇒	`60`
`perm(-3,5);`	⇒	`PERM(-3,5)`
`perm(a,b);`	⇒	`PERM(A,B)`

5.29 PLUS

PLUS Operator

The `plus` operator is both an infix and prefix n-ary addition operator. It exists because of the way in which REDUCE handles such operators internally, and is not recommended for use in algebraic mode programming. `plussign`, which has the identical effect, should be used instead.

`plus`(*expression*, *expression*{, *expression*}∗) or
expression `plus` *expression* {`plus` *expression*}∗

expression can be any valid REDUCE expression, including matrix expressions of the same dimensions.

Examples

```
a plus b plus c plus d;  ⇒  A + B + C + D
                              29
4.5 plus 10;             ⇒  ----
                              2


                              2     2
plus(x**2,y**2);         ⇒  X   + Y
```

5.30 QUOTIENT

QUOTIENT Operator

The `quotient` operator is both an infix and prefix binary operator that returns the quotient of its first argument divided by its second. It is also a unary `recip`rocal operator. It is identical to `/` and `slash`.

`quotient`(*expression*, *expression*) or *expression* `quotient` *expression* or `quotient`(*expression*) or `quotient` *expression*

expression can be any valid REDUCE scalar expression. Matrix expressions can also be used if the second expression is invertible and the matrices are of the correct dimensions.

Examples

```
                                 A
quotient(a,x+1);          ⇒   -------
                               X + 1
                                7
7 quotient 17;            ⇒   ----
                               17

on rounded;

4.5 quotient 2;           ⇒   2.25

quotient(x**2 + 3*x + 2,x+1);

                          ⇒   X + 2

matrix m,inverse;

m := mat((a,b),(c,d));    ⇒   M(1,1) := A;
                              M(1,2) := B;
                              M(2,1) := C
                              M(2,2) := D
```

```
inverse := quotient m;   ⇒   INVERSE(1,1) :=      D
                                             -----------
                                              A*D - B*C

                              INVERSE(1,2) := -     B
                                               -----------
                                                A*D - B*C

                              INVERSE(2,1) := -     C
                                               -----------
                                                A*D - B*C

                              INVERSE(2,2) :=      A
                                             -----------
                                              A*D - B*C
```

Comments

The `quotient` operator is left associative: `a quotient b quotient c` is equivalent to `(a quotient b) quotient c`.

If a matrix argument to the unary `quotient` is not invertible, or if the second matrix argument to the binary quotient is not invertible, an error message is given.

5.31 RAD2DEG

RAD2DEG — Operator

`rad2deg`(*expression*)

In `rounded` mode, if *expression* is a real number, the operator `rad2deg` will interpret it as radians, and convert it to the equivalent degrees. In all other cases, an expression in terms of the original operator is returned.

Examples

```
rad2deg 1;    ⇒   RAD2DEG(1)

on rounded;

ws;           ⇒   57.2957795131

rad2deg a;    ⇒   RAD2DEG(A)
```

5.32 RAD2DMS

RAD2DMS — Operator

`rad2dms`(*expression*)

In `rounded` mode, if *expression* is a real number, the operator `rad2dms` will interpret it as radians, and convert it to a list containing the equivalent degrees, minutes and seconds. In all other cases, an expression in terms of the original operator is returned.

Examples

```
rad2dms 1;    ⇒   RAD2DMS(1)

on rounded;

ws;           ⇒   {57,17,44.8062470964}

rad2dms a;    ⇒   RAD2DMS(A)
```

5.33 RECIP

RECIP — Operator

`recip` is the alphabetical name for the division operator `/` or `slash` used as a unary operator. The use of `/` is preferred.

Examples

```
recip a;  ⇒    1
              ---
               A
recip 2;  ⇒    1
              ---
               2
```

5.34 REMAINDER

REMAINDER Operator

The `remainder` operator returns the remainder after its first argument is divided by its second argument.

`remainder`(*expression*, *expression*)

expression can be any valid REDUCE polynomial, and is not limited to numeric values.

Examples

```
remainder(13,6);          ⇒  1

remainder(x**2 + 3*x + 2,x+1);
                          ⇒  0

remainder(x**3 + 12*x + 4,x**2 + 1);
                          ⇒  11*X + 4

remainder(sin(2*x),x*y);  ⇒  SIN(2*X)
```

Comments

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.

If the first argument to `remainder` contains a denominator not equal to 1, an error occurs.

5.35 ROUND

ROUND — Operator

round(*expression*)

If its argument has a numerical value, `round` rounds it to the nearest integer. For non-numeric arguments, the value is an expression in the original operator.

Examples

```
round 3.4;  ⇒  3
round 3.5;  ⇒  4
round a;    ⇒  ROUND(A)
```

5.36 SETMOD

SETMOD Command

The `setmod` command sets the modulus value for subsequent `modular` arithmetic.

`setmod` *integer*

integer must be positive, and greater than 1. It need not be a prime number.

Examples

```
setmod 6;         =>   1

on modular;

16;               =>   4
                        2
x^2 + 5x + 7;     =>   X  + 5*X + 1
                        X
x/3;              =>   ---
                        3
setmod 2;         =>   6
                        4
(x+1)^4;          =>   X  + 1

x/3;              =>   X
```

Comments

`setmod` returns the previous modulus, or 1 if none has been set before. `setmod` only has effect when `modular` is on.

Modular operations are done only on numbers such as coefficients of polynomials, not on the exponents. The modulus need not be prime. Attempts to divide by a power of the modulus produces an error message, since the operation is equivalent to dividing by 0. However, dividing by a factor of a non-prime modulus does not produce an error message.

5.37 SIGN

SIGN Operator

`sign` *expression*

`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.

Examples

`sign(-5)` ⇒ `-1`

`sign(-a^2*b)` ⇒ `-SIGN(B)`

Comments

Even powers of formal expressions are assumed to be positive only as long as the switch `complex` is off.

5.38 SQRT

SQRT Operator

The `sqrt` operator returns the square root of its argument.

`sqrt`(*expression*)

expression can be any REDUCE scalar expression.

Examples

`sqrt(16*a^3);`	⇒	`4*SQRT(A)*A`
`sqrt(17);`	⇒	`SQRT(17)`
`on rounded;`		
`sqrt(17);`	⇒	`4.12310562562`
`off rounded;`		
`sqrt(a*b*c^5*d^3*27);`	⇒	

```
                                                 2
       3*SQRT(D)*SQRT(C)*SQRT(B)*SQRT(A)*SQRT(3)*C *D
```

Comments

`sqrt` checks its argument for squared factors and removes them.

Numeric values for square roots that are not exact integers are given only when `rounded` is on.

Please note that `sqrt(a**2)` is given as `a`, which may be incorrect if `a` eventually has a negative value. If you are programming a calculation in which this is a concern, you can turn on the `precise` switch, which causes the absolute value of the square root to be returned.

5.39 TIMES

TIMES   Operator

The `times` operator is an infix or prefix n-ary multiplication operator. It is identical to `*`.

expression `times` *expression* {`times` *expression*}$*$

or `times`(*expression*, *expression*{, *expression*}$*$)

expression can be any valid REDUCE scalar or matrix expression. Matrix expressions must be of the correct dimensions. Compatible scalar and matrix expressions can be mixed.

Examples

```
var1 times var2;          ⇒   VAR1*VAR2

times(6,5);               ⇒   30

matrix aa,bb;

aa := mat((1),(2),(x))$

bb := mat((0,3,1))$

aa times bb times 5;      ⇒   [0   15    5 ]
                              [            ]
                              [0   30   10 ]
                              [            ]
                              [0  15*X  5*X]
```

6 Boolean Operators

6.1 boolean value

BOOLEAN VALUE — Concept

There are no extra symbols for the truth values true and false. Instead, `nil` and the number zero are interpreted as truth value false in algebraic programs (see `false`), while any different value is considered as true (see `true`).

6.2 EQUAL

EQUAL — Operator

The operator `equal` is an infix binary comparison operator. It is identical with `=`. It returns `true` if its two arguments are equal.

expression `equal` *expression*

Equality is given between floating point numbers and integers that have the same value.

Examples

```
on rounded;

a := 4;                      ⇒   A := 4

b := 4.0;                    ⇒   B := 4.0

if a equal b then write "true" else write "false";

                             ⇒   true

if a equal 5 then write "true" else write "false";

                             ⇒   false

if a equal sqrt(16) then write "true" else write "false";

                             ⇒   true
```

Comments

Comparison operators can only be used as conditions in conditional commands such as `if...then` and `repeat...until`. *equal* can also be used as a prefix operator. However, this use is not encouraged.

6.3 EVENP

EVENP Operator

The `evenp` logical operator returns `true` if its argument is an even integer, and `nil` if its argument is an odd integer. An error message is returned if its argument is not an integer.

`evenp`(*integer*) or `evenp` *integer*

integer must evaluate to an integer.

Examples

```
aa := 1782;                 ⇒   AA := 1782

if evenp aa then yes else no;

                            ⇒   YES

if evenp(-3) then yes else no;

                            ⇒   NO
```

Comments

Although you would not ordinarily enter an expression such as the last example above, note that the negative term must be enclosed in parentheses to be correctly parsed. The `evenp` operator can only be used in conditional statements such as `if`...`then`...`else` or `while`...`do`.

6.4 false

FALSE — Concept

The symbol `nil` and the number zero are considered as `boolean value` false if used in a place where a boolean value is required. Most builtin operators return `nil` as false value. Algebraic programs use better zero. Note that `nil` is not printed when returned as result to a top level evaluation.

6.5 FREEOF

FREEOF Operator

The `freeof` logical operator returns `true` if its first argument does not contain its second argument anywhere in its structure.

`freeof`(*expression*, *kernel*) or *expression* `freeof` *kernel*

expression can be any valid scalar REDUCE expression, *kernel* must be a kernel expression (see `kernel`).

Examples

```
a := x + sin(y)**2 + log sin z;
```

```
                                            2
                        ⇒   A := LOG(SIN(Z)) + SIN(Y)  + X
```

```
if freeof(a,sin(y)) then write "free" else write "not free";
```

```
                        ⇒   not free
```

```
if freeof(a,sin(x)) then write "free" else write "not free";
```

```
                        ⇒   free
```

```
if a freeof sin z then write "free" else write "not free";
```

```
                        ⇒   not free
```

Comments

Logical operators can only be used in conditional expressions such as `if`...`then` or `while`...`do`.

6.6 LEQ

LEQ Operator

The `leq` operator is a binary infix or prefix logical operator. It returns `true` if its first argument is less than or equal to its second argument. As an infix operator it is identical with `<=`.

`leq`(*expression*, *expression*) or *expression* `leq` *expression*

expression can be any valid REDUCE expression that evaluates to a number.

Examples

```
a := 15;                  ⇒  A := 15

if leq(a,25) then write "yes" else write "no";
                          ⇒  yes

if leq(a,15) then write "yes" else write "no";
                          ⇒  yes

if leq(a,5) then write "yes" else write "no";
                          ⇒  no
```

Comments

Logical operators can only be used in conditional statements such as `if...then...else` or `while...do`.

6.7 LESSP

LESSP Operator

The `lessp` operator is a binary infix or prefix logical operator. It returns `true` if its first argument is strictly less than its second argument. As an infix operator it is identical with `<`.

`lessp`(*expression*, *expression*) or *expression* `lessp` *expression*

expression can be any valid REDUCE expression that evaluates to a number.

Examples

```
a := 15;                    ⇒   A := 15

if lessp(a,25) then write "yes" else write "no";

                            ⇒   yes

if lessp(a,15) then write "yes" else write "no";

                            ⇒   no

if lessp(a,5) then write "yes" else write "no";

                            ⇒   no
```

Comments

Logical operators can only be used in conditional statements such as `if`...`then`...`else` or `while`...`do`.

6.8 MEMBER

MEMBER — Operator

expression `member` *list*

`member` is an infix binary comparison operator that evaluates to `true` if *expression* is `equal` to a member of the `list` *list*.

Examples

```
if a member {a,b} then 1 else 0;
```

$\Rightarrow$ `1`

```
if 1 member(1,2,3) then a else b;
```

$\Rightarrow$ `a`

```
if 1 member(1.0,2) then a else b;
```

$\Rightarrow$ `b`

Comments

Logical operators can only be used in conditional statements such as `if`...`then`...`else` or `while`...`do`. *member* can also be used as a prefix operator. However, this use is not encouraged. Finally, `equal` (`=`) is used for the test within the list, so expressions must be of the same type to match.

6.9 NEQ

NEQ Operator

The operator `neq` is an infix binary comparison operator. It returns `true` if its two arguments are not `equal`.

expression `neq` *expression*

An inequality is satisfied between floating point numbers and integers that have the same value.

Examples

```
on rounded;

a := 4;                      ⇒   A := 4

b := 4.0;                    ⇒   B := 4.0

if a neq b then write "true" else write "false";

                             ⇒   false

if a neq 5 then write "true" else write "false";

                             ⇒   true
```

Comments

Comparison operators can only be used as conditions in conditional commands such as `if`...`then` and `repeat`...`until`. *neq* can also be used as a prefix operator. However, this use is not encouraged.

6.10 NOT

NOT Operator

The `not` operator returns `true` if its argument evaluates to `nil`, and `nil` if its argument is `true`.

`not`(*logicalexpression*)

Examples

```
if not numberp(a) then write "indeterminate" else write a;
                              ⇒   indeterminate;
a := 10;                      ⇒   A := 10
if not numberp(a) then write "indeterminate" else write a;
                              ⇒   10
if not(numberp(a) and a < 0) then write "positive number";
                              ⇒   positive number
```

Comments

Logical operators can only be used in conditional statements such as `if`...`then`...`else` or `while`...`do`.

6.11 NUMBERP

NUMBERP Operator

The `numberp` operator returns `true` if its argument is a number, and `nil` otherwise.

`numberp`(*expression*) or `numberp` *expression*

expression can be any REDUCE scalar expression.

Examples

```
cc := 15.3;                ⇒   CC := 15.3

if numberp(cc) then write "number" else write "nonnumber";

                           ⇒   number

if numberp(cb) then write "number" else write "nonnumber";

                           ⇒   nonnumber
```

Comments

Logical operators can only be used in conditional expressions, such as `if`...`then`...`else` and `while`...`do`.

6.12 ORDP

ORDP Operator

The `ordp` logical operator returns `true` if its first argument is ordered ahead of its second argument in canonical internal ordering, or is identical to it.

`ordp`(*expression1*, *expression2*)

expression1 and *expression2* can be any valid REDUCE scalar expression.

Examples

```
if ordp(x**2 + 1,x**3 + 3) then write "yes" else write "no";
```

⇒ no

```
if ordp(101,100) then write "yes" else write "no";
```

⇒ yes

```
if ordp(x,x) then write "yes" else write "no";
```

⇒ yes

Comments

Logical operators can only be used in conditional expressions, such as `if`...`then`...`else` and `while`...`do`.

6.13 PRIMEP

PRIMEP Operator

`primep`(*expression*) or `primep` *simple_expression*

If *expression* evaluates to a integer, `primep` returns `true` if *expression* is a prime number (i.e., a number other than 0 and plus or minus 1 which is only exactly divisible by itself or a unit) and `nil` otherwise. If *expression* does not have an integer value, a type error occurs.

Examples

```
if primep 3 then write "yes" else write "no";
```

$\Rightarrow$ `YES`

`if primep a then 1;` $\Rightarrow$ `***** A invalid as integer`

6.14 TRUE

TRUE — Concept

Any value of the boolean part of a logical expression which is neither `nil` nor `0` is considered as `true`. Most builtin test and compare functions return `t` for `true` and `nil` for `false`.

Examples

```
if member(3,{1,2,3}) then 1 else -1;
                         ⇒ 1
if floor(1.7) then 1 else -1;
                         ⇒  1
if floor(0.7) then 1 else -1;
                         ⇒  -1
```

7 General Commands

7.1 BYE

BYE Command

The `bye` command ends the REDUCE session, returning control to the program (e.g., the operating system) that called REDUCE. When you are at the top level, the `bye` command exits REDUCE. `quit` is a synonym for `bye`.

7.2 CONT

CONT   Command

The command `cont` returns control to an interactive file after a `pause` command that has been answered with `n`.

Examples

Suppose you are in the middle of an interactive file.

```
                               ⇒  factorize(x**2 + 17*x + 60);
                               ⇒  {{X + 12,1},{X + 5,1}}
pause;                         ⇒  Cont? (Y or N)
n
saveas results;
factor1 := first results;      ⇒  FACTOR1 := {X + 12,1}
factor2 := second results;     ⇒  FACTOR2 := {X + 5,1}
cont;                          ⇒
       the file resumes
```

Comments

A `pause` allows you to enter your own REDUCE commands, change switch values, inquire about results, or other such activities. When you wish to resume operation of the interactive file, use `cont`.

7.3 DISPLAY

DISPLAY — Command

When given a numeric argument n, `display` prints the n most recent input statements, identified by prompt numbers. If an empty pair of parentheses is given, or if n is greater than the current number of statements, all the input statements since the beginning of the session are printed.

`display`(n) *or* `display`()

n should be a positive integer. However, if it is a real number, the truncated integer value is used, and if a non-numeric argument is used, all the input statements are printed.

Comments

The statements are displayed in upper case, with lines split at semicolons or dollar signs, as they are in editing. If long files have been input during the session, the `display` command is slow to format these for printing.

7.4 LOAD_PACKAGE

LOAD_PACKAGE — Command

The `load_package` command is used to load REDUCE packages, such as `gentran` that are not automatically loaded by the system.

`load_package "`*package_name*`"`

A package is only loaded once; subsequent calls of `load_package` for the same package name are ignored.

7.5 PAUSE

PAUSE Command

The `pause` command, given in an interactive file, stops operation and asks if you want to continue or not.

Examples

An interactive file is running, and at some point you see the question

```
Cont? (Y or N)
```

If you type

`y` Return

the file continues to run until the next pause or the end.

If you type

`n` Return

you will get a numbered REDUCE prompt, and be allowed to enter and execute any REDUCE statements. If you later wish to continue with the file, type

```
cont;
```

and the file resumes.

To use `pause` in your own interactive files, type

```
pause;
```

in the file wherever you want it.

Comments

`pause` does not allow you to continue without typing either `y` or `n`. Its use is to slow down scrolling of interactive files, or to let you change parameters or switch settings for the calculations.

If you have stopped an interactive file at a `pause,` and do not wish to resume the file, type `end;`. This does not end the REDUCE session, but stops input from the file. A second `end;` ends the REDUCE session. However, if you have pauses from more than one file stacked up, an `end;` brings you back to the top level, not the file directly above.

A `pause` typed from the terminal has no effect.

7.6 QUIT

QUIT Command

The `quit` command ends the REDUCE session, returning control to the program (e.g., the operating system) that called REDUCE. When you are at the top level, the `quit` command exits REDUCE. `bye` is a synonym for `quit`.

7.7 RECLAIM

RECLAIM Operator

Comments

REDUCE's memory is in a storage structure called a heap. As REDUCE statements execute, chunks of memory are used up. When these chunks are no longer needed, they remain idle. When the memory is almost full, the system executes a garbage collection, reclaiming space that is no longer needed, and putting all the free space at one end. Depending on the size of the image REDUCE is using, garbage collection needs to be done more or less often. A larger image means fewer but longer garbage collections. Regardless of memory size, if you ask REDUCE to do something ridiculous, like `factorial(2000)`, it may garbage collect many times.

7.8 REDERR

REDERR Command

The `rederr` command allows you to print an error message from inside a `procedure` or a `block` statement. The calculation is gracefully terminated.

`rederr` *message*

message is an error message, usually inside double quotation marks (a `string`).

Examples

```
procedure fac(n);
   if not (fixp(n) and n>=0)
     then  rederr "Choose nonneg. integer only"
    else for i := 0:n-1 product i+1;
```

	⇒	`fac`
`fac a;`	⇒	`***** Choose nonneg. integer only`
`fac 5;`	⇒	`120`

Comments

The above procedure finds the factorial of its argument. If n is not a positive integer or 0, an error message is returned.

If your procedure is executed in a file, the usual error message is printed, followed by `Cont?  (Y or N)`, just as any other error does from a file. Although the procedure is gracefully terminated, any switch settings or variable assignments you made before the error occurred are not undone. If you need to clean up such items before exiting, use a group statement, with the `rederr` command as its last statement.

7.9 RETRY

RETRY — Command

The `retry` command allows you to retry the latest statement that resulted in an error message.

Examples

```
matrix a;

det a;                   ⇒   ***** Matrix A not set

a := mat((1,2),(3,4));   ⇒   A(1,1) := 1
                             A(1,2) := 2
                             A(2,1) := 3
                             A(2,2) := 4
retry;                   ⇒   -2
```

Comments

`retry` remembers only the most recent statement that resulted in an error message. It allows you to stop and fix something obvious, then continue on your way without retyping the original command.

7.10 SAVEAS

SAVEAS Command

The `saveas` command saves the current workspace under the name of its argument.

`saveas` *identifier*

identifier can be any valid REDUCE identifier.

Examples

(The numbered prompts are shown below, unlike in most examples)

```
1: solve(x^2-3);    ⇒   {x=sqrt(3),x= - sqrt(3)}

2: saveas rts(0)$

3: rts(0);          ⇒   {x=sqrt(3),x= - sqrt(3)}
```

Comments

`saveas` works only for the current workspace, the last algebraic expression produced by REDUCE. This allows you to save a result that you did not assign to an identifier when you originally typed the input. For access to previous output use `ws`.

7.11 SHOWTIME

SHOWTIME Command

The `showtime` command prints the elapsed system time since the last call of this command or since the beginning of the session, if it has not been called before.

Examples

```
showtime;                ⇒   Time: 1020 ms

factorize(x^4 - 8x^4 + 8x^2 - 136x - 153);
                                       2
                         ⇒   {X - 9,X   + 17,X + 1}

showtime;                ⇒   Time: 920 ms
```

Comments

The time printed is either the elapsed cpu time or the elapsed wall clock time, depending on your system. `showtime` allows you to see the system time resources REDUCE uses in its calculations. Your time readings will of course vary from this example according to the system you use.

7.12 WRITE

WRITE — Command

The `write` command explicitly writes its arguments to the output device (terminal or file).

`write` *item*{,*item*}∗

item can be an expression, an assignment or a `string` enclosed in double quotation marks (").

Examples

```
write a, sin x, "this is a string";
                         ⇒   ASIN(X)this is a string

write a," ",sin x," this is a string";
                         ⇒   A SIN(X) this is a string

if not numberp(a) then write "the symbol ",a;
                         ⇒   the symbol A

array m(10);

for i := 1:5 do write m(i) := 2*i;
                         ⇒   M(1) := 2
                             M(2) := 4
                             M(3) := 6
                             M(4) := 8
                             M(5) := 10
m(4);                    ⇒   8
```

Comments

The items specified by a single `write` statement print on a single line unless they are too long. A printed line is always ended with a carriage return, so the next item printed starts a new line.

When an assignment statement is printed, the assignment is also made. This allows you to get feedback on filling slots in an array with a `for` statement, as shown in the last example above.

8 Algebraic Operators

8.1 APPEND

APPEND — Operator

The `append` operator constructs a new `list` from the elements of its two arguments (which must be lists).

append(*list*, *list*)

list must be a list, though it may be the empty list ({}). Any arguments beyond the first two are ignored.

Examples

```
alist := {1,2,{a,b}};       ⇒  ALIST := {1,2,{A,B}}
blist := {3,4,5,sin(y)};    ⇒  BLIST := {3,4,5,SIN(Y)}
append(alist,blist);        ⇒  {1,2,{A,B},3,4,5,SIN(Y)}
append(alist,{});           ⇒  {1,2,{A,B}}
append(list z,blist);       ⇒  {Z,3,4,5,SIN(Y)}
```

Comments

The new list consists of the elements of the second list appended to the elements of the first list. You can `append` new elements to the beginning or end of an existing list by putting the new element in a list (use curly braces or the operator `list`). This is particularly helpful in an iterative loop.

8.2 ARBINT

ARBINT Operator

The operator `arbint` is used to express arbitrary integer parts of an expression, e.g. in the result of `solve` when `allbranch` is on.

Examples

`solve(log(sin(x+3)),x);` $\Rightarrow$

```
{X=2*ARBINT(1)*PI - ASIN(1) - 3,
 X=2*ARBINT(1)*PI + ASIN(1) + PI - 3}
```

8.3 ARBCOMPLEX

ARBCOMPLEX Operator

The operator `arbcomplex` is used to express arbitrary scalar parts of an expression, e.g. in the result of `solve` when the solution is parametric in one of the variable.

Examples

```
solve({x+3=y-2z,y-3x=0},{x,y,z});
```

⇒

```
     2*ARBCOMPLEX(1) + 3
{X=---------------------,
              2

     3*ARBCOMPLEX(1) + 3
 Y=---------------------,
              2

 Z=ARBCOMPLEX(1)}
```

8.4 ARGLENGTH

ARGLENGTH — Operator

The operator `arglength` returns the number of arguments of the top-level operator in its argument.

`arglength`(*expression*)

expression can be any valid REDUCE algebraic expression.

Examples

`arglength(a + b + c + d);`	$\Rightarrow$	`4`
`arglength(a/b/c);`	$\Rightarrow$	`2`
`arglength(log(sin(df(r**3*x,x))));`		
	$\Rightarrow$	`1`

Comments

In the first example, `+` is an n-ary operator, so the number of terms is returned. In the second example, since `/` is a binary operator, the argument is actually (a/b)/c, so there are two terms at the top level. In the last example, no matter how deeply the operators are nested, there is still only one argument at the top level.

8.5 COEFF

COEFF — Operator

The `coeff` operator returns the coefficients of the powers of the specified variable in the given expression, in a `list`.

`coeff`(*expression*, *variable*)

expression is expected to be a polynomial expression, not a rational expression. Rational expressions are accepted when the switch `ratarg` is on. *variable* must be a kernel. The results are returned in a list.

Examples

```
                                    3      2
coeff((x+y)**3,x);           =>  {Y  ,3*Y  ,3*Y,1}

coeff((x+2)**4 + sin(x),x);  =>  {SIN(X) + 16,32,24,8,1}

high_pow;                    =>  4

low_pow;                     =>  0

ab := x**9 + sin(x)*x**7 + sqrt(y);
                                                        7    9
                             =>  AB := SQRT(Y) + SIN(X)*X  + X

coeff(ab,x);                 =>  {SQRT(Y),0,0,0,0,0,0,SIN(X),0,1}
```

Comments

The variables `high_pow` and `low_pow` are set to the highest and lowest powers of the variable, respectively, appearing in the expression.

The coefficients are put into a list, with the coefficient of the lowest (constant) term first. You can use the usual list access methods (`first`, `second`, `third`, `rest`, `length`, and `part`) to extract them. If a power does not appear in the expression, the corresponding element of the list is zero. Terms involving functions of the specified variable but not including powers of it (for example in the expression `x**4 + 3*x**2 + tan(x)`) are placed in the constant term.

Since the `coeff` command deals with the expanded form of the expression, you may get unexpected results when `exp` is off, or when `factor` or `ifactor` are on.

If you want only a specific coefficient rather than all of them, use the `coeffn` operator.

8.6 COEFFN

COEFFN Operator

The `coeffn` operator takes three arguments: an expression, a kernel, and a non-negative integer. It returns the coefficient of the kernel to that integer power, appearing in the expression.

`coeffn`(*expression*, *kernel*, *integer*)

expression must be a polynomial, unless `ratarg` is on which allows rational expressions. *kernel* must be a kernel, and *integer* must be a non-negative integer.

Examples

```
ff := x**7 + sin(y)*x**5 + y**4 + x + 7;
                                          5    7          4
                            =>    FF := SIN(Y)*X  + X  + X + Y  + 7

coeffn(ff,x,5);             =>    SIN(Y)

coeffn(ff,z,3);             =>    0
                                        5    7
coeffn(ff,y,0);             =>    SIN(Y)*X  + X  + X + 7
                                              2    5
                                         SIN(Y)*Y  + Y  + 1
rr := 1/y**2+y**3+sin(y);   =>    RR := --------------------
                                                2
                                               Y

on ratarg;

coeffn(rr,y,-2);            =>    ***** -2 invalid as COEFFN index
                                   1
coeffn(rr,y,5);             =>    ----
                                    2
                                   Y
```

Comments

If the given power of the kernel does not appear in the expression, `coeffn` returns 0. Negative powers are never detected, even if they appear in the expression and `ratarg` are on. `coeffn` with an integer argument of 0 returns any terms in the expression that do *not* contain the given kernel.

8.7 CONJ

CONJ Operator

`conj`(*expression*) or `conj` *simple_expression*

This operator 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`.

Examples

```
conj(1+i);     ⇒   1-I

conj(a+i*b);   ⇒

       REPART(A) - REPART(B)*I - IMPART(A)*I - IMPART(B)
```

8.8 CONTINUED_FRACTION

CONTINUED_FRACTION Operator

`continued_fraction`(num) or `continued_fraction`($num, size$)

This operator approximates the real number *num* (`rational` number, `rounded` number) into a continued fraction. The result is a list of two elements: the first one is the rational value of the approximation, the second one is the list of terms of the continued fraction which represents the same value according to the definition `t0 +1/(t1 + 1/(t2 + ...))`. Precision: the second optional parameter *size* is an upper bound for the absolute value of the result denominator. If omitted, the approximation is performed up to the current system precision.

Examples

```
continued_fraction pi;       ⇒
          1146408
        {---------,{3,7,15,1,292,1,1,1,2,1}}
          364913
                                    22
continued_fraction(pi,100);  ⇒   {----,{3,7}}
                                    7
```

8.9 DECOMPOSE

DECOMPOSE Operator

The `decompose` operator takes a multivariate polynomial as argument, and returns an expression and a `list` of `equation`s from which the original polynomial can be found by composition.

`decompose`(*expression*) or `decompose` *simple_expression*

Examples

```
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
```

Comments

Unlike factorization, this decomposition is not unique. Further details can be found in V.S. Alagar, M.Tanh, *Fast Polynomial Decomposition*, Proc. EUROCAL 1985, pp 150-153 (Springer) and J. von zur Gathen, *Functional Decomposition of Polynomials: the Tame Case*, J. Symbolic Computation (1990) 9, 281-299.

8.10 DEG

DEG Operator

The operator `deg` returns the highest degree of its variable argument found in its expression argument.

`deg`(*expression*, *kernel*)

expression is expected to be a polynomial expression, not a rational expression. Rational expressions are accepted when the switch `ratarg` is on. *variable* must be a `kernel`. The results are returned in a list.

Examples

```
deg((x+y)**5,x);              ⇒    5

deg((a+b)*(c+2*d)**2,d);      ⇒    2

deg(x**2 + cos(y),sin(x));

deg((x**2 + sin(x))**5,sin(x));

                              ⇒    5
```

8.11 DEN

DEN Operator

The `den` operator returns the denominator of its argument.

`den`(*expression*)

expression is ordinarily a rational expression, but may be any valid scalar REDUCE expression.

Examples

```
                                            2
a := x**3 + 3*x**2 + 12*x;  ⇒  A := X*(X  + 3*X + 12)

b := 4*x*y + x*sin(x);      ⇒  B := X*(SIN(X) + 4*Y)

den(a/b);                   ⇒  SIN(X) + 4*Y

den(aa/4 + bb/5);           ⇒  20

den(100/6);                 ⇒  3

den(sin(x));                ⇒  1
```

Comments

`den` returns the denominator of the expression after it has been simplified by REDUCE. As seen in the examples, this includes putting sums of rational expressions over a common denominator, and reducing common factors where possible. If the expression does not have any other denominator, 1 is returned.

Switch settings, such as `mcd` or `rational`, have an effect on the denominator of an expression.

8.12 DF

DF Operator

The df operator finds partial derivatives with respect to one or more variables.

df(*expression*, *var*&*optional*(, *number*){, *var*&*option*(, *number*)}*)

expression can be any valid REDUCE algebraic expression. *var* must be a kernel, and is the differentiation variable. *number* must be a non-negative integer.

Examples

```
df(x**2,x);                  =>   2*X
                                            2
df(x**2*y + sin(y),y);       =>   COS(Y) + X
df((x+y)**10,z);             =>   0
                                   6
df(1/x**2,x,2);              =>   ----
                                    4
                                   X
df(x**4*y + sin(y),y,x,3);   =>   24*X
for all x let df(tan(x),x) = sec(x)**2;
                                            2
df(tan(3*x),x);              =>   3*SEC(3*X)
```

Comments

An error message results if a non-kernel is entered as a differentiation operator. If the optional number is omitted, it is assumed to be 1. See the declaration depend to establish dependencies for implicit differentiation.

You can define your own differentiation rules, expanding REDUCE's capabilities, using the let command as shown in the last example above. Note that once you add your own rule for differentiating a function, it supersedes REDUCE's normal handling of that function for the duration of the REDUCE session. If you clear the rule (clearrules), you don't get back to the previous rule.

8.13 EXPAND_CASES

EXPAND_CASES Operator

When a `root_of` form in a result of `solve` has been converted to a `one_of` form, `expand_cases` can be used to convert this into form corresponding to the normal explicit results of `solve`. See `root_of`.

8.14 EXPREAD

EXPREAD Operator

`expread()`

`expread` reads one well-formed expression from the current input buffer and returns its value.

Examples

`expread(); a+b;` ⇒ `A + B`

8.15 FACTORIZE

FACTORIZE Operator

The `factorize` operator factors a given expression into a list of {factor,power} pairs.

`factorize`(*expression*)

expression should be a polynomial, otherwise an error will result.

Examples

```
fff := factorize(x^3 - y^3);
                                 2          2
                          ⇒   {{X  + X*Y + Y ,1},{X - Y,1}}
                                        2          2
fac1 := first fff;        ⇒   FAC1 := {{X  + X*Y + Y ,1}

factorize(x^15 - 1);      ⇒
             8    7    6    5    4
        {{ X  - X  + X  - X  + X  - X + 1,1},

           4    3    2
        {X  + X  + X  + X + 1,1},

           2
        {X  + X + 1,1},
        {X - 1,1}}
lastone := part(ws,length ws);

                          ⇒   LASTONE := {X - 1,1}

setmod 2;                 ⇒   1

on modular;
```

```
factorize(x^15 - 1);     =>   {{X^4 + X^3 + X^2 + X + 1,1},
                               {X^4 + X^3 + 1,1},
                               {X^4 + X + 1,1},
                               { X^2 + X + 1,1},
                               {X + 1,1}}
```

Comments

The `factorize` command returns the factor,power pairs as a `list`. You can therefore use the usual list access methods (`first`, `second`, `third`, `rest`, `length` and `part`) to extract these pairs.

If the *expression* given to `factorize` is an integer, it will be factored into its prime components. To factor any integer factor of a non-numerical expression, the switch `ifactor` should be turned on. Its default is off. `ifactor` has effect only when factoring is explicitly done by `factorize`, not when factoring is automatically done with the `factor` switch. If full factorization is not needed the switch `limitedfactors` allows you to reduce the computing time of calls to `factorize`.

Factoring can be done in a modular domain by calling `factorize` when `modular` is on. You can set the modulus with the `setmod` command. The last example above shows factoring modulo 2.

For general comments on factoring, see comments under the switch `factor`.

8.16 HYPOT

HYPOT **Operator**

hypot(*expression*,*expression*)

If `rounded` is on, and the two arguments evaluate to numbers, this operator returns the square root of the sums of the squares of the arguments in a manner that avoids intermediate overflow. In other cases, an expression in the original operator is returned.

Examples

```
hypot(3,4);   ⇒   HYPOT(3,4)

on rounded;

ws;           ⇒   5.0

hypot(a,b);   ⇒   HYPOT(A,B)
```

8.17 IMPART

IMPART Operator

`impart`(*expression*) or `impart` *simple_expression*

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`.

Examples

```
impart(1+i);      ⇒   1

impart(a+i*b);    ⇒   REPART(B) + IMPART(A)
```

8.18 INT

INT — Operator

The `int` operator performs analytic integration on a variety of functions.

`int`(*expression*, *kernel*)

expression can be any scalar expression. involving polynomials, log functions, exponential functions, or tangent or arctangent expressions. `int` attempts expressions involving error functions, dilogarithms and other trigonometric expressions. Integrals involving algebraic extensions (such as square roots) may not succeed. *kernel* must be a REDUCE `kernel`.

Examples

```
                                  3
                               X*(X  + 12)
int(x**3 + 3,x);          =>  -------------
                                   4


                                 2*X
                                E   *(COS(X) - 2*SIN(X))
int(sin(x)*exp(2*x),x);   =>  - --------------------------
                                          5
int(1/(x^2-2),x);         =>
        SQRT(2)*(LOG( - SQRT(2) + X) - LOG(SQRT(2) + X))
       --------------------------------------------------
                              4
int(sin(x)/(4 + cos(x)**2),x);
                                      COS(X)
                                ATAN(--------)
                                        2
                          =>  - ----------------
                                       2

                                   SQRT(X)*SQRT(X - 1)
int(1/sqrt(x^2-x),x);     =>  INT(---------------------,X)
                                          2
                                         X -X
```

Comments

Note that REDUCE couldn't handle the last integral with its default integrator, since the integrand involves a square root. However, the integral can be found using the `algint` package. Alternatively, you could add a rule using the `let` statement to evaluate this integral.

The arbitrary constant of integration is not shown. Definite integrals can be found by evaluating the result at the limits of integration (use `rounded`) and subtracting the lower from the higher. Evaluation can be easily done by the `sub` operator.

When `int` cannot find an integral it returns an expression involving formal `int` expressions unless the switch `failhard` has been set. If not all of the expression can be integrated, the switch `nolnr` controls whether a partially integrated result should be returned or not.

8.19 INTERPOL

INTERPOL Operator

`interpol` generates an interpolation polynomial.

interpol(*values*,*variable*,*points*)

values and *points* are `list`s of equal length and *variable* is an algebraic expression (preferably a `kernel`). The interpolation polynomial is generated 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.

Examples

```
f := for i:=1:4 collect(i**3-1);

                         ⇒   F := 0,7,26,63

p := {1,2,3,4};          ⇒   P := 1,2,3,4
                               3
interpol(f,x,p);         ⇒   X  - 1
```

Comments

The Aitken-Neville interpolation algorithm is used which guarantees a stable result even with rounded numbers and an ill-conditioned problem.

8.20 LCOF

LCOF Operator

The `lcof` operator returns the leading coefficient of a given expression with respect to a given variable.

`lcof`(*expression*, *kernel*)

expression is ordinarily a polynomial. If **`ratarg`** is on, a rational expression may also be used, otherwise an error results. *kernel* must be a **`kernel`**.

Examples

```
lcof((x+2*y)**5,y);        =>   32

lcof((x + y*sin(x))**2 + cos(x)*sin(x)**2,sin(x));
                                       2
                           =>   COS(X)  + Y
                                 2
lcof(x**2 + 3*x + 17,y);   =>   X  + 3*X + 17
```

Comments

If the kernel does not appear in the expression, `lcof` returns the expression.

8.21 LENGTH

LENGTH Operator

The `length` operator returns the number of items in a `list`, the number of terms in an expression, or the dimensions of an array or matrix.

`length(`*expr*`)` or `length` *expr*

expr can be a list structure, an array, a matrix, or a scalar expression.

Examples

```
alist := {a,b,{ww,xx,yy,zz}};
                          ⇒   ALIST := {A,B,{WW,XX,YY,ZZ}}
length alist;             ⇒   3
length third alist;       ⇒   4
dlist := {d};             ⇒   DLIST := {D}
length rest dlist;        ⇒   0
matrix mmm(4,5);
length mmm;               ⇒   {4,5}
array aaa(5,3,2);
length aaa;               ⇒   {6,4,3}
                                     2
                                    X  + 6*X + 9
eex := (x+3)**2/(x-y);    ⇒   EEX := --------------
                                        X - Y
length eex;               ⇒   5
```

Comments

An item in a list that is itself a list only counts as one item. An error message will be printed if `length` is called on a matrix which has not had its dimensions set. The `length` of an array includes the zeroth element of each dimension, showing the full number of elements allocated. (Declaring an array A with n elements allocates $A(0), A(1), \ldots, A(n)$.) The `length` of an expression is the total number of additive terms appearing in the numerator and denominator of the expression. Note that subtraction of a term is represented internally as addition of a negative term.

8.22 LHS

LHS Operator

The `lhs` operator returns the left-hand side of an `equation`, such as those returned in a list by `solve`.

`lhs`(*equation*) or `lhs` *equation*

equation must be an equation of the form
`left-hand side` = `right-hand side`.

Examples

```
                                                5       4       2
polly := (x+3)*(x^4+2x+1);  ⇒   POLLY := X  + 3*X  + 2*X  + 7*X + 3

pollyroots := solve(polly,x);

                            ⇒
                                        3      2
       POLLYROOTS := {X=ROOT_OF(X_  - X_  + X_ + 1,X_),
                     X=-1,
                     X=-3}
variable := lhs first pollyroots;

                            ⇒   VARIABLE := X
```

8.23 LIMIT

LIMIT — Operator

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'Hopital's rule is used in critical cases, with preprocessing of 1-1 forms and reformatting of product forms in order to apply l'Hopital's rule. A limited amount of bounded arithmetic is also employed where applicable.

`limit`(*expr*, *var*, *limpoint*) or
`limit!+`(*expr*, *var*, *limpoint*) or
`limit!-`(*expr*, *var*, *limpoint*)

where *expr* is an expression depending of the variable *var* (a `kernel`) and *limpoint* is the limit point. If the limit depends upon the direction of approach to the *limpoint*, the operators `limit!+` and `limit!-` may be used.

Examples

```
limit(x*cot(x),x,0);     ⇒  0

limit((2x+5)/(3x-2),x,infinity);
                                 2
                         ⇒  ---
                                 3
```

8.24 LPOWER

LPOWER — Operator

The `lpower` operator returns the leading power of an expression with respect to a kernel. 1 is returned if the expression does not depend on the kernel.

`lpower`(*expression*, *kernel*)

expression is ordinarily a polynomial. If `ratarg` is on, a rational expression may also be used, otherwise an error results. *kernel* must be a `kernel`.

Examples

```
                                 6
lpower((x+2*y)**6,y);    ⇒    Y

lpower((x + cos(x))**8 + df(x**2,x),cos(x));
                                      8
                         ⇒    COS(X)

lpower(x**3 + 3*x,y);    ⇒    1
```

8.25 LTERM

LTERM Operator

The `lterm` operator returns the leading term of an expression with respect to a kernel. The expression is returned if it does not depend on the kernel.

`lterm`(*expression*, *kernel*)

expression is ordinarily a polynomial. If `ratarg` is on, a rational expression may also be used, otherwise an error results. *kernel* must be a `kernel`.

Examples

```
                                    6
lterm((x+2*y)**6,y);     ⇒   64*Y

lterm((x + cos(x))**8 + df(x**2,x),cos(x));
                                      8
                         ⇒   COS(X)
                               3
lterm(x**3 + 3*x,y);     ⇒   X  + 3X
```

8.26 MAINVAR

MAINVAR Operator

The `mainvar` operator returns the main variable (in the system's internal representation) of its argument.

`mainvar`(*expression*)

expression is usually a polynomial, but may be any valid REDUCE scalar expression. In the case of a rational function, the main variable of the numerator is returned. The main variable returned is a `kernel`.

Examples

```
test := (a + b + c)**2;  ⇒

              2                     2                2
       TEST := A  + 2*A*B + 2*A*C + B  + 2*B*C + C

mainvar(test);           ⇒   A

korder c,b,a;

mainvar(test);           ⇒   C

mainvar(2*cos(x)**2);    ⇒   COS(X)

mainvar(17);             ⇒   0
```

Comments

The main variable is the first variable in the canonical ordering of kernels. Generally, alphabetically ordered functions come first, then alphabetically ordered identifiers (variables). Numbers come last, and as far as `mainvar` is concerned belong in the family `0`. The canonical ordering can be changed by the declaration `korder`, as shown above.

8.27 MAP

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.

`map`(*function*, *object*)

object is a list, a matrix or an operator expression.

function is the name of an operator for a single argument: the operator is evaluated once with each element of *object* as its single argument,

or 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,

or a replacement `rule` of the form

`var` =¿ `rep`

where *var* is a variable (a *kernel* without subscript) and *rep* is an expression which contains *var*. Here `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     3
⇒     2*x =3*(y -2)
```

Comments

You can use `map` in nested expressions. It is not allowed to apply `map` for a non-composed object, e.g. an identifier or a number.

8.28 MKID

MKID Command

The `mkid` command constructs an identifier, given a stem and an identifier or an integer.

`mkid`(*stem*, *leaf*)

stem can be any valid REDUCE identifier that does not include escaped special characters. *leaf* may be an integer, including one given by a local variable in a `for` loop, or any other legal group of characters.

Examples

```
mkid(x,3);              =>   X3

factorize(x^15 - 1);    =>   {X - 1,

                               2
                              X  + X + 1,

                               4    3    2
                              X  + X  + X  + X + 1,

                               8    7    5    4    3
                              X  - X  + X  - X  + X   - X + 1}
for i := 1:length ws do write set(mkid(f,i),part(ws,i));
                              8    7    5    4    3
                        =>   X  - X  + X  - X  + X  - X + 1

                              4    3    2
                             X  + X  + X  + X + 1

                              2
                             X  + X + 1
                             X - 1
```

Comments

You can use `mkid` to construct identifiers from inside procedures. This allows you to handle an unknown number of factors, or deal with variable amounts of data. It is particularly helpful to attach identifiers to the answers returned by `factorize` and `solve`.

8.29 NPRIMITIVE

NPRIMITIVE — Operator

`nprimitive`(*expression*) or `nprimitive` *simple_expression*

This operator returns the numerically-primitive part of any scalar expression. In other words, any overall integer factors in the expression are removed.

Examples

```
                            2            2
nprimitive((2x+2y)^2);  ⇒  X  + 2*X*Y + Y

nprimitive(3*a*b*c);    ⇒  3*A*B*C
```

8.30 NUM

NUM Operator

The `num` operator returns the numerator of its argument.

`num`(*expression*) or `num` *simple_expression*

expression can be any valid REDUCE scalar expression.

Examples

`num(100/6);`	⇒	`50`
`num(a/5 + b/6);`	⇒	`6*A + 5*B`
`num(sin(x));`	⇒	`SIN(X)`

Comments

`num` returns the numerator of the expression after it has been simplified by REDUCE. As seen in the examples, this includes putting sums of rational expressions over a common denominator, and reducing common factors where possible. If the expression is not a rational expression, it is returned unchanged.

8.31 ODESOLVE

ODESOLVE Operator

The `odesolve` package is a solver for ordinary differential equations. At the present time it has still 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.

`odesolve`(*expr*, *var1*, *var2*)

expr is a single scalar expression such that *expr*=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.

A differential in *expr* is expressed using the `df` operator. Note that in most cases you must declare explicitly *var1* to depend of *var2* using a `depend` declaration – otherwise the derivative might be evaluated to zero on input to `odesolve`.

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. A counter `!!arbconst` is used to arrange this.

Examples

```
depend y,x;

% A first-order linear equation, with an initial condition

ode:=df(y,x) + y * sin x/cos x - 1/cos x

odesolve(ode,y,x);      ⇒   {y=arbconst(1)*cos(x) + sin(x)}
```

8.32 ONE_OF

ONE_OF — Type

The operator `one_of` is used to represent an indefinite choice of one element from a finite set of objects.

Examples

```
x=one_of{1,2,5}
```

this equation encodes that x can take one of the values 1,2 or 5

REDUCE generates a `one_of` form in cases when an implicit `root_of` expression could be converted to an explicit solution set. A `one_of` form can be converted to a `solve` solution using `expand_cases`. See `root_of`.

8.33 PART

PART — Operator

The operator `part` permits the extraction of various parts or operators of expressions and `lists`.

`part`(*expression*, *integer*{, *integer*}∗)

expression can be any valid REDUCE expression or a list, *integer* may be an expression that evaluates to a positive or negative integer or 0. A positive integer n picks up the n th term, counting from the first term toward the end. A negative integer n picks up the n th term, counting from the back toward the front. The integer 0 picks up the operator (which is `LIST` when the expression is a **??**).

Examples

```
                                   2  3
part((x + y)**5,4);       ⇒   10*X *Y

                                2
part((x + y)**5,4,2);     ⇒   X

part((x + y)**5,4,2,1);   ⇒   X

part((x + y)**5,0);       ⇒   PLUS

                                     4
part((x + y)**5,-5);      ⇒   5*X *Y

part((x + y)**5,4) := sin(x);

                          ⇒

        5       4           3   2                      4     5
       X  + 5*X *Y + 10*X *Y  + SIN(X) + 5*X*Y  + Y

alist := {x,y,{aa,bb,cc},x**2*sqrt(y)};

                          ⇒

                                             2
       ALIST := {X,Y,{AA,BB,CC},SQRT(Y)*X }

part(alist,3,2);          ⇒   BB

part(alist,4,0);          ⇒   TIMES
```

Comments

Additional integer arguments after the first one examine the terms recursively, as shown above. In the third line, the fourth term is picked from the original polynomial, $10x^2y^3$, then the second term from that, x^2, and finally the first component, x. If an integer's absolute value is too large for the appropriate expression, a message is given.

`part` works on the form of the expression as printed, or as it would have been printed at that point of the calculation, bearing in mind the current switch settings. It is important to realize that the switch settings change the operation of `part`. `pri` must be on when `part` is used.

When `part` is used on a polynomial expression that has minus signs, the `+` is always returned as the top-level operator. The minus is found as a unary operator attached to the negative term.

`part` can also be used to change the relevant part of the expression or list as shown in the sixth example line. The `part` operator returns the changed expression, though original expression is not changed. You can also use `part` to change the operator.

8.34 PF

PF Operator

pf(*expression*,*variable*)

`pf` transforms *expression* into a `list` of partial fraction s with respect to the main variable, *variable*. `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.

Examples

```
pf(2/((x+1)^2*(x+2)),x);  ⇒    2       -2            2
                              {-------,-------,--------------}
                               X + 2   X + 1    2
                                               X  + 2*X + 1
off exp;
pf(2/((x+1)^2*(x+2)),x);  ⇒    2       - 2          2
                              {-------,-------,----------}
                               X + 2   X + 1          2
                                               (X + 1)
for each j in ws sum j;   ⇒           2
                              ----------------
                                              2
                               ( + 2)*(X + 1)
```

Comments

If you want the denominators in factored form, turn `exp` off, as shown in the second example above. As shown in the final example, the `for each` construct can be used to recombine the terms. Alternatively, one can use the operations on lists to extract any desired term.

8.35 PROD

PROD Operator

The operator `prod` returns the indefinite or definite product of a given expression.

`prod`(*expr*, k[, *lolim*[, *uplim*]])

where *expr* is the expression to be multiplied, k is the control variable (a `kernel`), and *lolim* and *uplim* uplim are the optional lower and upper limits. If *uplim* is not supplied the upper limit is taken as k. The Gosper algorithm is used. If there is no closed form solution, the operator returns the input unchanged.

Examples

`prod(k/(k-2),k);` ⇒ `k*( - k + 1)`

8.36 REDUCT

REDUCT Operator

The `reduct` operator returns the remainder of its expression after the leading term with respect to the kernel in the second argument is removed.

`reduct`(*expression*, *kernel*)

expression is ordinarily a polynomial. If `ratarg` is on, a rational expression may also be used, otherwise an error results. *kernel* must be a `kernel`.

Examples

```
                                  2             2
reduct((x+y)**3,x);        ⇒   Y*(3*X  + 3*X*Y + Y )

reduct(x + sin(x)**3,sin(x));

                           ⇒   X

reduct(x + sin(x)**3,y);   ⇒   0
```

Comments

If the expression does not contain the kernel, `reduct` returns 0.

8.37 REPART

REPART Operator

`repart`(*expression*) or `repart` *simple_expression*

This operator 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`.

Examples

`repart(1+i);` $\Rightarrow$ `1`

`repart(a+i*b);` $\Rightarrow$ `REPART(A) - IMPART(B)`

8.38 RESULTANT

RESULTANT Operator

The `resultant` operator computes the resultant of two polynomials with respect to a given variable. If the resultant is 0, the polynomials have a root in common.

`resultant`(*expression*, *expression*, *kernel*)

expression must be a polynomial containing *kernel* ; *kernel* must be a `kernel`.

Examples

```
resultant(x**2 + 2*x + 1,x+1,x);
```

$\Rightarrow$ 0

```
resultant(x**2 + 2*x + 1,x-3,x);
```

$\Rightarrow$ 16

```
resultant(z**3 + z**2 + 5*z + 5,
          z**4 - 6*z**3 + 16*z**2 - 30*z + 55,
          z);
```

$\Rightarrow$ 0

```
resultant(x**3*y + 4*x*y + 10,y**2 + 6*y + 4,y);
```

$\Rightarrow$

```
  6        5         4         3         2
 Y  + 18*Y  + 120*Y  + 360*Y  + 480*Y  + 288*Y + 64
```

Comments

The resultant is the determinant of the Sylvester matrix, formed from the coefficients of the two polynomials in the following way:

Given two polynomials:

$$a_0x^n + a_1x^{n-1} + \cdots + a_n$$

and

$$b_0x^n + b_1x^{n-1} + \cdots + b_n$$

form the (m+n)x(m+n-1) Sylvester matrix by the following means:

$$\begin{pmatrix} 0 & \ldots & 0 & 0 & a_0 & a_1 & \ldots & a_n \\ 0 & \ldots & 0 & a_0 & a_1 & \ldots & a_n & 0 \\ \vdots & & & \vdots & & & \vdots & \\ a_0 & a_1 & \ldots & a_n & 0 & 0 & \ldots & 0 \\ 0 & \ldots & 0 & 0 & b_0 & b_1 & \ldots & b_n \\ \vdots & & & \vdots & & & \vdots & \\ b_0 & b_1 & \ldots & b_n & 0 & 0 & \ldots & 0 \end{pmatrix}$$

If the determinant of this matrix is 0, the two polynomials have a common root. Finding the resultant of large expressions is time-consuming, due to the time needed to find a large determinant.

The sign conventions `resultant` uses are those given in the article, "Computing in Algebraic Extensions," by R. Loos, appearing in *Computer Algebra–Symbolic and Algebraic Computation*, 2nd ed., edited by B. Buchberger, G.E. Collins and R. Loos, and published by Springer-Verlag, 1983. These are:

$$\begin{aligned} \text{resultant}(p(x), q(x), x) &= (-1)^{\deg p(x) * \deg q(x)} \cdot \text{resultant}(q(x), p(x), x), \\ \text{resultant}(a, p(x), x) &= a^{\deg p(x)}, \\ \text{resultant}(a, b, x) &= 1 \end{aligned}$$

where $p(x)$ and $q(x)$ are polynomials which have x as a variable, and a and b are free of x.

Error messages are given if `resultant` is given a non-polynomial expression, or a non-kernel variable.

8.39 RHS

RHS — Operator

The `rhs` operator returns the right-hand side of an `equation`, such as those returned in a `list` by `solve`.

`rhs`(*equation*) or `rhs` *equation*

equation must be an equation of the form *left-hand side* = *right-hand side*.

Examples

```
roots := solve(x**2 + 6*x*y + 5x + 3y**2,x);
```

⇒

```
                              2
               SQRT(24*Y  + 60*Y + 25) + 6*Y + 5
ROOTS := {X= - -----------------------------------,
                              2

                              2
               SQRT(24*Y  + 60*Y + 25) - 6*Y - 5
          X= -----------------------------------}
                              2
```

```
root1 := rhs first roots;
```

⇒

```
                   2
        SQRT(24*Y  + 60*Y + 25) + 6*Y + 5
ROOT1 := - -----------------------------------
                          2
```

```
root2 := rhs second roots;
```

⇒

```
                2
        SQRT(24*Y  + 60*Y + 25) - 6*Y - 5
ROOT2 := -----------------------------------
                        2
```

Comments

An error message is given if `rhs` is applied to something other than an equation.

8.40 ROOT_OF

ROOT_OF Operator

When the operator solve is unable to find an explicit solution or if that solution would be too complicated, the result is presented as formal root expression using the internal operator root_of and a new local variable. An expression with a top level root_of is implicitly a list with an unknown number of elements since we can'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 if the original solve operator expression into a standard solve solution. To effect this, the operator expand_cases can be used.

Examples

```
                                        7     2
solve(a*x^7-x^2+1,x);  ⇒  {x=root_of(a*x_  - x_  + 1,x_)}

sub(a=0,ws);           ⇒  {x=one_of(1,-1)}

expand_cases ws;       ⇒  x=1,x=-1
```

The components of root_of and one_of expressions can be processed as usual with operators arglength and part. A higher power of a root_of expression with a polynomial as first argument is simplified by using the polynomial as a side relation.

8.41 SELECT

SELECT Operator

The `select` operator extracts from a list or from the arguments of an n–ary operator elements corresponding to a boolean predicate. The predicate pattern can be a unary procedure, an operator or an algebraic expression with one `free variable`.

`select`(*function*, *object*)

object is a `list`.

function is the name of an operator for a single argument: the operator is evaluated once with each element of *object* as its single argument,

or 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,

or a replacement `rule` of the form

`var =¿ rep`

where *var* is a variable (a *kernel* without subscript) and *rep* is an expression which contains *var*. Here `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 evaluation result of *function* is interpreted as `boolean value` corresponding to the conventions of REDUCE. The result value is built with the leading operator of the input expression.

Examples

```
select( ~w>0 , {1,-1,2,-3,3})
                                   ⇒   {1,2,3}

q:=(part((x+y)^5,0):=list)
                                          5       3   2        4
select(evenp deg(~w,y),q);  ⇒   {x  ,10*x  *y  ,5*x*y  }

select(evenp deg(~w,x),2x^2+3x^3+4x^4);
```

$$\Rightarrow \quad 2x^2 + 4x^4$$

8.42 SHOWRULES

SHOWRULES Operator

`showrules`(*expression*) or `showrules` *simple_expression*

`showrules` returns in `rule-list` form any `operator` rules associated with its argument.

Examples

```
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*.

Comments

An operator may have properties that cannot be displayed in such a form, such as the fact it is an **??** function, or has a definition defined as a procedure.

8.43 SOLVE

SOLVE Operator

The `solve` operator solves a single algebraic `equation` or a system of simultaneous equations.

`solve`(*expression*&*option*(, *kernel*)) or
`solve`({*expression*&*option*(, *expression*) ∗ }&*option*(, {*kernel*) ∗ }})

If the number of equations equals the number of distinct kernels, the optional kernel argument(s) may be omitted. *expression* is either a scalar expression or an `equation`. When more than one expression is given, the `list` of expressions is surrounded by curly braces. The optional list of `kernel`s follows, also in curly braces.

Examples

```
sss := solve(x^2 + 7);   ⇒   Unknown: X
                             SSS := {X= - SQRT(7)*I,
                                     X=SQRT(7)*I}

rhs first sss;           ⇒   - SQRT(7)*I
                                 2*ARBINT(1)*PI
solve(sin(x^2*y),y);     ⇒   {Y=-----------------
                                       2
                                      X

                                 PI*(2*ARBINT(1) + 1)
                              Y=----------------------}
                                          2
                                         X

off allbranch;

solve(sin(x**2*y),y);    ⇒   {Y=0}

solve({3x + 5y = -4,2*x + y = -10},{x,y});
                                     22      46
                         ⇒   {{X= - ----,Y=----}}
                                     7       7

solve({x + a*y + z,2x + 5},{x,y});
                                     5        2*Z - 5
                         ⇒   {{X= - ---,Y= - ---------}}
                                     2         2*A
```

```
ab := (x+2)^2*(x^6 + 17x + 1);
```

⇒

```
             8       7       6        3        2
AB := X  + 4*X  + 4*X  + 17*X  + 69*X  + 72*X + 4
```

```
                                                6
www := solve(ab,x);       ⇒   {X=ROOT_OF(X_  + 17*X_+ 1),X=-2}
```

```
root_multiplicities;      ⇒   {1,2}
```

Comments

Results of the `solve` operator are returned as `equations` in a `list`. You can use the usual list access methods (`first`, `second`, `third`, `rest` and `part`) to extract the desired equation, and then use the operators `rhs` and `lhs` to access the right-hand or left-hand expression of the equation. When `solve` is unable to solve an equation, it returns the unsolved part as the argument of `root_of`, with the variable renamed to avoid confusion, as shown in the last example above.

For one equation, `solve` uses square-free factorization, roots of unity, and the known inverses of the `log`, `sin`, `cos`, `acos`, `asin`, and exponentiation operators. The quadratic, cubic and quartic formulas are used if necessary, but these are applied only when the switch `fullroots` is set on; otherwise or when no closed form is available the result is returned as `root_of` expression. The switch `trigform` determines which type of cubic and quartic formula is used. The multiplicity of each solution is given in a list as the system variable `root_multiplicities`. For systems of simultaneous linear equations, matrix inversion is used. For nonlinear systems, the Groebner basis method is used.

Linear equation system solving is influenced by the switch `cramer`.

Singular systems can be solved when the switch `solvesingular` is on, which is the default setting. An empty list is returned the system of equations is inconsistent. For a linear inconsistent system with parameters the variable `requirements` constraints conditions for the system to become consistent.

For a solvable linear and polynomial system with parameters the variable `assumptions` contains a list side relations for the parameters: the solution is valid only as long as none of these expressions is zero.

If the switch `varopt` is on (default), the system rearranges the variable sequence for minimal computation time. Without `varopt` the user supplied variable sequence is maintained.

If the solution has free variables (dimension of the solution is greater than zero), these are represented by `arbcomplex` expressions as long as the switch `arbvars` is on (default). Without `arbvars` no explicit equations are generated for free variables.

Related information

`allbranch` **switch**

`arbvars` **switch**

`assumptions` **variable**

`fullroots` **switch**

`requirements` **variable**

`roots` **operator**

`root_of` **operator**

`trigform` **switch**

`varopt` **switch**

8.44 SORT

SORT Operator

The `sort` operator sorts the elements of a list according to an arbitrary comparison operator.

`sort`(*lst*, *comp*)

lst is a `list` of algebraic expressions. *comp* is a comparison operator which defines a partial ordering among the members of *lst*. *comp* may be one of the builtin comparison operators like `<`(`lessp`), `<=`(`leq`) etc., or *comp* may be the name of a comparison procedure. Such a procedure has two arguments, and it returns `true` if the first argument ranges before the second one, and 0 or `nil` otherwise. The result of `sort` is a new list which contains the elements of *lst* in a sequence corresponding to *comp*.

Examples

```
procedure ce(a,b);

if evenp a and not evenp b then 1 else 0;

for i:=1:10 collect random(50)

sort(ws,>=);              ⇒   {41,38,33,30,28,25,20,17,8,5}

sort(ws,<);               ⇒   {5,8,17,20,25,28,30,33,38,41}

sort(ws,ce);              ⇒  {8,20,28,30,38,5,17,25,33,41}

procedure cd(a,b);

if deg(a,x)>deg(b,x) then 1 else

if deg(a,x)<deg(b,x) then 0 else

if deg(a,y)>deg(b,y) then 1 else 0;

                                2      2
sort({x^2,y^2,x*y},cd);  ⇒  {x ,x*y,y }
```

8.45 STRUCTR

STRUCTR Operator

The `structr` operator breaks its argument expression into named subexpressions.

`structr`(*expression*&*option*(, *identifier*&*option*(, *identifier*)))

`structr`(*expression*[, *identifier*[, *identifier*...]])

expression may be any valid REDUCE scalar expression. *identifier* may be any valid REDUCE `identifier`. The first identifier is the stem for subexpression names, the second is the name to be assigned to the structured expression.

Examples

```
structr(sqrt(x**2 + 2*x) + sin(x**2*z));

                          ⇒   ANS1 + ANS2
                                  where

                                                 2
                                      ANS2 := SIN(X *Z)

                                                          1/2
                                      ANS1 := ((X + 2)*X)
ans3;                     ⇒   ANS3

on fort;

structr((x+1)**5 + tan(x*y*z),var,aa);

                          ⇒

        VAR1=TAN(X*Y*Z)
        AA=VAR1+X**5+5.*X**4+10.*X**3+10.X**2+5.*X+1
```

Comments

The second argument to `structr` is optional. If it is not given, the default stem `ANS` is used by REDUCE to construct names for the subexpression. The names are only for display purposes: REDUCE does not store the names and their values unless the switch `savestructr` is on.

If a third argument is given, the structured expression as a whole is named by this argument, when `fort` is on. The expression is not stored under this name. You can send these structured Fortran expressions to a file with the `out` command.

8.46 SUB

SUB Operator

The `sub` operator substitutes a new expression for a kernel in an expression.

`sub`(*kernel*=*expression*{, *kernel*=*expression*}∗, *expression*) or
`sub`({*kernel*=*expression*∗, *kernel*=`expression`}, *expression*)

kernel must be a `kernel`, *expression* can be any REDUCE scalar expression.

Examples

```
sub(x=3,y=4,(x+y)**3);  ⇒  343

x;                      ⇒  X

sub({cos=sin,sin=cos},cos a+sin b)

                        ⇒  COS(B) + SIN(A)
```

Comments

Note in the second example that operators can be replaced using the `sub` operator.

8.47 SUM

SUM Operator

The operator `sum` returns the indefinite or definite summation of a given expression.

`sum`(*expr*, *k*[, *lolim*[, *uplim*]])

where *expr* is the expression to be added, *k* is the control variable (a `kernel`), and *lolim* and *uplim* are the optional lower and upper limits. If *uplim* is not supplied the upper limit is taken as *k*. The Gosper algorithm is used. If there is no closed form solution, the operator returns the input unchanged.

Examples

```
                          2     2
sum(4n**3,n);         ⇒  n  *(n  + 2*n + 1)

sum(2a+2k*r,k,0,n-1); ⇒   n*(2*a + n*r - r)
```

8.48 WS

WS Operator

The `ws` operator alone returns the last result; `ws` with a number argument returns the results of the REDUCE statement executed after that numbered prompt.

`ws` or `ws`(*number*)

number must be an integer between 1 and the current REDUCE prompt number.

Examples

(In the following examples, unlike most others, the numbered prompt is shown.)

```
1: df(sin y,y);  ⇒   COS(Y)
                            2
2: ws^2;         ⇒   COS(Y)
3: df(ws 1,y);   ⇒   -SIN(Y)
```

Comments

`ws` and `ws(`*number*`)` can be used anywhere the expression they stand for can be used. Calling a number for which no result was produced, such as a switch setting, will give an error message.

The current workspace always contains the results of the last REDUCE command that produced an expression, even if several input statements that do not produce expressions have intervened. For example, if you do a differentiation, producing a result expression, then change several switches, the operator `ws;` returns the results of the differentiation. The current workspace (`ws`) can also be used inside files, though the numbered workspace contains only the `in` command that input the file.

There are three history lists kept in your REDUCE session. The first stores raw input, suitable for the statement editor. The second stores parsed input, ready to execute and accessible by `input`. The third stores results, when they are produced by statements, which are accessible by the `ws` *n* operator. If your session is very long, storage space begins to fill up with these expressions, so it is a good idea to end the session once in a while, saving needed expressions to files with the `saveas` and `out` commands.

An error message is given if a reference number has not yet been used.

9 Declarations

9.1 ALGEBRAIC

ALGEBRAIC Command

The `algebraic` command changes REDUCE's mode of operation to algebraic. When `algebraic` is used as an operator (with an argument inside parentheses) that argument is evaluated in algebraic mode, but REDUCE's mode is not changed.

Examples

```
algebraic;

symbolic;           =>   NIL
                          2
algebraic(x**2);    =>   X

x**2;               =>     ***** The symbol X has no value.
```

Comments

REDUCE's symbolic mode does not know about most algebraic commands. Error messages in this mode may also depend on the particular Lisp used for the REDUCE implementation.

9.2 ANTISYMMETRIC

ANTISYMMETRIC Declaration

When an operator is declared `antisymmetric`, its arguments are reordered to conform to the internal ordering of the system. If an odd number of argument interchanges are required to do this ordering, the sign of the expression is changed.

`antisymmetric` *identifier*{`,`*identifier*}∗

identifier is an identifier that has been declared as an operator.

Examples

```
operator m,n;

antisymmetric m,n;

m(x,n(1,2));          ⇒   - M( - N(2,1),X)

operator p;

antisymmetric p;

p(a,b,c);             ⇒   P(A,B,C)

p(b,a,c);             ⇒   - P(A,B,C)
```

Comments

If *identifier* has not been declared an operator, the flag `antisymmetric` is still attached to it. When *identifier* is subsequently used as an operator, the message `Declare` *identifier* `operator? (Y or N)` is printed. If the user replies `y`, the antisymmetric property of the operator is used.

Note in the first example, identifiers are customarily ordered alphabetically, while numbers are ordered from largest to smallest. The operators may have any desired number of arguments (less than 128).

9.3 ARRAY

ARRAY — Declaration

The `array` declaration declares a list of identifiers to be of type `array`, and sets all their entries to 0.

`array` *identifier*(*dimensions*) {`,`*identifier*(*dimensions*)}*

identifier may be any valid REDUCE identifier. If the identifier was already an array, a warning message is given that the array has been redefined. *dimensions* are of form *integer*{,*integer*}*.

Examples

```
array a(2,5),b(3,3,3),c(200);
```

`array a(3,5);`	⇒	`*** ARRAY A REDEFINED`
`a(3,4);`	⇒	`0`
`length a;`	⇒	`{4,6}`

Comments

Arrays are always global, even if defined inside a procedure or block statement. Their status as an array remains until the variable is reset by `clear`. Arrays may not have the same names as operators, procedures or scalar variables.

Array elements are referred to by the usual notation: `a(i,j)` returns the jth element of the ith row. The `assignment` operator `:=` is used to put values into the array. Arrays as a whole cannot be subject to assignment by `let` or `:=` ; the assignment operator `:=` is only valid for individual elements.

When you use `let` on an array element, the contents of that element become the argument to `let`. Thus, if the element contains a number or some other expression that is not a valid argument for this command, you get an error message. If the element contains an identifier, the identifier has the substitution rule attached to it globally. The same behavior occurs with `clear`. If the array element contains an identifier or simple_expression, it is cleared. Do *not* use `clear` to try to set an array element to 0. Because of the side effects of either `let` or `clear`, it is unwise to apply either of these to array elements.

Array indices always start with 0, so that the declaration `array a(5)` sets aside 6 units of space, indexed from 0 through 5, and initializes them to 0. The `length` command returns a list of the true number of elements in each dimension.

9.4 CLEAR

CLEAR   Command

The `clear` command is used to remove assignments or remove substitution rules from any expression.

> `clear` *identifier*{,*identifier*}+ or
> *let-type statement* `clear` *identifier*

identifier can be any `scalar`, `matrix`, or `array` variable or `procedure` name. *let-type statement* can be any general or specific `let` statement (see below in Comments).

Examples

```
array a(2,3);

a(2,2) := 15;   ⇒   A(2,2) := 15

clear a;

a(2,2);         ⇒   Declare A operator? (Y or N)

let x = y + z;

sin(x);         ⇒   SIN(Y + Z)

clear x;

sin(x);         ⇒   SIN(X)

let x**5 = 7;

clear x;

x**5;           ⇒   7

clear x**5;
                     5
x**5;           ⇒   X
```

Comments

Although it is not a good idea, operators of the same name but taking different numbers of arguments can be defined. Using a `clear` statement on any of these operators clears every one with the same name, even if the number of arguments is different.

The `clear` command is used to "forget" matrices, arrays, operators and scalar variables, returning their identifiers to the pristine state to be used for other purposes. When `clear` is applied to array elements, the contents of the array element becomes the argument for `clear`. Thus, you get an error message if the element contains a number, or some other expression that is not a legal argument to `clear`. If the element contains an identifier, it is cleared. When clear is applied to matrix elements, an error message is returned if the element evaluates to a number, otherwise there is no effect. Do *not* try to use `clear` to set array or matrix elements to 0. You will not be pleased with the results.

If you are trying to clear power or product substitution rules made with either `let` or `forall`...`let`, you must reproduce the rule, exactly as you typed it with the same arguments, up to but not including the equal sign, using the word `clear` instead of the word `let`. This is shown in the last example. Any other type of `let` or `forall`...`let` substitution can be cleared with just the variable or operator name. `match` behaves the same as `let` in this situation. There is a more complicated example under `forall`.

9.5 CLEARRULES

CLEARRULES Command

`clearrules` *list*{,*list*}+

The operator `clearrules` is used to remove previously defined `rule` lists from the system. *list* can be an explicit rule list, or evaluate to a rule list.

Examples

```
trig1 := {cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2, cos(~x)*sin(~y) => (sin(x+y)-sin(x-y))/2,

                                    COS(A - B) + COS(A + B)
let trig1; cos(a)*cos(b);  ⇒   -------------------------
                                              2

clearrules trig1; cos(a)*cos(b);

                               ⇒   COS(A)*COS(B)
```

9.6 DEFINE

DEFINE Command

The command `define` allows you to supply a new name for an identifier or replace it by any valid REDUCE expression.

`define` *identifier*=*substitution* {,*identifier*=*substitution*}*

identifier is any valid REDUCE identifier, *substitution* can be a number, an identifier, an operator, a reserved word, or an expression.

Examples

```
define is= :=, xx=y+z;

a is 10;          ⇒   A := 10
                       2             2
xx**2;            ⇒   Y  + 2*Y*Z + Z

xx := 10;         ⇒   Y + Z := 10
```

Comments

The renaming is done at the input level, and therefore takes precedence over any other replacement or substitution declared for the same identifier. It remains in effect until the end of the REDUCE session. Be careful with it, since you cannot easily undo it without ending the session.

9.7 DEPEND

DEPEND Declaration

`depend` declares that its first argument depends on the rest of its arguments.

`depend` *kernel*{ , *kernel*}+

kernel must be a legal variable name or a prefix operator (see `kernel`).

Examples

```
depend y,x;

df(y**2,x);          ⇒   2*DF(Y,X)*Y

depend z,cos(x),y;

df(sin(z),cos(x));   ⇒   COS(Z)*DF(Z,COS(X))

df(z**2,x);          ⇒   2*DF(Z,X)*Z

nodepend z,y;

df(z**2,x);          ⇒   2*DF(Z,X)*Z

cc := df(y**2,x);    ⇒   CC := 2*DF(Y,X)*Y

y := tan x;          ⇒   Y := TAN(X);
                                         2
cc;                  ⇒   2*TAN(X)*(TAN(X)  + 1)
```

Comments

Dependencies can be removed by using the declaration `nodepend`. The differentiation operator uses this information, as shown in the examples above. Linear operators also use knowledge of dependencies (see `linear`). Note that dependencies can be nested: Having declared y to depend on x, and z to depend on y, we see that the chain rule was applied to the derivative of a function of z with respect to x. If the explicit function of the dependency is later entered into the system, terms with `DF(Y,X)`, for example, are expanded when they are displayed again, as shown in the last example. The boolean operator `freeof` allows you to check the dependency between two algebraic objects.

9.8 EVEN

EVEN Declaration

`even` *identifier*{,*identifier*}∗

This declaration is used to declare an operator *even* in its first argument. Expressions involving an operator declared in this manner are transformed if the first argument contains a minus sign. Any other arguments are not affected.

Examples

```
even f;
```

`f(-a)`	⇒	`F(A)`
`f(-a,-b)`	⇒	`F(A,-B)`

9.9 FACTOR

FACTOR — Declaration

When a kernel is declared by `factor`, all terms involving fixed powers of that kernel are printed as a product of the fixed powers and the rest of the terms.

`factor` *kernel* {,*kernel*}*

kernel must be a `kernel` or a `list` of `kernel`s.

Examples

```
a := (x + y + z)**2;      ⇒

                2                     2             2
       A := X  + 2*X*Y + 2*X*Z + Y  + 2*Y*Z + Z

factor y;
                                 2                  2            2
a;                          ⇒   Y  + 2*Y*(X + Z) + X  + 2*X*Z + Z

factor sin(x);

c := df(sin(x)**4*x**2*z,x);

                            ⇒
                      4                3         2
       C := 2*SIN(X) *X*Z + 4*SIN(X) *COS(X)*X *Z

remfac sin(x);
                                          3
c;                          ⇒   2*SIN(X) *X*Z*(2*COS(X)*X + SIN(X))
```

Comments

Use the `factor` declaration to display variables of interest so that you can see their powers more clearly, as shown in the example. Remove this special treatment with the declaration `remfac`. The `factor` declaration is only effective when the switch `pri` is on.

The `factor` declaration is not a factoring command; to factor expressions use the `factor` switch or the `factorize` command.

The `factor` declaration is helpful in such cases as Taylor polynomials where the explicit powers of the variable are expected at the top level, not buried in various factored forms.

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

9.10 FORALL

FORALL Command

The `forall` or (preferably) `for all` command is used as a modifier for `let` statements, indicating the universal applicability of the rule, with possible qualifications.

`for all` *identifier*{,*identifier*}∗ `let` *let statement*

or

`for all` *identifier*{,*identifier*}∗ `such that` *condition* `let` *let statement*

identifier may be any valid REDUCE identifier, *let statement* can be an operator, a product or power, or a group or block statement. *condition* must be a logical or comparison operator returning true or false.

Examples

```
for all x let f(x) = sin(x**2);

                         ⇒   Declare F operator ? (Y or N)

y
                                  2
f(a);                    ⇒   SIN(A )

operator pos;

for all x such that x>=0 let pos(x) = sqrt(x + 1);

pos(5);                  ⇒   SQRT(6)

pos(-5);                 ⇒   POS(-5)

clear pos;

pos(5);                  ⇒   Declare POS operator ? (Y or N)

for all a such that numberp a let x**a = 1;

x**4;                    ⇒   1

clear x**a;              ⇒   *** X**A not found
```

```
for all a clear x**a;

x**4;                        ⇒    1

for all a such that numberp a clear x**a;

                                   4
x**4;                        ⇒    X
```

Comments

Substitution rules defined by `for all` or `for all`...`such that` commands that involve products or powers are cleared by reproducing the command, with exactly the same variable names used, up to but not including the equal sign, with `clear` replacing `let`, as shown in the last example. Other substitutions involving variables or operator names can be cleared with just the name, like any other variable.

The `match` command can also be used in product and power substitutions. The syntax of its use and clearing is exactly like `let`. A `match` substitution only replaces the term if it is exactly like the pattern, for example `match x**5 = 1` replaces only terms of `x**5` and not terms of higher powers.

It is easier to declare your potential operator before defining the `for all` rule, since the system will ask you to declare it an operator anyway. Names of declared arrays or matrices or scalar variables are invalid as operator names, to avoid ambiguity. Either `for all`...`let` statements or procedures are often used to define operators. One difference is that procedures implement "call by value" meaning that assignments involving their formal parameters do not change the calling variables that replace them. If you use assignment statements on the formal parameters in a `for all`...`let` statement, the effects are seen in the calling variables. Be careful not to redefine a system operator unless you mean it: the statement `for all x let sin(x)=0;` has exactly that effect, and the usual definition for sin(x) has been lost for the remainder of the REDUCE session.

9.11 INFIX

INFIX — Declaration

`infix` declares identifiers to be infix operators.

`infix` *identifier*{,*identifier*}*

identifier can be any valid REDUCE identifier, which has not already been declared an operator, array or matrix, and is not reserved by the system.

Examples

```
infix aa;

for all x,y let aa(x,y) = cos(x)*cos(y) - sin(x)*sin(y);

x aa y;              ⇒   COS(X)*COS(Y) - SIN(X)*SIN(Y)
                            SQRT(3)
pi/3 aa pi/2;        ⇒   - ---------
                               2

aa(pi,pi);           ⇒   1
```

Comments

A `let` statement must be used to attach functionality to the operator. Note that the operator is defined in prefix form in the `let` statement. After its definition, the operator may be used in either prefix or infix mode. The above operator *aa* finds the cosine of the sum of two angles by the formula

$$\cos(x + y) = \cos x \cos y - \sin x \sin y.$$

Precedence may be attached to infix operators with the `precedence` declaration.

User-defined infix operators may be used in prefix form. If they are used in infix form, a space must be left on each side of the operator to avoid ambiguity. Infix operators are always binary.

9.12 INTEGER

INTEGER Declaration

The `integer` declaration must be made immediately after a `begin` (or other variable declaration such as `real` and `scalar`) and declares local integer variables. They are initialized to 0.

`integer` *identifier*{,*identifier*}∗

identifier may be any valid REDUCE identifier, except `t` or `nil`.

Comments

Integer variables remain local, and do not share values with variables of the same name outside the `begin`...`end` block. When the block is finished, the variables are removed. You may use the words `real` or `scalar` in the place of `integer`. `integer` does not indicate typechecking by the current REDUCE; it is only for your own information. Declaration statements must immediately follow the `begin`, without a semicolon between `begin` and the first variable declaration.

Any variables used inside `begin`...`end` blocks that were not declared `scalar`, `real` or `integer` are global, and any change made to them inside the block affects their global value. Any `array` or `matrix` declared inside a block is always global.

9.13 KORDER

KORDER Declaration

The `korder` declaration changes the internal canonical ordering of kernels.

`korder` *kernel*{`,`*kernel*}∗

kernel must be a REDUCE `kernel` or a `list` of `kernel`s.

Comments

The declaration `korder` changes the internal ordering, but not the print ordering, so the effects cannot be seen on output. However, in some calculations, the order of the variables can have significant effects on the time and space demands of a calculation. If you are doing a demanding calculation with several kernels, you can experiment with changing the canonical ordering to improve behavior.

The first kernel in the argument list is given the highest priority, the second gets the next highest, and so on. Kernels not named in a `korder` ordering otherwise. A new `korder` declaration replaces the previous one. To return to canonical ordering, use the command `korder nil`.

To change the print ordering, use the declaration `order`.

9.14 LET

LET Command

The `let` command defines general or specific substitution rules.

`let` *identifier* = *expression*{,*identifier* = *expression*}*

identifier can be any valid REDUCE identifier except an array, and in some cases can be an expression; *expression* can be any valid REDUCE expression.

Examples

```
let a = sin(x);

b := a;                 ⇒   B := SIN X;

let c = a;

                             SIN(X)
exp(a);                 ⇒   E

                                   2
a := x**2;              ⇒   A := X

                               2
                              X
exp(a);                 ⇒   E

                             SIN(X)
exp(b);                 ⇒   E

                               2
                              X
exp(c);                 ⇒   E

let m + n = p;

                              5
(m + n)**5;             ⇒   P

operator h;

let h(u,v) = u - v;

h(u,v);                 ⇒   U - V

h(x,y);                 ⇒   H(X,Y)

array q(10);
```

```
let q(1) = 15;        ⇒

        ***** Substitution for 0 not allowed
```

The `let` command is also used to activate a `rule sets`.

`let` *list*{,*list*}+

list can be an explicit `rule list`, or evaluate to a rule list.

Examples

```
trig1 := {cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2, cos(~x)*sin(~y) => (sin(x+y)-sin(x-y))/2,
                                COS(A - B) + COS(A + B)
let trig1; cos(a)*cos(b);  ⇒   -------------------------
                                           2
```

Comments

A `let` command returns no value, though the substitution rule is entered. Assignment rules made by `assign` and `let` rules are at the same level, and cancel each other. There is a difference in their operation, however, as shown in the first example: a `let` assignment tracks the changes in what it is assigned to, while a `:=` assignment is fixed at the value it originally had.

The use of expressions as left-hand sides of `let` statements is a little complicated. The rules of operation are:

(i) Expressions of the form A*B = C do not change A, B or C, but set A*B to C.

(ii) Expressions of the form A+B = C substitute C - B for A, but do not change B or C.

(iii) Expressions of the form A-B = C substitute B + C for A, but do not change B or C.

(iv) Expressions of the form A/B = C substitute B*C for A, but do not change B or C.

(v) Expressions of the form A**N = C substitute C for A**N in every expression of a power of A to N or greater. An asymptotic command such as A**N = 0 sets all terms involving A to powers greater than or equal to N to 0. Finite fields may be generated by requiring modular arithmetic (the `modular` switch) and defining the primitive polynomial via a `let` statement.

`let` substitutions involving expressions are cleared by using the `clear` command with exactly the same expression.

Note when a simple `let` statement is used to assign functionality to an operator, it is valid only for the exact identifiers used. For the use of the `let` command to attach more general functionality to an operator, see `forall`.

Arrays as a whole cannot be arguments to `let` statements, but matrices as a whole can be legal arguments, provided both arguments are matrices. However, it is important to note that the two matrices are then linked. Any change to an element of one matrix changes the corresponding value in the other. Unless you want this behavior, you should not use `let` for matrices. The assignment operator `assign` can be used for non-tracking assignments, avoiding the side effects. Matrices are redimensioned as needed in `let` statements.

When array or matrix elements are used as the left-hand side of `let` statements, the contents of that element is used as the argument. When the contents is a number or some other expression that is not a valid left-hand side for `let`, you get an error message. If the contents is an identifier or simple expression, the `let` rule is globally attached to that identifier, and is in effect not only inside the array or matrix, but everywhere. Because of such unwanted side effects, you should not use `let` with array or matrix elements. The assignment operator `:=` can be used to put values into array or matrix elements without the side effects.

Local variables declared inside `begin`...`end` blocks cannot be used as the left-hand side of `let` statements. However, `begin`...`end` blocks themselves can be used as the right-hand side of `let` statements. The construction:

`for all` *vars* `let`*operator*(*vars*)`=`*block*

is an alternative to the

`procedure` *name*(*vars*)`;`*block*

construction. One important difference between the two constructions is that the *vars* as formal parameters to a procedure have their global values protected against change by the procedure, while the *vars* of a `let` statement are changed globally by its actions.

Be careful in using a construction such as `let x = x + 1` except inside a controlled loop statement. The process of resubstitution continues until a stack overflow message is given.

The `let` statement may be used to make global changes to variables from inside procedures. If `x` is a formal parameter to a procedure, the command `let x =` ... makes the change to the calling variable. For example, if a procedure was defined by

```
procedure f(x,y);
let x = 15;
```

and the procedure was called as

```
f(a,b);
```

`a` would have its value changed to 15. Be careful when using `let` statements inside procedures to avoid unwanted side effects.

It is also important to be careful when replacing `let` statements with other `let` statements. The overlapping of these substitutions can be unpredictable. Ordinarily the latest-entered rule is the first to be applied. Sometimes the previous rule is superseded completely; other times it stays around as a special case. The order of entering a set of related `let` expressions is very important to their eventual behavior. The best approach is to assume that the rules will be applied in an arbitrary order.

9.15 LINEAR

LINEAR — Declaration

An operator can be declared linear in its first argument over powers of its second argument by the declaration `linear`.

`linear` *operator*{`,`*operator*}∗

operator must have been declared to be an operator. Be careful not to use a system operator name, because this command may change its definition. The operator being declared must have at least two arguments, and the second one must be a `kernel`.

Examples

```
operator f;

linear f;

f(0,x);              ⇒   0

f(-y,x);             ⇒   - F(1,X)*Y

f(y+z,x);            ⇒   F(1,X)*(Y + Z)

f(y*z,x);            ⇒   F(1,X)*Y*Z

depend z,x;

f(y*z,x);            ⇒   F(Z,X)*Y
                           1
f(y/z,x);            ⇒   F(---,X)*Y
                           Z

depend y,x;
                           Y
f(y/z,x);            ⇒   F(---,X)
                           Z

nodepend z,x;
                          F(Y,X)
f(y/z,x);            ⇒   --------
                            Z
                            SIN(X)
f(2*e**sin(x),x);    ⇒   2*F(E      ,X)
```

Comments

Even when the operator has not had its functionality attached, it exhibits linear properties as shown in the examples. Notice the difference when dependencies are added. Dependencies are also in effect when the operator's first argument contains its second, as in the last line above.

For a fully-developed example of the use of linear operators, refer to the article in the *Journal of Computational Physics*, Vol. 14 (1974), pp. 301-317, "Analytic Computation of Some Integrals in Fourth Order Quantum Electrodynamics," by J.A. Fox and A.C. Hearn. The article includes the complete listing of REDUCE procedures used for this work.

9.16 LINELENGTH

LINELENGTH Declaration

The `linelength` declaration sets the length of the output line. Default is 80.

`linelength` *expression*

To change the linelength, *expression* must evaluate to a positive integer less than 128 (although this varies from system to system), and should not be less than 20 or so for proper operation.

Comments

`linelength` returns the previous linelength. If you want the current linelength value, but not change it, say `linelength nil`.

9.17 LISP

LISP Command

The `lisp` command changes REDUCE's mode of operation to symbolic. When `lisp` is followed by an expression, that expression is evaluated in symbolic mode, but REDUCE's mode is not changed. This command is equivalent to `symbolic`.

Examples

```
lisp;                        ⇒   NIL

car '(a b c d e);            ⇒   A

algebraic;

c := (lisp car '(first second))**2;
                                           2
                             ⇒   C := FIRST
```

9.18 LISTARGP

LISTARGP Declaration

`listargp` *operator*{,*operator*}*

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. This process can be inhibited for a specific operator, or list of operators, by using the declaration `listargp`.

Examples

```
log {a,b,c};    ⇒   LOG(A),LOG(B),LOG(C)

listargp log;

log {a,b,c};    ⇒   LOG(A,B,C)
```

Comments

It is possible to inhibit such distribution globally by turning on the switch `listargs`. In addition, if an operator has more than one argument, no such distribution occurs, so `listargp` has no effect.

9.19 NODEPEND

NODEPEND Declaration

The `nodepend` declaration removes the dependency declared with `depend`.

`nodepend` *dep-kernel*{,*kernel*}+

dep-kernel must be a kernel that has had a dependency declared upon the one or more other kernels that are its other arguments.

Examples

```
depend y,x,z;

df(sin y,x);     ⇒   COS(Y)*DF(Y,X)

df(sin y,x,z);   ⇒

       COS(Y)*DF(Y,X,Z) - DF(Y,X)*DF(Y,Z)*SIN(Y)

nodepend y,z;

df(sin y,x);     ⇒   COS(Y)*DF(Y,X)

df(sin y,x,z);   ⇒   0
```

Comments

A warning message is printed if the dependency had not been declared by `depend`.

9.20 MATCH

MATCH Command

The match command is similar to the let command, except that it matches only explicit powers in substitution.

match *expr* = *expression*{,*expr* =*expression*}*

expr is generally a term involving powers, and is limited by the rules for the let command. *expression* may be any valid REDUCE scalar expression.

Examples

```
match c**2*a**2 = d; (a+c)**4;
                               4      3          3    4
                        ⇒   A  + 4*A *C + 4*A*C  + C  + 6*D

match a+b = c;

a + 2*b;                ⇒   B + C
                             2    2              2
(a + b + c)**2;         ⇒   A  - B  + 2*B*C + 3*C

clear a+b;

(a + b + c)**2;         ⇒
          2                       2            2
         A  + 2*A*B + 2*A*C + B  + 2*B*C + C

let p*r = s;

match p*q = ss;
                             2            2
(a + p*r)**2;           ⇒   A  + 2*A*S + S
                             2              2  2
(a + p*q)**2;           ⇒   A  + 2*A*SS + P *Q
```

Comments

Note in the last example that a + b has been explicitly matched after the squaring was done, replacing each single power of a by c - b. This kind of substitution, although following the rules, is confusing and could lead to unrecognizable results.

It is better to use `match` with explicit powers or products only. `match` should not be used inside procedures for the same reasons that `let` should not be.

Unlike `let` substitutions, `match` substitutions are executed after all other operations are complete. The last example shows the difference. `match` commands can be cleared by using `clear`, with exactly the expression that the original `match` took. `match` commands can also be done more generally with `for all` or `forall`... `such that` commands.

9.21 NONCOM

NONCOM — Declaration

`noncom` declares that already-declared operators are noncommutative under multiplication.

`noncom` *operator*{,*operator*}∗

operator must have been declared an `operator`, or a warning message is given.

Examples

```
operator f,h;

noncom f;

f(a)*f(b) - f(b)*f(a);  ⇒   F(A)*F(B) - F(B)*F(A)

h(a)*h(b) - h(b)*h(a);  ⇒   0

operator comm;

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

f(1)*f(2);              ⇒   F(1)*F(2)

f(2)*f(1);              ⇒   COMM(2,1) + F(1)*F(2)
```

Comments

The last example introduces the commutator of $f(x)$ and $f(y)$ for all x and y. The equality check is to prevent an infinite loop. The operator f can have other functionality attached to it if desired, or it can remain an indeterminate operator.

9.22 NONZERO

NONZERO Declaration

nonzero *identifier*{,*identifier*}∗

If an operator f is declared odd, then f(0) is replaced by zero unless f is also declared *non zero* by the declaration nonzero.

Examples

```
odd f;

f(0)        ⇒   0

nonzero f;

f(0)        ⇒   F(0)
```

9.23 ODD

ODD Declaration

`odd` *identifier*{,*identifier*}*

This declaration is used to declare an operator *odd* in its first argument. Expressions involving an operator declared in this manner are transformed if the first argument contains a minus sign. Any other arguments are not affected.

Examples

`odd f;`

`f(-a)`	⇒	`-F(A)`
`f(-a,-b)`	⇒	`-F(A,-B)`
`f(a,-b)`	⇒	`F(A,-B)`

Comments

If say `f` is declared odd, then `f(0)` is replaced by zero unless `f` is also declared *non zero* by the declaration `nonzero`.

9.24 OFF

OFF Command

The `off` command is used to turn switches off.

`off` *switch*{,*switch*}*

switch can be any `switch` name. There is no problem if the switch is already off. If the switch name is mistyped, an error message is given.

9.25 ON

ON   Command

The `on` command is used to turn switches on.

`on` *switch*{,*switch*}∗

switch can be any `switch` name. There is no problem if the switch is already on. If the switch name is mistyped, an error message is given.

9.26 OPERATOR

OPERATOR Declaration

Use the `operator` declaration to declare your own operators.

`operator` *identifier*{,*identifier*}∗

identifier can be any valid REDUCE identifier, which is not the name of a `matrix`, `array`, scalar variable or previously-defined operator.

Examples

```
operator dis,fac;

let dis(~x,~y) = sqrt(x^2 + y^2);

dis(1,2);               ⇒   SQRT(5)
                                  2
dis(a,10);              ⇒   SQRT(A  + 100)

on rounded;

dis(1.5,7.2);           ⇒   7.35459040329

let fac(~n) = if n=0 then 1
               else if not(fixp n and n>0)
                then rederr "choose non-negative integer"
               else for i := 1:n product i;

fac(5);                 ⇒   120

fac(-2);                ⇒   ***** choose non-negative integer
```

Comments

The first operator is the Euclidean distance metric, the distance of point (x, y) from the origin. The second operator is the factorial.

Operators can have various properties assigned to them; they can be declared `infix`, `linear`, `symmetric`, `antisymmetric`, or `noncommutative`. The default operator is prefix, nonlinear, and commutative. Precedence can also be assigned to operators using the declaration `precedence`.

Functionality is assigned to an operator by a `let` statement or a `forall`...`let` statement, (or possibly by a procedure with the name of the operator). Be careful not to redefine a system operator by accident. REDUCE permits you to redefine system operators, giving you a warning message that the operator was already defined. This flexibility allows you to add mathematical rules that do what you want them to do, but can produce odd or erroneous behavior if you are not careful.

You can declare operators from inside `procedure`s, as long as they are not local variables. Operators defined inside procedures are global. A formal parameter may be declared as an operator, and has the effect of declaring the calling variable as the operator.

9.27 ORDER

ORDER Declaration

The `order` declaration changes the order of precedence of kernels for display purposes only.

`order` *identifier*{,*identifier*}∗

kernel must be a valid `kernel` or `operator` name complete with argument or a `list` of such objects.

Examples

```
x + y + z + cos(a);   ⇒   COS(A) + X + Y + Z

order z,y,x,cos(a);

x + y + z + cos(a);   ⇒   Z + Y + X + COS(A)

                           2             2
(x + y)**2;           ⇒   Y  + 2*Y*X + X

order nil;

                                 2                   2
(z + cos(z))**2;      ⇒   COS(Z)  + 2*COS(Z)*Z + Z
```

Comments

`order` affects the printing order of the identifiers only; internal order is unchanged. Change internal order of evaluation with the declaration `korder`. You can use `order` to feature variables or functions you are particularly interested in.

Declarations made with `order` are cumulative: kernels in new order declarations are ordered behind those in previous declarations, and previous declarations retain their relative order. Of course, specific kernels named in new declarations are removed from previous ones and given the new priority. Return to the standard canonical printing order with the statement `order nil`.

The print order specified by `order` commands is not in effect if the switch `pri` is off.

9.28 PRECEDENCE

PRECEDENCE — Declaration

The `precedence` declaration attaches a precedence to an infix operator.

`precedence` *operator*,*known_operator*

operator should have been declared an operator but may be a REDUCE identifier that is not already an operator, array, or matrix. *known_operator* must be a system infix operator or have had its precedence already declared.

Examples

```
operator f,h;

precedence f,+;

precedence h,*;

a + f(1,2)*c;     ⇒   (1 F 2)*C + A

a + h(1,2)*c;     ⇒   1 H 2*C + A

a*1 f 2*c;        ⇒   A F 2*C

a*1 h 2*c;        ⇒   1 H 2*A*C
```

Comments

The operator whose precedence is being declared is inserted into the infix operator precedence list at the next higher place than *known_operator*.

Attaching a precedence to an operator has the side effect of declaring the operator to be infix. If the identifier argument for `precedence` has not been declared to be an operator, an attempt to use it causes an error message. After declaring it to be an operator, it becomes an infix operator with the precedence previously given. Infix operators may be used in prefix form; if they are used in infix form, a space must be left on each side of the operator to avoid ambiguity. Declared infix operators are always binary.

To see the infix operator precedence list, enter symbolic mode and type `preclis!*;`. The lowest precedence operator is listed first.

All prefix operators have precedence higher than infix operators.

9.29 PRECISION

PRECISION Declaration

The `precision` declaration sets the number of decimal places used when `rounded` is on. Default is system dependent, and normally about 12.

`precision`(*integer*) or `precision` *integer*

integer must be a positive integer. When *integer* is 0, the current precision is displayed, but not changed. There is no upper limit, but precision of greater than several hundred causes unpleasantly slow operation on numeric calculations.

Examples

```
on rounded;

7/9;           ⇒   0.777777777778

precision 20;  ⇒   20

7/9;           ⇒   0.77777777777777777778

sin(pi/4);     ⇒   0.7071067811865475244
```

Comments

Trailing zeroes are dropped, so sometimes fewer than 20 decimal places are printed as in the last example. Turn on the switch `fullprec` if you want to print all significant digits. The `rounded` mode carries calculations to two more places than given by `precision`, and rounds off.

9.30 PRINT_PRECISION

PRINT_PRECISION — Declaration

`print_precision`(*integer*) or `print_precision` *integer*

In `rounded` mode, numbers are normally printed to the specified precision. If the user wishes to print such numbers with less precision, the printing precision can be set by the declaration `print_precision`.

Examples

```
on rounded;

1/3;                   ⇒    0.333333333333

print_precision 5;

1/3                    ⇒    0.33333
```

9.31 REAL

REAL   Declaration

The `real` declaration must be made immediately after a `begin` (or other variable declaration such as `integer` and `scalar`) and declares local integer variables. They are initialized to zero.

`real` *identifier*{,*identifier*}∗

identifier may be any valid REDUCE identifier, except `t` or `nil`.

Comments

Real variables remain local, and do not share values with variables of the same name outside the `begin`...`end` block. When the block is finished, the variables are removed. You may use the words `integer` or `scalar` in the place of `real`. `real` does not indicate typechecking by the current REDUCE; it is only for your own information. Declaration statements must immediately follow the `begin`, without a semicolon between `begin` and the first variable declaration.

Any variables used inside a `begin`...`end block` that were not declared `scalar`, `real` or `integer` are global, and any change made to them inside the block affects their global value. Any **??** or **??** declared inside a block is always global.

9.32 REMFAC

REMFAC Declaration

The `remfac` declaration removes the special factoring treatment of its arguments that was declared with `factor`.

`remfac` *kernel*{,*kernel*}+

kernel must be a `kernel` or `operator` name that was declared as special with the `factor` declaration.

9.33 SCALAR

SCALAR — Declaration

The `scalar` declaration must be made immediately after a `begin` (or other variable declaration such as `integer` and `real`) and declares local scalar variables. They are initialized to 0.

`scalar` *identifier*{,*identifier*}∗

identifier may be any valid REDUCE identifier, except `t` or `nil`.

Comments

Scalar variables remain local, and do not share values with variables of the same name outside the `begin`...`end` `block`. When the block is finished, the variables are removed. You may use the words `real` or `integer` in the place of `scalar`. `real` and `integer` do not indicate typechecking by the current REDUCE; they are only for your own information. Declaration statements must immediately follow the `begin`, without a semicolon between `begin` and the first variable declaration.

Any variables used inside `begin`...`end` blocks that were not declared `scalar`, `real` or `integer` are global, and any change made to them inside the block affects their global value. Arrays declared inside a block are always global.

9.34 SCIENTIFIC_NOTATION

SCIENTIFIC_NOTATION Declaration

`scientific_notation`(m) or `scientific_notation`($\{m,n\}$)

m and n are positive integers. `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.

When `scientific_notation` is called with the numerical argument m 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. When `scientific_notation` is called with a list $\{m,n\}$, 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.

Examples

```
on rounded;

12345.6;                  ⇒   12345.6

123456.5;                 ⇒  1.234565e+5

0.00000000000000012;      ⇒   1.2e-16

scientific_notation 20;   ⇒   5,11

5: 123456.7;              ⇒   123456.7

0.00000000000000012;      ⇒   0.00000000000000012
```

9.35 SHARE

SHARE — Declaration

The `share` declaration allows access to its arguments by both algebraic and symbolic modes.

`share` *identifier*{,*identifier*}∗

identifier can be any valid REDUCE identifier.

Comments

Programming in `symbolic` as well as algebraic mode allows you a wider range of techniques than just algebraic mode alone. Expressions do not cross the boundary since they have different representations, unless the `share` declaration is used. For more information on using symbolic mode, see the *REDUCE User's Manual*, and the *Standard Lisp Report*.

You should be aware that a previously-declared array is destroyed by the `share` declaration. Scalar variables retain their values. You can share a declared `matrix` that has not yet been dimensioned so that it can be used by both modes. Values that are later put into the matrix are accessible from symbolic mode too, but not by the usual matrix reference mechanism. In symbolic mode, a matrix is stored as a list whose first element is `MAT`, and whose next elements are the rows of the matrix stored as lists of the individual elements. Access in symbolic mode is by the operators `first`, `second`, `third` and `rest`.

9.36 SYMBOLIC

SYMBOLIC Command

The `symbolic` command changes REDUCE's mode of operation to symbolic. When `symbolic` is followed by an expression, that expression is evaluated in symbolic mode, but REDUCE's mode is not changed. It is equivalent to the `lisp` command.

Examples

```
symbolic;                      ⇒   NIL

cdr '(a b c);                  ⇒   (B C)

algebraic;

x + symbolic car '(y z);       ⇒   X + Y
```

9.37 SYMMETRIC

SYMMETRIC Declaration

When an operator is declared `symmetric`, its arguments are reordered to conform to the internal ordering of the system.

`symmetric` *identifier*{,*identifier*}∗

identifier is an identifier that has been declared an operator.

Examples

```
operator m,n;

symmetric m,n;

m(y,a,sin(x));   ⇒   M(SIN(X),A,Y)

n(z,m(b,a,q));   ⇒   N(M(A,B,Q),Z)
```

Comments

If *identifier* has not been declared to be an operator, the flag `symmetric` is still attached to it. When *identifier* is subsequently used as an operator, the message `Declare`*identifier* `operator ?  (Y or N)` is printed. If the user replies `y`, the symmetric property of the operator is used.

9.38 TR

TR   Declaration

The `tr` declaration is used to trace system or user-written procedures. It is only useful to those with a good knowledge of both Lisp and the internal formats used by REDUCE.

`tr` *name*{,*name*}∗

name is the name of a REDUCE system procedure or one of your own procedures.

Examples

The system procedure `prepsq` *is traced, which prepares REDUCE standard forms for printing by converting them to a Lisp prefix form.*

```
tr prepsq;     ⇒   (PREPSQ)

x**2 + y;      ⇒

       PREPSQ entry:
         Arg 1: (((((X . 2) . 1) ((Y . 1) . 1)) . 1)
       PREPSQ return value = (PLUS (EXPT X 2) Y)
       PREPSQ entry:
         Arg 1: (1 . 1)
       PREPSQ return value = 1

        2
       X  + Y
untr prepsq;  ⇒   (PREPSQ)
```

Comments

This example is for a PSL-based system; the above format will vary if other Lisp systems are used.

When a procedure is traced, the first lines show entry to the procedure and the arguments it is given. The value returned by the procedure is printed upon exit. If you are tracing several procedures, with a call to one of them inside the other, the inner trace will be indented showing procedure nesting. There are no trace options. However, the format of the trace depends on the underlying Lisp system used. The trace can be removed with the command `untr`. Note that `trace`, below, is a matrix operator, while `tr` does procedure tracing.

9.39 UNTR

UNTR Declaration

The `untr` declaration is used to remove a trace from system or user-written procedures declared with `tr`. It is only useful to those with a good knowledge of both Lisp and the internal formats used by REDUCE.

`untr` *name*{,*name*}*

name is the name of a REDUCE system procedure or one of your own procedures that has previously been the argument of a `tr` declaration.

9.40 VARNAME

VARNAME — Declaration

The declaration `varname` instructs REDUCE to use its argument as the default Fortran (when `fort` is on) or `structr` identifier and identifier stem, rather than using `ANS`.

`varname` *identifier*

identifier can be any combination of one or more alphanumeric characters. Try to avoid REDUCE reserved words.

Examples

```
varname ident;                  =>   IDENT

on fort;

x**2 + 1;                       =>   IDENT=X**2+1.

off fort,exp;
                                           3
structr(((x+y)**2 + z)**3);     =>  IDENT2
                                        where

                                                        2
                                           IDENT2 := IDENT1  + Z
                                    IDENT1 := X + Y
```

Comments

`exp` was turned off so that `structr` could show the structure. If `exp` had been on, the expression would have been expanded into a polynomial.

9.41 WEIGHT

WEIGHT Command

The `weight` command is used to attach weights to kernels for asymptotic constraints.

`weight` *kernel* =*number*

kernel must be a REDUCE `kernel`, *number* must be a positive integer, not 0.

Examples

```
a := (x+y)**4;   ⇒
              4      3        2  2        3    4
       A := X  + 4*X *Y + 6*X *Y  + 4*X*Y  + Y

weight x=2,y=3;

wtlevel 8;
                       4
a;               ⇒   X

wtlevel 10;
                       2     2            2
a;               ⇒   X *(6*Y  + 4*X*Y + X )

int(x**2,x);     ⇒   ***** X invalid as KERNEL
```

Comments

Weights and `wtlevel` are used for asymptotic constraints, where higher-order terms are considered insignificant.

Weights are originally equivalent to 0 until set by a `weight` command. To remove a weight from a kernel, use the `clear` command. Weights once assigned cannot be changed without clearing the identifier. Once a weight is assigned to a kernel, it is no longer a kernel and cannot be used in any REDUCE commands or operators that require kernels, until the weight is cleared. Note that terms are ordered by greatest weight.

The weight level of the system is set by `wtlevel`, initially at 2. Since no kernels have weights, no effect from `wtlevel` can be seen. Once you assign weights to kernels, you must set `wtlevel` correctly for the desired operation. When weighted variables appear in a term, their weights are summed for the total weight of the term (powers of variables multiply their weights). When a term exceeds the weight level of the system, it is discarded from the result expression.

9.42 WHERE

WHERE — Operator

The where operator provides an infix notation for one-time substitutions for kernels in expressions.

expression `where` *kernel* =*expression* {,*kernel* =*expression*}*

expression can be any REDUCE scalar expression, *kernel* must be a `kernel`. Alternatively a `rule` or a `rule list` can be a member of the right-hand part of a `where` expression.

Examples

```
x**2 + 17*x*y + 4*y**2 where x=1,y=2;

                              =>  51

for i := 1:5 collect x**i*q where q= for j := 1:i product j;

                                         2    3     4      5
                              =>  {X,2*X ,6*X ,24*X ,120*X }

x**2 + y + z where z=y**3,y=3;

                                   2    3
                              =>  X  + Y  + 3
```

Comments

Substitution inside a `where` expression has no effect upon the values of the kernels outside the expression. The `where` operator has the lowest precedence of all the infix operators, which are lower than prefix operators, so that the substitutions apply to the entire expression preceding the `where` operator. However, `where` is applied before command keywords such as `then`, `repeat`, or `do`.

A `rule` or a `rule set` in the right-hand part of the `where` expression act as if the rules were activated by `let` immediately before the evaluation of the expression and deactivated by `clearrules` immediately afterwards.

`where` gives you a natural notation for auxiliary variables in expressions. As the second example shows, the substitute expression can be a command to be evaluated. The substitute assignments are made in parallel, rather than sequentially, as the last example shows. The expression resulting from the first round of substitutions is not reexamined to see if any further such substitutions can be made. `where` can also be used to define auxiliary variables in `procedure` definitions.

9.43 WHILE

WHILE Command

The `while` command causes a statement to be repeatedly executed until a given condition is true. If the condition is initially false, the statement is not executed at all.

`while` *condition* `do` *statement*

condition is given by a logical operator, *statement* must be a single REDUCE statement, or a `group` (`<<`...`>>`) or `begin`...`end` `block`.

Examples

```
a := 10;                          ⇒   A := 10

while a <= 12 do <<write a; a := a + 1>>;

                                  ⇒   10

11

12

while a < 5 do <<write a; a := a + 1>>;
```

⇒ *nothing is printed*

9.44 WTLEVEL

WTLEVEL — Command

In conjunction with **weight**, **wtlevel** is used to implement asymptotic constraints. Its default value is 2.

`wtlevel` *expression*

To change the weight level, *expression* must evaluate to a positive integer that is the greatest weight term to be retained in expressions involving kernels with weight assignments. `wtlevel` returns the new weight level. If you want the current weight level, but not change it, say `wtlevel nil`.

Examples

```
                         4       3         2  2         3    4
(x+y)**4;          =>   X  + 4*X *Y + 6*X *Y  + 4*X*Y  + Y

weight x=2,y=3;

wtlevel 8;
                         4
(x+y)**4;          =>   X

wtlevel 10;
                         2     2              2
(x+y)**4;          =>   X *(6*Y  + 4*X*Y + X )

int(x**2,x);       =>   ***** X invalid as KERNEL
```

Comments

`wtlevel` is used in conjunction with the command `weight` to enable asymptotic constraints. Weight of a term is computed by multiplying the weights of each variable in it by the power to which it has been raised, and adding the resulting weights for each variable. If the weight of the term is greater than `wtlevel`, the term is dropped from the expression, and not used in any further computation involving the expression.

Once a weight has been attached to a `kernel`, it is no longer recognized by the system as a kernel, though still a variable. It cannot be used in REDUCE commands and operators that need kernels. The weight attachment can be undone with a `clear` command. `wtlevel` can be changed as desired.

10 Input and Output

10.1 IN

IN Command

The `in` command takes a list of file names and inputs each file into the system.

`in` *filename*{,*filename*}∗

filename must be in the current directory, or be a valid pathname. If the file name is not an identifier, double quote marks (`"`) are needed around the file name.

Comments

A message is given if the file cannot be found, or has a mistake in it.

Ending the command with a semicolon causes the file to be echoed to the screen; ending it with a dollar sign does not echo the file. If you want some but not all of a file echoed, turn the switch `echo` on or off in the file.

An efficient way to develop procedures in REDUCE is to write them into a file using a system editor of your choice, and then input the files into an active REDUCE session. REDUCE reparses the procedure as it takes information from the file, overwriting the previous procedure definition. When it accepts the procedure, it echoes its name to the screen. Data can also be input to the system from files.

Files to be read in should always end in `end;` to avoid end-of-file problems. Note that this is an additional `end;` to any ending procedures in the file.

10.2 INPUT

INPUT — Command

The `input` command returns the input expression to the REDUCE numbered prompt that is its argument.

`input`(*number*) or `input` *number*

number must be between 1 and the current REDUCE prompt number.

Comments

An expression brought back by `input` can be reexecuted with new values or switch settings, or used as an argument in another expression. The command `ws` brings back the results of a numbered REDUCE statement. Two lists contain every input and every output statement since the beginning of the session. If your session is very long, storage space begins to fill up with these expressions, so it is a good idea to end the session once in a while, saving needed expressions to files with the `saveas` and `out` commands.

Switch settings and `let` statements can also be reexecuted by using `input`.

An error message is given if a number is called for that has not yet been used.

10.3 OUT

OUT Command

The `out` command directs output to the filename that is its argument, until another `out` changes the output file, or `shut` closes it.

`out` *filename* or `out "`*pathname* `"` or `out t`

filename must be in the current directory, or be a valid complete file description for your system. If the file name is not in the current directory, quote marks are needed around the file name. If the file already exists, a message is printed allowing you to decide whether to supersede the contents of the file with new material.

Comments

To restore output to the terminal, type `out t`, or `shut` the file. When you use `out t`, the file remains available, and if you open it again (with another `out`), new material is appended rather than overwriting.

To write a file using `out` that can be input at a later time, the switch `nat` must be turned off, so that the standard linear form is saved that can be read in by `in`. If `nat` is on, exponents are printed on the line above the expression, which causes trouble when REDUCE tries to read the file.

There is a slight complication if you are using the `out` command from inside a file to create another file. The `echo` switch is normally off at the top-level and on while reading files (so you can see what is being read in). If you create a file using `out` at the top-level, the result lines are printed into the file as you want them. But if you create such a file from inside a file, the `echo` switch is on, and every line is echoed, first as you typed it, then as REDUCE parsed it, and then once more for the file. Therefore, when you create a file *from* a file, you need to turn `echo` off explicitly before the `out` command, and turn it back on when you `shut` the created file, so your executing file echoes as it should. This behavior also means that as you watch the file execute, you cannot see the lines that are being put into the `out` file. As soon as you turn `echo` on, you can see output again.

10.4 SHUT

SHUT — Command

The `shut` command closes output files.

`shut` *filename*{,*filename*}∗

filename must have been a file opened by `out`.

Comments

A file that has been opened by `out` must be `shut` before it is brought in by `in`. Files that have been opened by `out` should always be `shut` before the end of the REDUCE session, to avoid either loss of information or the printing of extraneous information into the file. In most systems, terminating a session by `bye` closes all open output files.

11 Elementary Functions

11.1 ACOS

ACOS   Operator

The `acos` operator returns the arccosine of its argument.

`acos`(*expression*) or `acos` *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

```
acos(ab);               ⇒   ACOS(AB)

acos 15;                ⇒   ACOS(15)

                                      2  2
                             SQRT( - X *Y  + 1)*Y
df(acos(x*y),x);        ⇒   ----------------------
                                   2  2
                                  X *Y  - 1

on rounded;

res := acos(sqrt(2)/2); ⇒   RES := 0.785398163397

res-pi/4;               ⇒   0
```

Comments

An explicit numeric value is not given unless the switch `rounded` is on and the argument has an absolute numeric value less than or equal to 1.

11.2 ACOSH

ACOSH Operator

`acosh` represents the hyperbolic arccosine of its argument. It takes an arbitrary scalar expression as its argument. The derivative of `acosh` is known to the system. Numerical values may also be found by turning on the switch `rounded`.

`acosh`(*expression*) or `acosh` *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

```
acosh a;               =>   ACOSH(A)

acosh(0);              =>   ACOSH(0)
                                     4
                             2*SQRT(A  - 1)*A
df(acosh(a**2),a);     =>   ------------------
                                   4
                                  A  - 1
int(acosh(x),x);       =>   INT(ACOSH(X),X)
```

Comments

You may attach functionality by defining `acosh` to be the inverse of `cosh`. This is done by the commands

```
put('cosh,'inverse,'acosh);
put('acosh,'inverse,'cosh);
```

You can write a procedure to attach integrals or other functions to `acosh`. You may wish to add a check to see that its argument is properly restricted.

11.3 ACOT

ACOT Operator

`acot` represents the arccotangent of its argument. It takes an arbitrary scalar expression as its argument. The derivative of `acot` is known to the system. Numerical values may also be found by turning on the switch `rounded`.

`acot`(*expression*) or `acot` *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name. You can add functionality yourself with `let` and procedures.

11.4 ACOTH

ACOTH Operator

`acoth` represents the inverse hyperbolic cotangent of its argument. It takes an arbitrary scalar expression as its argument. The derivative of `acoth` is known to the system. Numerical values may also be found by turning on the switch `rounded`.

`acoth`(*expression*) or `acoth` *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name. You can add functionality yourself with `let` and procedures.

11.5 ACSC

ACSC — Operator

The `acsc` operator returns the arccosecant of its argument.

`acsc`(*expression*) or `acsc` *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

```
acsc(ab);                ⇒   ACSC(AB)

acsc 15;                 ⇒   ACSC(15)
                                      2  2
                              -SQRT(X *Y  - 1)
df(acsc(x*y),x);         ⇒   ------------------
                                    2  2
                                X*(X *Y  - 1)
on rounded;

res := acsc(2/sqrt(3));  ⇒   RES := 1.0471975512

res-pi/3;                ⇒   0
```

Comments

An explicit numeric value is not given unless the switch `rounded` is on and the argument has an absolute numeric value less than or equal to 1.

11.6 ACSCH

ACSCH Operator

The `acsch` operator returns the hyperbolic arccosecant of its argument.

`acsch`(*expression*) or `acsch` *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

```
acsch(ab);          =>   ACSCH(AB)

acsch 15;           =>   ACSCH(15)

                                  2  2
                          -SQRT(X *Y  + 1)
df(acsch(x*y),x);   =>   ------------------
                                2  2
                            X*(X *Y  + 1)

on rounded;

res := acsch(3);    =>   RES := 0.327450150237
```

Comments

An explicit numeric value is not given unless the switch `rounded` is on and the argument has an absolute numeric value less than or equal to 1.

11.7 ASEC

ASEC Operator

The `asec` operator returns the arccosecant of its argument.

`asec`(*expression*) or `asec` *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

```
asec(ab);             ⇒   ASEC(AB)

asec 15;              ⇒   ASEC(15)

                                  2  2
                            SQRT(X *Y  - 1)
df(asec(x*y),x);      ⇒   -----------------
                                 2  2
                             X*(X *Y  - 1)

on rounded;

res := asec sqrt(2);  ⇒   RES := 0.785398163397

res-pi/4;             ⇒   0
```

Comments

An explicit numeric value is not given unless the switch `rounded` is on and the argument has an absolute numeric value greater or equal to 1.

11.8 ASECH

ASECH Operator

`asech` represents the hyperbolic arccosecant of its argument. It takes an arbitrary scalar expression as its argument. The derivative of `asech` is known to the system. Numerical values may also be found by turning on the switch `rounded`.

`asech`(*expression*) or `asech` *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

```
asech a;                ⇒   ASECH(A)

asech(1);               ⇒   0
                                     4
                             2*SQRT(- A  + 1)
df(acosh(a**2),a);      ⇒   ------------------
                                  4
                              A*(A  - 1)
int(asech(x),x);        ⇒   INT(ASECH(X),X)
```

Comments

You may attach functionality by defining `asech` to be the inverse of `sech`. This is done by the commands

```
put('sech,'inverse,'asech);
put('asech,'inverse,'sech);
```

You can write a procedure to attach integrals or other functions to `asech`. You may wish to add a check to see that its argument is properly restricted.

11.9 ASIN

ASIN Operator

The `asin` operator returns the arcsine of its argument.

`asin`(*expression*) or `asin` *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

```
asin(givenangle);   =>   ASIN(GIVENANGLE)

asin(5);            =>   ASIN(5)

                                        2
                           2*SQRT( - 4*X  + 1))
df(asin(2*x),x);    =>   - ----------------------
                                      2
                                   4*X  - 1

on rounded;

asin .5;            =>   0.523598775598

asin(sqrt(3));      =>   ASIN(1.73205080757)

asin(sqrt(3)/2);    =>   1.04719755120
```

Comments

A numeric value is not returned by `asin` unless the switch `rounded` is on and its argument has an absolute value less than or equal to 1.

11.10 ASINH

ASINH Operator

The `asinh` operator returns the hyperbolic arcsine of its argument. The derivative of `asinh` and some simple transformations are known to the system.

`asinh`(*expression*) or `asinh` *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

```
asinh d;              ⇒   ASINH(D)

asinh(1);             ⇒   ASINH(1)
                                       2
                            2*SQRT(4*X  + 1))
df(asinh(2*x),x);     ⇒   -------------------
                                    2
                                 4*X  + 1
```

Comments

You may attach further functionality by defining `asinh` to be the inverse of `sinh`. This is done by the commands

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

A numeric value is not returned by `asinh` unless the switch `rounded` is on and its argument evaluates to a number.

11.11 ATAN

ATAN Operator

The atan operator returns the arctangent of its argument.

atan(*expression*) or atan *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

```
atan(middle);        =>   ATAN(MIDDLE)

on rounded;

atan 45;             =>   1.54857776147

off rounded;
                                                  2
                           2*ATAN(X)*X - LOG(X  + 1)
int(atan(x),x);      =>   ---------------------------
                                           2
                            2*Y
df(atan(y**2),y);    =>   --------
                            4
                           Y  + 1
```

Comments

A numeric value is not returned by atan unless the switch rounded is on and its argument evaluates to a number.

11.12 ATANH

ATANH Operator

The `atanh` operator returns the hyperbolic arctangent of its argument. The derivative of `asinh` and some simple transformations are known to the system.

`atanh`(*expression*) or `atanh` *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

```
atanh aa;           ⇒   ATANH(AA)

atanh(1);           ⇒   ATANH(1)
                            - X
df(atanh(x*y),y);   ⇒   -----------
                          2  2
                         X *Y  - 1
```

Comments

A numeric value is not returned by `asinh` unless the switch `rounded` is on and its argument evaluates to a number. You may attach additional functionality by defining `atanh` to be the inverse of `tanh`. This is done by the commands

```
put('tanh,'inverse,'atanh);
put('atanh,'inverse,'tanh);
```

11.13 ATAN2

ATAN2 Operator

`atan2`(*expression*, *expression*)

expression is any valid scalar REDUCE expression. In `rounded` mode, if a numerical value exists, `atan2` returns the principal value of the arc tangent of the second argument divided by the first in the range [-pi,+pi] radians, using the signs of both arguments to determine the quadrant of the return value. An expression in terms of `atan2` is returned in other cases.

Examples

`atan2(3,2);` ⇒ `ATAN2(3,2);`

`on rounded;`

`atan2(3,2);` ⇒ `0.982793723247`

`atan2(a,b);` ⇒ `ATAN2(A,B);`

`atan2(1,0);` ⇒ `1.57079632679`

Comments

`atan2` returns a numeric value only if `rounded` is on. Then `atan2` is calculated to the current degree of floating point precision.

11.14 COS

COS Operator

The `cos` operator returns the cosine of its argument.

`cos`(*expression*) or `cos` *simple_expression*

expression is any valid scalar REDUCE expression, *simple_expression* is a single identifier or begins with a prefix operator name.

Examples

`cos abc;`	⇒	`COS(ABC)`
`cos(pi);`	⇒	`-1`
`cos 4;`	⇒	`COS(4)`
`on rounded;`		
`cos(4);`	⇒	`- 0.653643620864`
`cos log 5;`	⇒	`- 0.0386319699339`

Comments

`cos` returns a numeric value only if `rounded` is on. Then the cosine is calculated to the current degree of floating point precision.

11.15 COSH

COSH Operator

The `cosh` operator returns the hyperbolic cosine of its argument. The derivative of `cosh` and some simple transformations are known to the system.

`cosh`(*expression*) or `cosh` *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

`cosh b;`	⇒	`COSH(B)`
`cosh(0);`	⇒	`1`
`df(cosh(x*y),x);`	⇒	`SINH(X*Y)*Y`
`int(cosh(x),x);`	⇒	`SINH(X)`

Comments

You may attach further functionality by defining its inverse (see `acosh`). A numeric value is not returned by `cosh` unless the switch `rounded` is on and its argument evaluates to a number.

11.16 COT

COT Operator

`cot` represents the cotangent of its argument. It takes an arbitrary scalar expression as its argument. The derivative of `acot` and some simple properties are known to the system.

`cot`(*expression*) or `cot` *simple_expression*

expression may be any scalar REDUCE expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

```
cot(a)*tan(a);     ⇒   COT(A)*TAN(A))

cot(1);            ⇒   COT(1)
                                     2
df(cot(2*x),x);    ⇒   - 2*(COT(2*X)  + 1)
```

Comments

Numerical values of expressions involving `cot` may be found by turning on the switch `rounded`.

11.17 COTH

COTH — Operator

The `coth` operator returns the hyperbolic cotangent of its argument. The derivative of `coth` and some simple transformations are known to the system.

`coth`(*expression*) or `coth` *simple_expression*

expression may be any scalar REDUCE expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

```
                                    2
df(coth(x*y),x);  ⇒   - Y*(COTH(X*Y)  - 1)

coth acoth z;     ⇒   Z
```

Comments

You can write `let` statements and procedures to add further functionality to `coth` if you wish. Numerical values of expressions involving `coth` may also be found by turning on the switch `rounded`.

11.18 CSC

CSC Operator

The `csc` operator returns the cosecant of its argument. The derivative of `csc` and some simple transformations are known to the system.

`csc`(*expression*) or `csc` *simple_expression*

expression may be any scalar REDUCE expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

```
csc(q)*sin(q);      ⇒   CSC(Q)*SIN(Q)

df(csc(x*y),x);     ⇒   -COT(X*Y)*CSC(X*Y)*Y
```

Comments

You can write `let` statements and procedures to add further functionality to `csc` if you wish. Numerical values of expressions involving `csc` may also be found by turning on the switch `rounded`.

11.19 CSCH

CSCH   Operator

The `cosh` operator returns the hyperbolic cosecant of its argument. The derivative of `csch` and some simple transformations are known to the system.

`csch`(*expression*) or `csch` *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

`csch b;`	⇒	`CSCH(B)`
`csch(0);`	⇒	`0`
`df(csch(x*y),x);`	⇒	`- COTH(X*Y)*CSCH(X*Y)*Y`
`int(csch(x),x);`	⇒	`INT(CSCH(X),X)`

Comments

A numeric value is not returned by `csch` unless the switch `rounded` is on and its argument evaluates to a number.

11.20 ERF

ERF — Operator

The `erf` operator represents the error function, defined by

$$erf(x) = \frac{2}{\sqrt{\pi}} \int e^{-x^2}\,dx$$

A limited number of its properties are known to the system, including the fact that it is an odd function. Its derivative is known, and from this, some integrals may be computed. However, a complete integration procedure for this operator is not currently included.

Examples

```
erf(0);              =>   0

erf(-a);             =>   - ERF(A)

                           4*SQRT(PI)*X
df(erf(x**2),x);     =>   --------------
                                4
                               X
                              E  *PI


                             2
                            X
                           E  *ERF(X)*PI*X + SQRT(PI)
int(erf(x),x);       =>   ----------------------------
                                       2
                                      X
                                     E  *PI
```

11.21 EXP

EXP Operator

The `exp` operator returns `e` raised to the power of its argument.

`exp`(*expression*) or `exp` *simple_expression*

expression can be any valid REDUCE scalar expression. *simple_expression* must be a single identifier or begin with a prefix operator.

Examples

```
                      SIN X
exp(sin(x));     =>  E
                      11
exp(11);         =>  E

on rounded;

exp sin(pi/3);   =>  2.37744267524
```

Comments

Numeric values are returned only when `rounded` is on. The single letter `e` with the exponential operator `^` or `**` may be substituted for `exp` without change of function.

11.22 SEC

SEC Operator

The `sec` operator returns the secant of its argument.

`sec`(*expression*) or `sec` *simple_expression*

expression is any valid scalar REDUCE expression, *simple_expression* is a single identifier or begins with a prefix operator name.

Examples

```
sec abc;      ⇒   SEC(ABC)

sec(pi);      ⇒   -1

sec 4;        ⇒   SEC(4)

on rounded;

sec(4);       ⇒   - 1.52988565647

sec log 5;    ⇒   - 25.8852966005
```

Comments

`sec` returns a numeric value only if `rounded` is on. Then the secant is calculated to the current degree of floating point precision.

11.23 SECH

SECH Operator

The `sech` operator returns the hyperbolic secant of its argument.

`sech`(*expression*) or `sech` *simple_expression*

expression is any valid scalar REDUCE expression, *simple_expression* is a single identifier or begins with a prefix operator name.

Examples

`sech abc;`	⇒	`SECH(ABC)`
`sech(0);`	⇒	`1`
`sech 4;`	⇒	`SECH(4)`
`on rounded;`		
`sech(4);`	⇒	`0.0366189934737`
`sech log 5;`	⇒	`0.384615384615`

Comments

`sech` returns a numeric value only if `rounded` is on. Then the expression is calculated to the current degree of floating point precision.

11.24 SIN

SIN Operator

The `sin` operator returns the sine of its argument.

`sin`(*expression*) or `sin` *simple_expression*

expression is any valid scalar REDUCE expression, *simple_expression* is a single identifier or begins with a prefix operator name.

Examples

```
sin aa;        ⇒   SIN(AA)

sin(pi/2);     ⇒   1

on rounded;

sin 3;         ⇒   0.14112000806

sin(pi/2);     ⇒   1.0
```

Comments

`sin` returns a numeric value only if `rounded` is on. Then the sine is calculated to the current degree of floating point precision. The argument in this case is assumed to be in radians.

11.25 SINH

SINH Operator

The `sinh` operator returns the hyperbolic sine of its argument. The derivative of `sinh` and some simple transformations are known to the system.

`sinh`(*expression*) or `sinh` *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

```
sinh b;              =>   SINH(B)

sinh(0);             =>   0
                                     2
df(sinh(x**2),x);    =>   2*COSH(X )*X
                           COSH(4*X)
int(sinh(4*x),x);    =>   -----------
                               4

on rounded;

sinh 4;              =>   27.2899171971
```

Comments

You may attach further functionality by defining its inverse (see `asinh`). A numeric value is not returned by `sinh` unless the switch `rounded` is on and its argument evaluates to a number.

11.26 TAN

TAN Operator

The `tan` operator returns the tangent of its argument.

`tan`(*expression*) or `tan` *simple_expression*

expression is any valid scalar REDUCE expression, *simple_expression* is a single identifier or begins with a prefix operator name.

Examples

```
tan a;                    ⇒   TAN(A)
                                    PI
tan(pi/5);                ⇒   TAN(----)
                                    5
on rounded; tan(pi/5);    ⇒   0.726542528005
```

Comments

`tan` returns a numeric value only if `rounded` is on. Then the tangent is calculated to the current degree of floating point accuracy.

When `rounded` is on, no check is made to see if the argument of `tan` is a multiple of $\pi/2$, for which the tangent goes to positive or negative infinity. (Of course, since REDUCE uses a fixed-point representation of $\pi/2$, it produces a large but not infinite number.) You need to make a check for multiples of $\pi/2$ in any program you use that might possibly ask for the tangent of such a quantity.

11.27 TANH

TANH Operator

The `tanh` operator returns the hyperbolic tangent of its argument. The derivative of `tanh` and some simple transformations are known to the system.

`tanh`(*expression*) or `tanh` *simple_expression*

expression may be any scalar REDUCE expression, not an array, matrix or vector expression. *simple_expression* must be a single identifier or begin with a prefix operator name.

Examples

```
tanh b;                 =>   TANH(B)

tanh(0);                =>   0
                                            2
df(tanh(x*y),x);        =>   Y*( - TANH(X*Y)  + 1)
                                  2*X
int(tanh(x),x);         =>   LOG(E    + 1) - X

on rounded; tanh 2;     =>   0.964027580076
```

Comments

You may attach further functionality by defining its inverse (see `atanh`). A numeric value is not returned by `tanh` unless the switch `rounded` is on and its argument evaluates to a number.

12 General Switches

12.1 SWITCHES

SWITCHES — Introduction

Switches are set on or off using the commands `on` or `off`, respectively. The default setting of the switches described in this section is `off` unless stated otherwise.

12.2 ALGINT

ALGINT   Switch

When the `algint` switch is on, the algebraic integration module (which must be loaded from the REDUCE library) is used for integration.

Comments

Loading `algint` from the library automatically turns on the `algint` switch. An error message will be given if `algint` is turned on when the `algint` has not been loaded from the library.

12.3 ALLBRANCH

ALLBRANCH Switch

When `allbranch` is on, the operator `solve` selects all branches of solutions. When `allbranch` is off, it selects only the principal branches. Default is `on`.

Examples

`solve(log(sin(x+3)),x);` ⇒

```
{X=2*ARBINT(1)*PI - ASIN(1) - 3,
 X=2*ARBINT(1)*PI + ASIN(1) + PI - 3}
```

`off allbranch;`

`solve(log(sin(x+3)),x);` ⇒ `X=ASIN(1) - 3`

Comments

`arbint`(1) indicates an arbitrary integer, which is given a unique identifier by REDUCE, showing that there are infinitely many solutions of this type. When `allbranch` is off, the single canonical solution is given.

12.4 ALLFAC

ALLFAC Switch

The `allfac` switch, when on, causes REDUCE to factor out automatically common products in the output of expressions. Default is `on`.

Examples

```
                                                   3
x + x*y**3 + x**2*cos(z);  =>   X*(COS(Z)*X + Y  + 1)

off allfac;
                                             2      3
x + x*y**3 + x**2*cos(z);  =>   COS(Z)*X  + X*Y  + X
```

Comments

The `allfac` switch has no effect when `pri` is off. Although the switch setting stays as it was, printing behavior is as if it were off.

12.5 ARBVARS

ARBVARS Switch

When `arbvars` is on, the solutions of singular or underdetermined systems of equations are presented in terms of arbitrary complex variables (see `arbcomplex`). Otherwise, the solution is parametrized in terms of some of the input variables. Default is `on`.

Examples

```
solve({2x + y,4x + 2y},{x,y});
```

⇒

```
               arbcomplex(1)
      {{x= - ---------------,y=arbcomplex(1)}}
                     2
```

```
solve({sqrt(x)+ y**3-1},{x,y});
```

```
                                      6       3
⇒    {{y=arbcomplex(2),x=y  - 2*y  + 1}}
```

```
off arbvars;
```

```
solve({2x + y,4x + 2y},{x,y});
```

```
               y
⇒    {{x= - ---}}
               2
```

```
solve({sqrt(x)+ y**3-1},{x,y});
```

```
            6       3
⇒    {{x=y  - 2*y  + 1}}
```

Comments

With `arbvars` off, the return value {{}} means that the equations given to `solve` imply no relation among the input variables.

12.6 BALANCED_MOD

BALANCED_MOD Switch

`modular` numbers are normally produced in the range [0,...n), where n is the current modulus. With `balanced_mod` on, the range [-n/2,n/2], or more precisely [-floor((n-1)/2), ceiling((n-1)/2)], is used instead.

Examples

```
setmod 7;          ⇒   1

on modular;

4;                 ⇒   4

on balanced_mod;

4;                 ⇒   -3
```

12.7 BFSPACE

BFSPACE Switch

Floating point numbers are normally printed in a compact notation (either fixed point or in scientific notation if `SCIENTIFIC_NOTATION` has been used). In some (but not all) cases, it helps comprehensibility if spaces are inserted in the number at regular intervals. The switch `bfspace`, if on, will cause a blank to be inserted in the number after every five characters.

Examples

```
on rounded;

1.2345678;   ⇒   1.2345678

on bfspace;

1.2345678;   ⇒   1.234 5678
```

Comments

`bfspace` is normally off.

12.8 COMBINEEXPT

COMBINEEXPT Switch

REDUCE is in general poor at surd simplification. However, when the switch `combineexpt` is on, the system attempts to combine exponentials whenever possible.

Examples

```
                                              1    1
                                             ---  ---
3^(1/2)*3^(1/3)*3^(1/6);  ⇒  SQRT(3)*3 3 *3 6

on combineexpt;

ws;                       ⇒  3
```

12.9 COMBINELOGS

COMBINELOGS Switch

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.

Examples

```
on expandlogs;

log(x*y);          ⇒   LOG(X) + LOG(Y)

on combinelogs;

ws;                ⇒   LOG(X*Y)
```

Comments

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.

12.10 COMP

COMP Switch

When `comp` is on, any succeeding function definitions are compiled into a faster-running form. Default is `off`.

Examples

The following procedure finds Fibonacci numbers recursively. Create a new file "refib" in your current directory with the following lines in it:

```
procedure refib(n);
   if fixp n and n >= 0 then
     if n <= 1 then 1
       else refib(n-1) + refib(n-2)
    else rederr "nonnegative integer only";
end;
```

Now load REDUCE and run the following:

`on time;`	⇒	`Time: 100 ms`
`in "refib"$`	⇒	`Time: 0 ms`
	⇒	REFIB
	⇒	`Time: 260 ms`
	⇒	`Time: 20 ms`
`refib(80);`	⇒	37889062373143906
	⇒	`Time: 14840 ms`
`on comp;`	⇒	`Time: 80 ms`
`in "refib"$`	⇒	`Time: 20 ms`
	⇒	REFIB
	⇒	`Time: 640 ms`
`refib(80);`	⇒	37889062373143906
	⇒	`Time: 10940 ms`

Comments

Note that the compiled procedure runs faster. Your time messages will differ depending upon which system you have. Compiled functions remain so for the duration of the REDUCE session, and are then lost. They must be recompiled if wanted in another session. With the switch `time` on as shown above, the CPU time used in executing the command is returned in milliseconds. Be careful not to leave `comp` on unless you want it, as it makes the processing of procedures much slower.

12.11 COMPLEX

COMPLEX Switch

When the `complex` switch is on, full complex arithmetic is used in simplification, function evaluation, and factorization. Default is `off`.

Examples

```
                                 2    2
factorize(a**2 + b**2);     ⇒   {{A  + B ,1}}

on complex;

factorize(a**2 + b**2);     ⇒   {{A + I*B,1},{A - I*B,1}}

(x**2 + y**2)/(x + i*y);    ⇒   X - I*Y

on rounded;                 ⇒

       *** Domain mode COMPLEX changed to COMPLEX_FLOAT

sqrt(-17);                  ⇒   4.12310562562*I

log(7*i);                   ⇒   1.94591014906 + 1.57079632679*I
```

Comments

Complex floating-point can be done by turning on `rounded` in addition to `complex`. With `complex` off however, REDUCE knows that i is the square root of -1 but will not carry out more complicated complex operations. If you want complex denominators cleared by multiplication by their conjugates, turn on the switch `rationalize`.

12.12 CREF

CREF Switch

The switch `cref` invokes the CREF cross-reference program that processes a set of procedure definitions to produce a summary of their entry points, undefined procedures, non-local variables and so on. The program will also check that procedures are called with a consistent number of arguments, and print a diagnostic message otherwise.

The output is alphabetized on the first seven characters of each function name.

To invoke the cross-reference program, `cref` is first turned on. This causes the program to load and the cross-referencing process to begin. After all the required definitions are loaded, turning `cref` off will cause a cross-reference listing to be produced.

Comments

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`.

12.13 CRAMER

CRAMER   Switch

When the `cramer` switch is on, `matrix` inversion and linear equation solving (operator `solve`) is done by Cramer's rule, through exterior multiplication. Default is `off`.

Examples

```
on time;                ⇒   Time: 80 ms

off output;             ⇒   Time: 100 ms


mm := mat((a,b,c,d,f),(a,a,c,f,b),(b,c,a,c,d), (c,c,a,b,f),
          (d,a,d,e,f));

                        ⇒   Time: 300 ms

inverse := 1/mm;        ⇒   Time: 18460 ms

on cramer;              ⇒   Time: 80 ms

cramersinv := 1/mm;     ⇒   Time: 9260 ms
```

Comments

Your time readings will vary depending on the REDUCE version you use. After you invert the matrix, turn on `output` and ask for one of the elements of the inverse matrix, such as `cramersinv(3,2)`, so that you can see the size of the expressions produced.

Inversion of matrices and the solution of linear equations with dense symbolic entries in many variables is generally considerably faster with `cramer` on. However, inversion of numeric-valued matrices is slower. Consider the matrices you're inverting before deciding whether to turn `cramer` on or off. A substantial portion of the time in matrix inversion is given to formatting the results for printing. To save this time, turn `output` off, as shown in this example or terminate the expression with a dollar sign instead of a semicolon. The results are still available to you in the workspace associated with your prompt number, or you can assign them to an identifier for further use.

12.14 DEFN

DEFN Switch

When the switch `defn` is on, the Standard Lisp equivalent of the input statement or procedure is printed, but not evaluated. Default is `off`.

Examples

```
on defn;

17/3;                  ⇒   (AEVAL (LIST 'QUOTIENT 17 3))

df(sin(x),x,2);        ⇒

       (AEVAL (LIST 'DF (LIST 'SIN 'X) 'X 2))

procedure coshval(a);
   begin scalar g;
      g := (exp(a) + exp(-a))/2;
      return g
   end;

                       ⇒

       (AEVAL
         (PROGN
           (FLAG '(COSHVAL) 'OPFN)
           (DE COSHVAL (A)
             (PROG (G)
               (SETQ G
                 (AEVAL
                    (LIST
                       'QUOTIENT
                       (LIST
                          'PLUS
                          (LIST 'EXP A)
                          (LIST 'EXP (LIST 'MINUS A)))
                       2)))
              (RETURN G)))) )
coshval(1);            ⇒   (AEVAL (LIST 'COSHVAL 1))
```

```
off defn;

coshval(1);              ⇒   Declare COSHVAL operator? (Y or N)

n

procedure coshval(a);
   begin scalar g;
      g := (exp(a) + exp(-a))/2;
      return g
   end;
                         ⇒   COSHVAL

on rounded;

coshval(1);              ⇒   1.54308063482
```

Comments

The above function `coshval` finds the hyperbolic cosine (cosh) of its argument. When `defn` is on, you can see the Standard Lisp equivalent of the function, but it is not entered into the system as shown by the message `Declare COSHVAL operator?`. It must be reentered with `defn` off to be recognized. This procedure is used as an example; a more efficient procedure would eliminate the unnecessary local variable with

```
procedure coshval(a);
   (exp(a) + exp(-a))/2;
```

12.15 DEMO

DEMO — Switch

The `demo` switch is used for interactive files, causing the system to pause after each command in the file until you type a [Return]. Default is `off`.

Comments

The switch `demo` has no effect on top level interactive statements. Use it when you want to slow down operations in a file so you can see what is happening.

You can either include the `on demo` command in the file, or enter it from the top level before bringing in any file. Unlike the `pause` command, `on demo` does not permit you to interrupt the file for questions of your own.

12.16 DFPRINT

DFPRINT   Switch

When dfprint is on, expressions in the differentiation operator df are printed in a more "natural" notation, with the differentiation variables appearing as subscripts. In addition, if the switch noarg is on (the default), the arguments of the differentiated operator are suppressed.

Examples

```
operator f;

df(f x,x);          ⇒    DF(F(X),X);

on dfprint;

ws;                 ⇒    F_X

df(f(x,y),x,y);     ⇒    F_X_,_Y

off noarg;

ws;                 ⇒    F(X,Y)_X
```

12.17 DIV

DIV Switch

When `div` is on, the system divides any simple factors found in the denominator of an expression into the numerator. Default is `off`.

Examples

```
on div;

                            2   -2
a := x**2/y**2;  =>   A := X  *Y

                          1  2   -2   -1
b := a/(3*z);    =>   B := ---*X  *Y   *Z
                          3

off div;

                         2
                        X
a;               =>   ----
                         2
                        Y
                           2
                          X
b;               =>   ---------
                          2
                       3*Y  *Z
```

Comments

The `div` switch only has effect when the `pri` switch is on. When `pri` is off, regardless of the setting of `div`, the printing behavior is as if `div` were off.

12.18 ECHO

ECHO Switch

The `echo` switch is normally off for top-level entry, and on when files are brought in. If `echo` is turned on at the top level, your input statements are echoed to the screen (thus appearing twice). Default `off` (but note default `on` for files).

Comments

If you want to display certain portions of a file and not others, use the commands `off echo` and `on echo` inside the file. If you want no display of the file, use the input command

`in` *filename*`$`

rather than using the semicolon delimiter.

Be careful when you use commands within a file to generate another file. Since `echo` is on for files, the output file echoes input statements (unlike its behavior from the top level). You should explicitly turn off `echo` when writing output, and turn it back on when you're done.

12.19 ERRCONT

ERRCONT Switch

When the `errcont` switch is on, error conditions do not stop file execution. Error messages will be printed whether `errcont` is on or off.

Default is `off`.

Comments

The table below shows REDUCE behavior under the settings of `errcont` and `int` :

Behavior in Case of Error in Files		
errcont	int	Behavior when errors in files are encountered
off	off	Message is printed and parsing continues, but no further statements are executed; no commands from keyboard accepted except `bye` or `end`
off	on	Message is printed, and you are asked if you wish to continue. (This is the default behavior)
on	off	Message is printed, and file continues to execute without pause
on	on	Message is printed, and file continues to execute without pause

12.20 EVALLHSEQP

EVALLHSEQP Switch

Under normal circumstances, the right-hand-side of an `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.

12.21 EXP

EXP Switch

When the `exp` switch is on, powers and products of expressions are expanded. Default is `on`.

Examples

```
                                3       2
(x+1)**3;                 =>   X  + 3*X  + 3*X + 1

(a + b*i)*(c + d*i);      =>   A*C + A*D*I + B*C*I - B*D

off exp;
                                      3
(x+1)**3;                 =>   (X + 1)

(a + b*i)*(c + d*i);      =>   (A + B*I)*(C + D*I)

length((x+1)**2/(y+1));   =>   2
```

Comments

Note that REDUCE knows that $i^2 = -1$. When `exp` is off, equivalent expressions may not simplify to the same form, although zero expressions still simplify to zero. Several operators that expect a polynomial argument behave differently when `exp` is off, such as `length`. Be cautious about leaving `exp` off.

12.22 EXPANDLOGS

EXPANDLOGS Switch

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.

Examples

```
on expandlogs;

log(x*y);          ⇒   LOG(X) + LOG(Y)

on combinelogs;

ws;                ⇒   LOG(X*Y)
```

Comments

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.

12.23 EZGCD

EZGCD Switch

When `ezgcd` and `gcd` are on, greatest common divisors are computed using the EZ GCD algorithm that uses modular arithmetic (and is usually faster). Default is `off`.

Comments

As a side effect of the gcd calculation, the expressions involved are factored, though not the heavy-duty factoring of `factorize`. The EZ GCD algorithm was introduced in a paper by J. Moses and D.Y.Y. Yun in *Proceedings of the ACM*, 1973, pp. 159-166.

Note that the `gcd` switch must also be on for `ezgcd` to have effect.

12.24 FACTOR

FACTOR   Switch

When the factor switch is on, input expressions and results are automatically factored.

Examples

```
on factor;

aa := 3*x**3*a + 6*x**2*y*a + 3*x**3*b + 6*x**2*y*b

+ x*y*a + 2*y**2*a + x*y*b + 2*y**2*b;
                                                  2
                         =>   AA := (A + B)*(3*X  + Y)*(X + 2*Y)

off factor;

aa;                      =>
            3         2                    2         3         2
       3*A*X  + 6*A*X *Y + A*X*Y + 2*A*Y  + 3*B*X  + 6*B*X *Y

+ B*X*Y + 2*B*Y^{2}

on factor;
                                    2
ab := x**2 - 2;          =>   AB := X  - 2
```

Comments

REDUCE factors univariate and multivariate polynomials with integer coefficients, finding any factors that also have integer coefficients. The factoring is done by reducing multivariate problems to univariate ones with symbolic coefficients, and then solving the univariate ones modulo small primes. The results of these calculations are merged to determine the factors of the original polynomial. The factorizer normally selects evaluation points and primes using a random number generator. Thus, the detailed factoring behavior may be different each time any particular problem is tackled.

When the factor switch is turned on, the exp switch is turned off, and when the factor switch is turned off, the exp switch is turned on, whether it was on previously or not.

When the switch `trfac` is on, informative messages are generated at each call to the factorizer. The `trallfac` switch causes the production of a more verbose trace message. It takes precedence over `trfac` if they are both on.

To factor a polynomial explicitly and store the results, use the operator `factorize`.

12.25 FAILHARD

FAILHARD — Switch

When the `failhard` switch is on, the integration operator `int` terminates with an error message if the integral cannot be done in closed terms. Default is off.

Comments

Use the `failhard` switch when you are dealing with complicated integrals and want to know immediately if REDUCE was unable to handle them. The integration operator sometimes returns a formal integration form that is more complicated than the original expression, when it is unable to complete the integration.

12.26 FORT

FORT — Switch

When fort is on, output is given Fortran-compatible syntax. Default is off.

Examples

```
on fort;

df(sin(7*x + y),x);          ⇒   ANS=7.*COS(7*X+Y)

on rounded;

b := log(sin(pi/5 + n*pi));  ⇒

       B=LOG(SIN(3.14159265359*N+0.628318530718))
```

Comments

REDUCE results can be written to a file (using out) and used as data by Fortran programs when fort is in effect. fort knows about correct statement length, continuation characters, defining a symbol when it is first used, and other Fortran details.

The GENTRAN package offers many more possibilities than the fort switch. It produces Fortran (or C or Ratfor) code from REDUCE procedures or structured specifications, including facilities for producing double precision output.

12.27 FORTUPPER

FORTUPPER Switch

When `fortupper` is on, any Fortran-style output appears in upper case. Default is `off`.

Examples

```
on fort;

df(sin(7*x + y),x);  ⇒   ans=7.*cos(7*x+y)

on fortupper;

df(sin(7*x + y),x);  ⇒   ANS=7.*COS(7*X+Y)
```

12.28 FULLPREC

FULLPREC — Switch

Trailing zeroes of rounded numbers to the full system precision are normally not printed. If this information is needed, for example to get a more understandable indication of the accuracy of certain data, the switch `fullprec` can be turned on.

Examples

```
on rounded;

1/2;           ⇒   0.5

on fullprec;

ws;            ⇒   0.500000000000
```

Comments

This is just an output options which neither influences the accuracy of the computation nor does it give additional information about the precision of the results. See also `scientific_notation`.

12.29 FULLROOTS

FULLROOTS Switch

Since roots of cubic and quartic polynomials can often be very messy, a switch `fullroots` controls the production of results in closed form. `solve` will apply the formulas for explicit forms for degrees 3 and 4 only if `fullroots` is `on`. Otherwise the result forms are built using `root_of`. Default is `off`.

12.30 GC

GC Switch

With the `gc` switch, you can turn the garbage collection messages on or off. The form of the message depends on the particular Lisp used for the REDUCE implementation.

Comments

See `reclaim` for an explanation of garbage collection. REDUCE does garbage collection when needed even if you have turned the notices off.

12.31 GCD

GCD Switch

When gcd is on, common factors in numerators and denominators of expressions are canceled. Default is off.

Examples

```
(2*(f*h)**2 - f**2*g*h - (f*g)**2 - f*h**3 + f*h*g**2
   - h**4 + g*h**3)/(f**2*h - f**2*g - f*h**2 + 2*f*g*h
   - f*g**2 - g*h**2 + g**2*h);
```

⇒

```
  2  2     2          2  2      2        3      3    4
 F *G  + F *G*H - 2*F *H  - F*G *H + F*H  - G*H  + H
-----------------------------------------------------
   2      2        2                    2    2        2
  F *G - F *H + F*G  - 2*F*G*H + F*H  - G *H + G*H
```

```
on gcd;
```

```
ws;
```

⇒

```
               2
 F*G + 2*F*H + H
-----------------
      F + G
```

```
e2 := a*c + a*d + b*c + b*d;
```

⇒ `E2 := A*C + A*D + B*C + B*D`

```
off exp;
```

```
e2;
```

⇒ `(A + B)*(C + D)`

Comments

Even with gcd off, a check is automatically made for common variable and numerical products in the numerators and denominators of expression, and the appropriate cancellations made. Thus the example demonstrating the use of gcd is somewhat complicated. Note when exp is off, gcd has the side effect of factoring the expression.

12.32 HORNER

HORNER Switch

When the horner switch is on, polynomial expressions are printed in Horner's form for faster and safer numerical evaluation. Default is off. The leading variable of the expression is selected as Horner variable. To select the Horner variable explicitly use the korder declaration.

Examples

```
on horner;

(13p-4q)^3;  ⇒
                    3             2
       ( - 64)*q  + p*(624*q  + p*(( - 2028)*q + p*2197))

korder q;

ws;          ⇒
                 3                  2
       2197*p  + q*(( - 2028)*p  + q*(624*p + q*(-64)))
```

12.33 IFACTOR

IFACTOR Switch

When the `ifactor` switch is on, any integer terms appearing as a result of the `factorize` command are factored themselves into primes. Default is `off`. If the argument of `factorize` is an integer, `ifactor` has no effect, since the integer is always factored.

Examples

```
factorize(4*x**2 + 28*x + 48);

                          ⇒   {{4,1},{X + 4,1},{X + 3,1}}

factorize(22587);         ⇒   {{3,1},{7529,1}}

on ifactor;

factorize(4*x**2 + 28*x + 48);

                          ⇒   {{2,2},{X + 4,1},{X + 3,1}}

factorize(22587);         ⇒   {{3,1},{7529,1}}
```

Comments

Constant terms that appear within nonconstant polynomial factors are not factored.

The `ifactor` switch affects only factoring done specifically with `factorize`, not on factoring done automatically when the `factor` switch is on.

12.34 INT

INT Switch

The `int` switch specifies an interactive mode of operation. Default `on`.

Comments

There is no reason to turn `int` off during interactive calculations, since there are no benefits to be gained. If you do have `int` off while inputting a file, and REDUCE finds an error, it prints the message "Continuing with parsing only." In this state, REDUCE accepts only `end;` or `bye;` from the keyboard; everything else is ignored, even the command `on int`.

12.35 INTSTR

INTSTR Switch

If intstr (for "internal structure") is on, arguments of an operator are printed in a more structured form.

Examples

```
operator f;

f(2x+2y);     ⇒   F(2*X + 2*Y)

on intstr;

ws;           ⇒   F(2*(X + Y))
```

12.36 LCM

LCM Switch

The `lcm` switch instructs REDUCE to compute the least common multiple of denominators whenever rational expressions occur. Default is `on`.

Examples

```
off lcm;

z := 1/(x**2 - y**2) + 1/(x-y)**2;
                                                2*X*(X - Y)
                              ⇒   Z := ----------------------------
                                          4      3           3    4
                                         X  - 2*X *Y + 2*X*Y  - Y
on lcm;
                                           2*X*(X - Y)
z;                            ⇒   ----------------------------
                                    4      3           3    4
                                   X  - 2*X *Y + 2*X*Y  - Y
zz := 1/(x**2 - y**2) + 1/(x-y)**2;
                                                   2*X
                              ⇒   ZZ := -----------------------
                                           3    2        2    3
                                          X  - X *Y - X*Y  + Y
on gcd;
                                               2*X
z;                            ⇒   -----------------------
                                    3    2        2    3
                                   X  - X *Y - X*Y  + Y
```

Comments

Note that `lcm` has effect only when rational expressions are first combined. It does not examine existing structures for simplifications on display. That is shown above when z is entered with `lcm` off. It remains unsimplified even after `lcm` is turned back on. However, a new variable containing the same expression is simplified on entry. The switch `gcd` does examine existing structures, as shown in the last example line above.

Full greatest common divisor calculations become expensive if work with large rational expressions is required. A considerable savings of time can be had if a full gcd check is made only when denominators are combined, and only a partial check for numerators. This is the effect of the `lcm` switch.

12.37 LESSSPACE

LESSSPACE Switch

You can turn on the switch `lessspace` if you want fewer blank lines in your output.

12.38 LIMITEDFACTORS

LIMITEDFACTORS Switch

To get limited factorization in cases where it is too expensive to use full multivariate polynomial factorization, the switch `limitedfactors` can be turned on. In that case, only "inexpensive" factoring operations, such as square-free factorization, will be used when `factorize` is called.

Examples

```
a := (y-x)^2*(y^3+2x*y+5)*(y^2-3x*y+7)
                                        2
factorize a;            ⇒   {- 3*X*Y + Y  + 7,1}

                                      3
                            {2*X*Y + Y  + 5,1},
                            {X - Y,2}}
on limitedfactors;

factorize a;            ⇒
            2  2        4        3          5      3      2
      {- 6*X *Y  - 3*X*Y  + 2*X*Y  - X*Y + Y  + 7*Y  + 5*Y  + 35,1},
      {X - Y,2}}
```

12.39 LIST

LIST Switch

The `list` switch causes REDUCE to print each term in any sum on separate lines.

Examples

```
x**2*(y**2 + 2*y) + x*(y**2 + z)/(2*a);

                                         2               2
                           =>   X*(2*A*X*Y  + 4*A*X*Y + Y  + Z)
                                --------------------------------
                                              2*A

on list;

                                          2
ws;                        =>   (X*(2*A*X*Y
                                  + 4*A*X*Y

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

12.40 LISTARGS

LISTARGS Switch

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. This process can be inhibited globally by turning on the switch `listargs`.

Examples

`log {a,b,c};` ⇒ `LOG(A),LOG(B),LOG(C)`

`on listargs;`

`log {a,b,c};` ⇒ `LOG(A,B,C)`

Comments

It is possible to inhibit such distribution for a specific operator by using the declaration `listargp`. In addition, if an operator has more than one argument, no such distribution occurs, so `listargs` has no effect.

12.41 MCD

MCD Switch

When `mcd` is on, sums and differences of rational expressions are put on a common denominator. Default is `on`.

Examples

```
                       5*A + B*X + B
a/(x+1) + b/5;   ⇒    ---------------
                         5*(X + 1)

off mcd;

                               -1
a/(x+1) + b/5;   ⇒    (X + 1)   *A + 1/5*B

1/6 + 1/7;       ⇒    13/42
```

Comments

Even with `mcd` off, rational expressions involving only numbers are still put over a common denominator.

Turning `mcd` off is useful when explicit negative powers are needed, or if no greatest common divisor calculations are desired, or when differentiating complicated rational expressions. Results when `mcd` is off are no longer in canonical form, and expressions equivalent to zero may not simplify to 0. Some operations, such as factoring cannot be done while `mcd` is off. This option should therefore be used with some caution. Turning `mcd` off is most valuable in intermediate parts of a complicated calculation, and should be turned back on for the last stage.

12.42 MODULAR

MODULAR Switch

When `modular` is on, polynomial coefficients are reduced by the modulus set by `setmod`. If no modulus has been set, `modular` has no effect.

Examples

```
setmod 2;                 ⇒   1

on modular;
                                2     2
(x+y)**2;                 ⇒   X  + Y

145*x**2 + 20*x**3 + 17 + 15*x*y;
                                2
                          ⇒   X  + X*Y + 1
```

Comments

Modular operations are only conducted on the coefficients, not the exponents. The modulus is not restricted to being prime. When the modulus is prime, division by a number not relatively prime to the modulus results in a *Zero divisor* error message. When the modulus is a composite number, division by a power of the modulus results in an error message, but division by an integer which is a factor of the modulus does not. The representation of modular number can be influenced by `balanced_mod`.

12.43 MSG

MSG Switch

When `msg` is off, the printing of warning messages is suppressed. Error messages are still printed.

Comments

Warning messages include those about redimensioning an `array` or declaring an `operator` where one is expected.

12.44 MULTIPLICITIES

MULTIPLICITIES Switch

When solve is applied to a set of equations with multiple roots, solution multiplicities are normally stored in the global variable root_multiplicities rather than the solution list. If you want the multiplicities explicitly displayed, the switch multiplicities should be turned on. In this case, root_multiplicities has no value.

Examples

```
solve(x^2=2x-1,x);      ⇒  X=1

root_multiplicities;    ⇒  2

on multiplicities;

solve(x^2=2x-1,x);      ⇒  X=1,X=1

root_multiplicities;    ⇒
```

12.45 NAT

NAT Switch

When `nat` is on, output is printed to the screen in natural form, with raised exponents. `nat` should be turned off when outputting expressions to a file for future input. Default is `on`.

Examples

```
                      3       2            2    3
(x + y)**3;  ⇒   X  + 3*X *Y + 3*X*Y  + Y

off nat;

(x + y)**3;  ⇒   X**3 + 3*X**2*Y + 3*X*Y**2 + Y**3$

on fort;

(x + y)**3;  ⇒   ANS=X**3+3.*X**2*Y+3.*X*Y**2+Y**3
```

Comments

With `nat` off, a dollar sign is printed at the end of each expression. An output file written with `nat` off is ready to be read into REDUCE using the command `in`.

12.46 NERO

NERO — Switch

When `nero` is on, zero assignments (such as matrix elements) are not printed.

Examples

```
matrix a; a := mat((1,0),(0,1));
```

⇒
```
A(1,1) := 1
A(1,2) := 0
A(2,1) := 0
A(2,2) := 1
```

```
on nero;
```

`a;` ⇒
```
MAT(1,1) := 1
MAT(2,2) := 1
```

`a(1,2);` ⇒

nothing is printed.

`b := 0;` ⇒

nothing is printed.

```
off nero;
```

`b := 0;` ⇒ `B := 0`

Comments

`nero` is often used when dealing with large sparse matrices, to avoid being overloaded with zero assignments.

12.47 NOARG

NOARG Switch

When `dfprint` is on, expressions in the differentiation operator `df` are printed in a more "natural" notation, with the differentiation variables appearing as subscripts. When `noarg` is on (the default), the arguments of the differentiated operator are also suppressed.

Examples

```
operator f;

df(f x,x);     ⇒   DF(F(X),X);

on dfprint;

ws;            ⇒   F_X

off noarg;

ws;            ⇒   F(X)_X
```

12.48 NOLNR

NOLNR Switch

When `nolnr` is on, the linear properties of the integration operator `int` are suppressed if the integral cannot be found in closed terms.

Comments

REDUCE uses the linear properties of integration to attempt to break down an integral into manageable pieces. If an integral cannot be found in closed terms, these pieces are returned. When the `nolnr` switch is off, as many of the pieces as possible are integrated. When it is on, if any piece fails, the rest of them remain unevaluated.

12.49 NOSPLIT

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.

12.50 NUMVAL

NUMVAL Switch

With `rounded` on, elementary functions with numerical arguments will return a numerical answer where appropriate. If you wish to inhibit this evaluation, `numval` should be turned off. It is normally on.

Examples

```
on rounded;

cos 3.4;      ⇒   - 0.966798192579

off numval;

cos 3.4;      ⇒   COS(3.4)
```

12.51 OUTPUT

OUTPUT Switch

When `output` is off, no output is printed from any REDUCE calculation. The calculations have their usual effects other than printing. Default is `on`.

Comments

Turn output `off` if you do not wish to see output when executing large files, or to save the time REDUCE spends formatting large expressions for display. Results are still available with `ws`, or in their assigned variables.

12.52 OVERVIEW

OVERVIEW Switch

When overview is on, the amount of detail reported by the factorizer switches trfac and trallfac is reduced.

12.53 PERIOD

PERIOD Switch

When `period` is on, periods are added after integers in Fortran-compatible output (when `fort` is on). There is no effect when `fort` is off. Default is `on`.

12.54 PRECISE

PRECISE Switch

When the `precise` switch is on, simplification of roots of even powers returns absolute values, a more precise answer mathematically. Default is `on`.

Examples

```
sqrt(x**2);        ⇒   X

(x**2)**(1/4);     ⇒   SQRT(X)

on precise;

sqrt(x**2);        ⇒   ABS(X)

(x**2)**(1/4);     ⇒   SQRT(ABS(X))
```

Comments

In many types of mathematical work, simplification of powers and surds can proceed by the fastest means of simplifying the exponents arithmetically. When it is important to you that the positive root be returned, turn `precise` on. One situation where this is important is when graphing square-root expressions such as $\sqrt{x^2 + y^2}$ to avoid a spike caused by REDUCE simplifying $\sqrt{y^2}$ to y when x is zero.

12.55 PRET

PRET Switch

When `pret` is on, input is printed in standard REDUCE format and then evaluated.

Examples

```
on pret;

(x+1)^3;                 ⇒    (x + 1)**3;

                               3      2
                              X  + 3*X  + 3*X + 1

procedure fac(n);
   if not (fixp(n) and n>=0)
     then rederr "Choose nonneg. integer only"
    else for i := 0:n-1 product i+1;

                         ⇒

       procedure fac n;
          if not (fixp n and n>=0)
            then rederr "Choose nonneg. integer only"
           else for i := 0:n - 1 product i + 1;
       FAC
fac 5;                   ⇒   fac 5;
                             120
```

Comments

Note that all input is converted to lower case except strings (which keep the same case) all operators with a single argument have had the parentheses removed, and all infix operators have had a space added on each side. In addition, syntactical constructs like `if`...`then`...`else` are printed in a standard format.

12.56 PRI

PRI Switch

When `pri` is on, the declarations `order` and `factor` can be used, and the switches `allfac`, `div`, `rat`, and `revpri` take effect when they are on. Default is `on`.

Comments

Printing of expressions is faster with `pri` off. The expressions are then returned in one standard form, without any of the display options that can be used to feature or display various parts of the expression. You can also gain insight into REDUCE's representation of expressions with `pri` off.

12.57 RAISE

RAISE Switch

When `raise` is on, lower case letters are automatically converted to upper case on input. `raise` is normally on.

Comments

This conversion affects the internal representation of the letter, and is independent of the case with which a letter is printed, which is normally lower case.

12.58 RAT

RAT   Switch

When the `rat` switch is on, and kernels have been selected to display with the `factor` declaration, the denominator is printed with each term rather than one common denominator at the end of an expression.

Examples

```
(x+1)/x + x**2/sin y;  ⇒
                                 3
        SIN(Y)*X + SIN(Y) + X
       ------------------------   factor x;
               SIN(Y)*X
                              3
                             X  + X*SIN(Y) + SIN(Y)
(x+1)/x + x**2/sin y;  ⇒    ------------------------   on rat;
                                    X*SIN(Y)
                               2
                              X            -1
(x+1)/x + x**2/sin y;  ⇒    -------- + 1 + X
                             SIN(Y)
```

Comments

The `rat` switch only has effect when the `pri` switch is on. When `pri` is off, regardless of the setting of `rat`, the printing behavior is as if `rat` were off. `rat` only has effect upon the display of expressions, not their internal form.

12.59 RATARG

RATARG Switch

When `ratarg` is on, rational expressions can be given to operators such as `coeff` and `lterm` that normally require polynomials in one of their arguments. When `ratarg` is off, rational expressions cause an error message.

Examples

```
aa := x/y**2 + 1/x + y/x**2;
```

```
                                  3      2    3
                                 X  + X*Y  + Y
                     =>   AA := ----------------
                                      2  2
                                     X *Y
```

```
coeff(aa,x);         =>
```

```
              3      2    3
             X  + X*Y  + Y
      ***** ---------------- invalid as POLYNOMIAL
                  2  2
                 X *Y
```

```
on ratarg;
```

```
                              Y    1       1
coeff(aa,x);         =>   {----,----,0,-------}
                            2    2       2  2
                           X    X       X *Y
```

12.60 RATIONAL

RATIONAL Switch

When `rational` is on, polynomial expressions with rational coefficients are produced.

Examples

```
                          2*X + 3*Y
x/2 + 3*y/4;         ⇒   -----------
                              4
                           2
                          X  + 5*X + 17
(x**2 + 5*x + 17)/2; ⇒   ---------------
                                2
on rational;
                          1         3
x/2 + 3y/4;          ⇒   ---*(X + ---*Y)
                          2         2
                          1     2
(x**2 + 5*x + 17)/2; ⇒   ---*(X  + 5*X + 17)
                          2
```

Comments

By using `rational`, polynomial expressions with rational coefficients can be used in some commands that expect polynomials. With `rational` off, such a polynomial becomes a rational expression, with denominator the least common multiple of the denominators of the rational number coefficients.

12.61 RATIONALIZE

RATIONALIZE Switch

When the `rationalize` switch is on, denominators of rational expressions that contain complex numbers or root expressions are simplified by multiplication by their conjugates.

Examples

```
qq := (1+sqrt(3))/(sqrt(3)-7);
                                      SQRT(3) + 1
                          ⇒   QQ := -------------
                                      SQRT(3) - 7

on rationalize;
                                - 4*SQRT(3) - 5
qq;                       ⇒   -----------------
                                      23
                                 2/3       1/3
                                6    - 4*6    + 16
2/(4 + 6**(1/3));         ⇒   --------------------
                                        35
                                2*I - 1
(i-1)/(i+3);              ⇒   ---------
                                   5

off rationalize;
                                I - 1
(i-1)/(i+3);              ⇒   -------
                                I + 3
```

12.62 RATPRI

RATPRI — Switch

When the `ratpri` switch is on, rational expressions and fractions are printed as two lines separated by a fraction bar, rather than in a linear style. Default is `on`.

Examples

```
3/17;         =>     3
                    ----
                     17

2/b + 3/y;    =>     3*B + 2*Y
                    -----------
                        B*Y

off ratpri;

3/17;         =>    3/17

2/b + 3/y;    =>    (3*B + 2*Y)/(B*Y)
```

12.63 REVPRI

REVPRI — Switch

When the `revpri` switch is on, terms are printed in reverse order from the normal printing order.

Examples

```
                                         5     2
x**5 + x**2 + 18 + sqrt(y);   =>   SQRT(Y) + X  + X  + 18

a + b + c + w;                =>   A + B + C + W

on revpri;
                                      2     5
x**5 + x**2 + 18 + sqrt(y);   =>   17 + X  + X  + SQRT(Y)

a + b + c + w;                =>   W + C + B + A
```

Comments

Turn `revpri` on when you want to display a polynomial in ascending rather than descending order.

12.64 RLISP88

RLISP88 — Switch

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). It supports different looping constructs from the traditional Rlisp, and treats "-" as a letter unless separated by spaces. Turning on the switch `rlisp88` converts to Rlisp '88 parsing conventions in symbolic mode, and enables the use of Rlisp '88 extensions. Turning off the switch reverts to the traditional Rlisp and the previous mode ((`symbolic` or `algebraic`) in force before `rlisp88` was turned on.

12.65 ROUNDALL

ROUNDALL Switch

In `rounded` mode, rational numbers are normally converted to a floating point representation. If `roundall` is off, this conversion does not occur. `roundall` is normally `on`.

Examples

```
on rounded;

1/2;            ⇒   0.5

off roundall;
         {
1/2;    ---}{2}   ⇒
         1
```

12.66 ROUNDBF

ROUNDBF Switch

When `rounded` is on, the normal defaults cause underflows to be converted to zero. If you really want the small number that results in such cases, `roundbf` can be turned on.

Examples

`on rounded;`

`exp(-100000.1^2);` ⇒ `0`

`on roundbf;`

`exp(-100000.1^2);` ⇒ `1.18441281937E-4342953505`

Comments

If a polynomial is input in `rounded` mode at the default precision into any `roots` function, and it is not possible to represent any of the coefficients of the polynomial precisely in the system floating point representation, the switch `roundbf` will be automatically turned on. All rounded computation will use the internal bigfloat representation until the user subsequently turns `roundbf` off. (A message is output to indicate that this condition is in effect.)

12.67 ROUNDED

ROUNDED — Switch

When `rounded` is on, floating-point arithmetic is enabled, with precision initially at a system default value, which is usually 12 digits. The precise number can be found by the command `precision`(0).

Examples

```
pi;           ⇒   PI
                   5
35/217;       ⇒   ----
                   31
on rounded;
pi;           ⇒   3.14159265359
35/217;       ⇒   0.161
sqrt(3);      ⇒   1.73205080756
```

Comments

If more than the default number of decimal places are required, use the `precision` command to set the required number.

12.68 SAVESTRUCTR

SAVESTRUCTR Switch

When `savestructr` is on, results of the `structr` command are returned as a list whose first element is the representation for the expression and the remaining elements are equations showing the relationships of the generated variables.

Examples

```
off exp;

structr((x+y)^3 + sin(x)^2);

                       ⇒   ANS3
                              where

                                                3       2
                                 ANS3 := ANS1  + ANS2
                                 ANS2 := SIN(X)
                                 ANS1 := X + Y
ans3;                  ⇒   ANS3

on savestructr;

structr((x+y)^{3} + sin(x)^{2});

                       ⇒
                      3       2
       ANS3,ANS3=ANS1  + ANS2 ,ANS2=SIN(X),ANS1=X + Y
                                 3         2
ans3 where rest ws;    ⇒   (X + Y)  + SIN(X)
```

Comments

In normal operation, `structr` is only a display command. With `savestructr` on, you can access the various parts of the expression produced by `structr`.

The generic system names use the stem `ANS`. You can change this to your own stem by the command `varname`. REDUCE adds integers to this stem to make unique identifiers.

12.69 SOLVESINGULAR

SOLVESINGULAR Switch

When `solvesingular` is on, singular or underdetermined systems of linear equations are solved, using arbitrary real, complex or integer variables in the answer. Default is `on`.

Examples

```
solve({2x + y,4x + 2y},{x,y});
```

⇒

```
        ARBCOMPLEX(1)
{{X= - ---------------,Y=ARBCOMPLEX(1)}}
              2
```

```
solve({7x + 15y - z,x - y - z},{x,y,z});
```

⇒

```
   8*ARBCOMPLEX(3)
{{X=-----------------
          11

       3*ARBCOMPLEX(3)
  Y= - -----------------
             11

  Z=ARBCOMPLEX(3)}}
```

```
off solvesingular;
```

```
solve({2x + y,4x + 2y},{x,y});
```

⇒

```
***** SOLVE given singular equations
```

```
solve({7x + 15y - z,x - y - z},{x,y,z});
```

⇒

```
***** SOLVE given singular equations
```

Comments

The integer following the identifier `arbcomplex` above is assigned by the system, and serves to identify the variable uniquely. It has no other significance.

12.70 TIME

TIME Switch

When `time` is on, the system time used in executing each REDUCE statement is printed after the answer is printed.

Examples

```
on time;              ⇒   Time: 4940 ms
                                      2
df(sin(x**2 + y),y);  ⇒   COS(X  + Y )
                          Time: 180 ms
solve(x**2 - 6*y,x);  ⇒   {X= - SQRT(Y)*SQRT(6),
                           X=SQRT(Y)*SQRT(6)}
                          Time: 320 ms
```

Comments

When `time` is first turned on, the time since the beginning of the REDUCE session is printed. After that, the time used in computation, (usually in milliseconds, though this is system dependent) is printed after the results of each command. Idle time or time spent typing in commands is not counted. If `time` is turned off, the first reading after it is turned on again gives the time elapsed since it was turned off. The time printed is CPU or wall clock time, depending on the system.

12.71 TRALLFAC

TRALLFAC Switch

When `trallfac` is on, a more detailed trace of factorizer calls is generated.

Comments

The `trallfac` switch takes precedence over `trfac` if they are both on. `trfac` gives a factorization trace with less detail in it. When the `factor` switch is on also, all input polynomials are sent to the factorizer automatically and trace information is generated. The `out` command saves the results of the factoring, but not the trace.

12.72 TRFAC

TRFAC Switch

When `trfac` is on, a narrative trace of any calls to the factorizer is generated. Default is `off`.

Comments

When the switch `factor` is on, and `trfac` is on, every input polynomial is sent to the factorizer, and a trace generated. With `factor` off, only polynomials that are explicitly factored with the command `factorize` generate trace information.

The `out` command saves the results of the factoring, but not the trace. The `trallfac` switch gives trace information to a greater level of detail.

12.73 TRIGFORM

TRIGFORM Switch

When `fullroots` is on, `solve` will compute the roots of a cubic or quartic polynomial is closed form. When `trigform` is on, the roots will be expressed by trigonometric forms. Otherwise nested surds are used. Default is `on`.

12.74 TRINT

TRINT — Switch

When `trint` is on, a narrative tracing various steps in the integration process is produced.

Comments

The `out` command saves the results of the integration, but not the trace.

12.75 TRNONLNR

TRNONLNR Switch

When trnonlnr is on, a narrative tracing various steps in the process for solving non-linear equations is produced.

Comments

trnonlnr can only be used after the solve package has been loaded (e.g., by an explicit call of load_package). The out command saves the results of the equation solving, but not the trace.

12.76 VAROPT

VAROPT Switch

When `varopt` is on, the sequence of variables is optimized by `solve` with respect to execution speed. Otherwise, the sequence given in the call to `solve` is preserved. Default is `on`.

In combination with the switch `arbvars`, `varopt` can be used to control variable elimination.

Examples

```
off arbvars;

                                          x         x
solve({x+2z,x-3y},{x,y,z});  ⇒   {{y=---,z= - ---}}
                                          3         2
                                                1
solve({x*y=1,z=x},{x,y,z});  ⇒   {{z=x,y=---}}
                                                x
off varopt;
                                                        2*z
solve({x+2z,x-3y},{x,y,z});  ⇒   {{x= - 2*z,y= - -----}}
                                                         3
                                          1
solve({x*y=1,z=x},{x,y,z});  ⇒   {{y=---,x=z}}
                                          z
```

13 Matrix Operations

13.1 COFACTOR

COFACTOR Operator

The operator `cofactor` returns the cofactor of the element in row *row* and column *column* of a `matrix`. Errors occur if *row* or *column* do not evaluate to integer expressions or if the matrix is not square.

`cofactor`($matrix\_expression, row, column$)

Examples

```
cofactor(mat((a,b,c),(d,e,f),(p,q,r)),2,2);
```

⇒ `A*R - C*P`

```
cofactor(mat((a,b,c),(d,e,f)),1,1);
```

⇒ `***** non-square matrix`

13.2 DET

DET Operator

The `det` operator returns the determinant of its (square `matrix`) argument.

`det`(*expression*) or `det` *expression*

expression must evaluate to a square matrix.

Examples

```
matrix m,n;

m := mat((a,b),(c,d));   ⇒   M(1,1) := A
                             M(1,2) := B
                             M(2,1) := C
                             M(2,2) := D

det m;                   ⇒   A*D - B*C

n := mat((1,2),(1,2));   ⇒   N(1,1) := 1
                             N(1,2) := 2
                             N(2,1) := 1
                             N(2,2) := 2

det(n);                  ⇒   0

det(5);                  ⇒   5
```

Comments

Given a numerical argument, `det` returns the number. However, given a variable name that has not been declared of type matrix, or a non-square matrix, `det` returns an error message.

13.3 MAT

MAT — Operator

The `mat` operator is used to represent a two-dimensional `matrix`.

`mat`((*expr*{,*expr*}*){(*expr*{,*expr*}*)}*)

expr may be any valid REDUCE scalar expression.

Examples

```
mat((1,2),(3,4));          ⇒  MAT(1,1) := 1
                              MAT(2,3) := 2
                              MAT(2,1) := 3
                              MAT(2,2) := 4

mat(2,1);                  ⇒  ***** Matrix mismatch
                              Cont? (Y or N)

matrix qt;

qt := ws;                  ⇒  QT(1,1) := 1
                              QT(1,2) := 2
                              QT(2,1) := 3
                              QT(2,2) := 4

matrix a,b;

a := mat((x),(y),(z));     ⇒  A(1,1) := X
                              A(2,1) := Y
                              A(3,1) := Z

b := mat((sin x,cos x,1)); ⇒  B(1,1) := SIN(X)
                              B(1,2) := COS(X)
                              B(1,3) := 1
```

Comments

Matrices need not have a size declared (unlike arrays). `mat` redimensions a matrix variable as needed. It is necessary, of course, that all rows be the same length. An anonymous matrix, as shown in the first example, must be named before it can be referenced (note error message). When using `mat` to fill a $1 \times n$ matrix, the row of values must be inside a second set of parentheses, to eliminate ambiguity.

13.4 MATEIGEN

MATEIGEN Operator

The `mateigen` operator calculates the eigenvalue equation and the corresponding eigenvectors of a `matrix`.

`mateigen`$(matrix - id, tag - id)$

matrix-id must be a declared matrix of values, and *tag-id* must be a legal REDUCE identifier.

Examples

```
aa := mat((2,5),(1,0))$

                                    2
mateigen(aa,alpha);        ⇒   {{ALPHA  - 2*ALPHA - 5,
                                 1,

                                              5*ARBCOMPLEX(1)
                                 MAT(1,1) := -----------------,
                                                 ALPHA - 2

                                 MAT(2,1) := ARBCOMPLEX(1)
                                 }}
                                                 2
charpoly := first first ws;  ⇒   CHARPOLY := ALPHA  - 2*ALPHA - 5

bb := mat((1,0,1),(1,1,0),(0,0,1))$

mateigen(bb,lamb);         ⇒   {{LAMB - 1,3,
                                 [      0      ]
                                 [ARBCOMPLEX(2)]
                                 [      0      ]
                                 }}
```

Comments

The `mateigen` operator returns a list of lists of three elements. The first element is a square free factor of the characteristic polynomial; the second element is its multiplicity; and the third element is the corresponding eigenvector. If the characteristic polynomial can be completely factored, the product of the first elements of all the sublists will produce the minimal polynomial. You can access the various parts of the answer with the usual list access operators.

If the matrix is degenerate, more than one eigenvector can be produced for the same eigenvalue, as shown by more than one arbitrary variable in the eigenvector. The identification numbers of the arbitrary complex variables shown in the examples above may not be the same as yours. Note that since `lambda` is a reserved word in REDUCE, you cannot use it as a *tag-id* for this operator.

13.5 MATRIX

MATRIX Declaration

Identifiers are declared to be of type `matrix`.

`matrix` *identifier* &option (*index*, *index*)
{,*identifier* &option (*index*, *index*)}*

identifier must not be an already-defined operator or array or the name of a scalar variable. Dimensions are optional, and if used appear inside parentheses. *index* must be a positive integer.

Examples

```
matrix a,b(1,4),c(4,4);

b(1,1);                      ⇒   0

a(1,1);                      ⇒   ***** Matrix A not set

a := mat((x0,y0),(x1,y1));   ⇒   A(1,1) := X0
                                 A(1,2) := Y0
                                 A(2,1) := X0
                                 A(2,2) := X1
length a;                    ⇒   {2,2}
                                              2
b := a**2;                   ⇒   B(1,1) := X0  + X1*Y0
                                 B(1,2) := Y0*(X0 + Y1)
                                 B(2,1) := X1*(X0 + Y1)

                                                      2
                                 B(2,2) := X1*Y0 + Y1
```

Comments

When a matrix variable has not been dimensioned, matrix elements cannot be referenced until the matrix is set by the `mat` operator. When a matrix is dimensioned in its declaration, matrix elements are set to 0. Matrix elements cannot stand for themselves. When you use `let` on a matrix element, there is no effect unless the element contains a constant, in which case an error message is returned. The same behavior occurs with `clear`. Do *not* use `clear` to try to set a matrix element to 0. `let` statements can be applied to matrices as a whole, if the right-hand side of the expression is a matrix expression, and the left-hand side identifier has been declared to be a matrix.

Arithmetical operators apply to matrices of the correct dimensions. The operators `+` and `-` can be used with matrices of the same dimensions. The operator `*` can be used to multiply $m \times n$ matrices by $n \times p$ matrices. Matrix multiplication is non-commutative. Scalars can also be multiplied with matrices, with the result that each element of the matrix is multiplied by the scalar. The operator `/` applied to two matrices computes the first matrix multiplied by the inverse of the second, if the inverse exists, and produces an error message otherwise. Matrices can be divided by scalars, which results in dividing each element of the matrix. Scalars can also be divided by matrices when the matrices are invertible, and the result is the multiplication of the scalar by the inverse of the matrix. Matrix inverses can by found by `1/A` or `/A`, where `A` is a matrix. Square matrices can be raised to positive integer powers, and also to negative integer powers if they are nonsingular.

When a matrix variable is assigned to the results of a calculation, the matrix is redimensioned if necessary.

13.6 NULLSPACE

NULLSPACE Operator

nullspace(*matrix_expression*)

nullspace calculates for its matrix argument, 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.

Examples

```
nullspace mat((1,2,3,4),(5,6,7,8));
```

```
⇒          {
             [ 1  ]
             [    ]
             [ 0  ]
             [    ]
             [ - 3]
             [    ]
             [ 2  ]
             ,
             [ 0  ]
             [    ]
             [ 1  ]
             [    ]
             [ - 2]
             [    ]
             [ 1  ]
            }
```

Comments

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.

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.

13.7 RANK

RANK   Operator

`rank`(*matrix_expression*)

`rank` calculates the rank of its matrix argument.

Examples

`rank mat((a,b,c),(d,e,f));` $\Rightarrow$ `2`

Comments

The argument to `rank` can also be a `list` of lists, interpreted either as a row matrix or a set of equations. 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 `rank` in applications different from classical linear algebra.

13.8 TP

TP — Operator

The `tp` operator returns the transpose of its `matrix` argument.

`tp` *identifier* or `tp`(*identifier*)

identifier must be a matrix, which either has had its dimensions set in its declaration, or has had values put into it by `mat`.

Examples

```
matrix m,n;

m := mat((1,2,3),(4,5,6))

n := tp m;                 ⇒   N(1,1) := 1
                               N(1,2) := 4
                               N(2,1) := 2
                               N(2,2) := 5
                               N(3,1) := 3
                               N(3,2) := 6
```

Comments

In an assignment statement involving `tp`, the matrix identifier on the left-hand side is redimensioned to the correct size for the transpose.

13.9 TRACE

TRACE Operator

The `trace` operator finds the trace of its `matrix` argument.

`trace`(*expression*) or `trace` *simple_expression*

expression or *simple_expression* must evaluate to a square matrix.

Examples

```
matrix a;

a := mat((x1,y1),(x2,y2))$

trace a;                        ⇒   X1 + Y2
```

Comments

The trace is the sum of the entries along the diagonal of a square matrix. Given a non-matrix expression, or a non-square matrix, `trace` returns an error message.

14 Groebner package

14.1 Groebner bases

GROEBNER BASES — Introduction

The GROEBNER package calculates `Groebner bases` using the `Buchberger algorithm` and provides related algorithms for arithmetic with ideal bases, such as ideal quotients, Hilbert polynomials (`Hollmann algorithm`), basis conversion (`Faugere-Gianni-Lazard-Mora algorithm`), independent variable set (`Kredel-Weispfenning algorithm`).

Some routines of the Groebner package are used by `solve` - in that context the package is loaded automatically. However, if you want to use the package by explicit calls you must load it by

```
load_package groebner;
```

For the common parameter setting of most operators in this package see `ideal parameters`.

14.2 Ideal Parameters

IDEAL PARAMETERS Concept

Most operators of the `Groebner` package compute expressions in a polynomial ring which given as $R[var,var,...]$ where R is the current REDUCE coefficient domain. All algebraically exact domains of REDUCE are supported. The package can operate over rings and fields. The operation mode is distinguished automatically. In general the ring mode is a bit faster than the field mode. The factoring variant can be applied only over domains which allow you factoring of multivariate polynomials.

The variable sequence *var* is either declared explicitly as argument in form of a `list` in `torder`, or it is extracted automatically from the expressions. In the second case the current REDUCE system order is used (see `korder`) for arranging the variables. If some kernels should play the role of formal parameters (the ground domain R then is the polynomial ring over these), the variable sequences must be given explicitly.

All REDUCE `kernel`s can be used as variables. But please note, that all variables are considered as independent. E.g. when using `sin(a)` and `cos(a)` as variables, the basic relation `sin(a)^2+cos(a)^2-1=0` must be explicitly added to an equation set because the Groebner operators don't include such knowledge automatically.

The terms (monomials) in polynomials are arranged according to the current `term order`. Note that the algebraic properties of the computed results only are valid as long as neither the ordering nor the variable sequence changes.

The input expressions *exp* can be polynomials p, rational functions n/d or equations lh=rh built from polynomials or rational functions. Apart from the `tracing` algorithms `groebnert` and `preducet`, where the equations have a specific meaning, equations are converted to simple expressions by taking the difference of the left-hand and right-hand sides *lh-rh=¿p*. Rational functions are converted to polynomials by converting the expression to a common denominator form first, and then using the numerator only *n=¿p*. So eventual zeros of the denominators are ignored.

A basis on input or output of an algorithm is coded as `list` of expressions $\{exp,exp,...\}$
.

14.3 Term order

14.4 Term order

TERM ORDER Introduction

For all `Groebner` operations the polynomials are represented in distributive form: a sum of terms (monomials). The terms are ordered corresponding to the actual `term order` which is set by the `torder` operator, and to the actual variable sequence which is either given as explicit parameter or by the system `kernel` order.

14.5 torder

TORDER Operator

The operator `torder` sets the actual variable sequence and term order.

1. simple term order:

`torder`(vl, m)

where vl is a `list` of variables (`kernels`) and m is the name of a simple `term order` mode 14.7, 14.8, 14.9 or another implemented parameterless mode.

2. stepped term order:

`torder` (vl, m, n)

where m is the name of a two step term order, one of `gradlexgradlex term order`, `gradlexrevgradlex term order`, `lexgradlex term order` or `lexrevgradlex term order`, and n is a positive integer.

3. weighted term order

`torder` $(vl, \texttt{weighted}, n, n, ...)$;

where the n are positive integers, see `weighted term order`.

4. matrix term order

`torder` $(vl, \texttt{matrix}, m)$;

where m is a matrix with integer elements, see `torder_compile`.

5. compiled term order

`torder` $(vl, \texttt{co})$;

where co is the name of a routine generated by `torder_compile`.

`torder` sets the variable sequence and the term order mode. If the an empty list is used as variable sequence, the automatic variable extraction is activated. The defaults are the empty variable list an the `lex term order`. The previous setting is returned as a list.

Alternatively to the above syntax the arguments of `torder` may be collected in a `list` and passed as one argument to `torder`.

14.6 torder_compile

TORDER_COMPILE   Operator

A matrix can be converted into a compilable LISP program for faster execution by using

`torder_compile`(*name*, *mat*)

where *name* is an identifier for the new term order and *mat* is an integer matrix to be used as `matrix term order`. Afterwards the term order can be activated by using *name* in a `torder` expression. The resulting program is compiled if the switch `comp` is on, or if the `torder_compile` expression is part of a compiled module.

14.7 lex term order

LEX TERM ORDER — Concept

The terms are ordered lexicographically: two terms t1 t2 are compared for their degrees along the fixed variable sequence: t1 is higher than t2 if the first different degree is higher in t1. This order has the `elimination property` for `groebner basis` calculations. If the ideal has a univariate polynomial in the last variable the groebner basis will contain such polynomial. `Lex` is best suited for solving of polynomial equation systems.

14.8 gradlex term order

GRADLEX TERM ORDER — Concept

The terms are ordered first with their total degree, and if the total degree is identical the comparison is `lex term order`. With `groebner` basis calculations this term order produces polynomials of lowest degree.

14.9 revgradlex term order

REVGRADLEX TERM ORDER **Concept**

The terms are ordered first with their total degree (degree sum), and if the total degree is identical the comparison is the inverse of `lex term order`. With `groebner` and `groebnerf` calculations this term order is similar to `gradlex term order`; it is known as most efficient ordering with respect to computing time.

14.10 gradlexgradlex term order

GRADLEXGRADLEX TERM ORDER — Concept

The terms are separated into two groups where the second parameter of the `torder` call determines the length of the first group. For a comparison first the total degrees of both variable groups are compared. If both are equal `gradlex term order` comparison is applied to the first group, and if that does not decide `gradlex term order` is applied for the second group. This order has the elimination property for the variable groups. It can be used e.g. for separating variables from parameters.

14.11 gradlexrevgradlex term order

GRADLEXREVGRADLEX TERM ORDER Concept

Similar to `gradlexgradlex term order`, but using `revgradlex term order` for the second group.

14.12 lexgradlex term order

LEXGRADLEX TERM ORDER — Concept

Similar to `gradlexgradlex term order`, but using `lex term order` for the first group.

14.13 lexrevgradlex term order

LEXREVGRADLEX TERM ORDER — Concept

Similar to `gradlexgradlex term order`, but using `lex term order` for the first group `revgradlex term order` for the second group.

14.14 weighted term order

WEIGHTED TERM ORDER Concept

establishes a graduated ordering similar to `gradlex term order`, where the exponents first are multiplied by the given weights. If there are less weight values than variables, the weight list is extended by ones. If the weighted degree comparison is not decidable, the `lex term order` is used.

14.15 graded term order

GRADED TERM ORDER — Concept

establishes a cascaded term ordering: first a graduated ordering similar to `gradlex term order` is used, where the exponents first are multiplied by the given weights. If there are less weight values than variables, the weight list is extended by ones. If the weighted degree comparison is not decidable, the term ordering described in the following parameters of the `torder` command is used.

14.16 matrix term order

MATRIX TERM ORDER Concept

Any arbitrary term order mode can be installed by a matrix with integer elements where the row length corresponds to the variable number. The matrix must have at least as many rows as columns. It must have full rank, and the top nonzero element of each column must be positive.

The matrix `term order mode` defines a term order where the exponent vectors of the monomials are first multiplied by the matrix and the resulting vectors are compared lexicographically.

If the switch `comp` is on, the matrix is converted into a compiled LISP program for faster execution. A matrix can also be compiled explicitly, see `torder_compile`.

14.17 Basic Groebner operators

14.18 gvars

GVARS — Operator

`gvars`($\{exp, exp, ...\}$)

where *exp* are expressions or `equation`s.

`gvars` extracts from the expressions the `kernels` which can play the role of variables for a `groebner` or `groebnerf` calculation.

14.19 groebner

GROEBNER Operator

`groebner({exp, ...})`

where {`exp`, ... } is a list of expressions or equations.

The operator `groebner` implements the Buchberger algorithm for computing Groebner bases for a given set of expressions with respect to the given set of variables in the order given. As a side effect, the sequence of variables is stored as a REDUCE list in the shared variable `gvarslast` - this is important in cases where the algorithm rearranges the variable sequence because `groebopt` is `on`.

Examples

`groebner({x**2+y**2-1,x-y})` ⇒ `{X - Y,2*Y**2 -1}`

Related information

`groebnerf` **operator**

`gvarslast` **variable**

`groebopt` **switch**

`groebprereduce` **switch**

`groebfullreduction` **switch**

`gltbasis` **switch**

`gltb` **variable**

`glterms` **variable**

`groebstat` **switch**

`trgroeb` **switch**

`trgroebs` **switch**

`groebprot` **switch**

`groebprotfile` **variable**

`groebnert` **operator**

14.20 groebner_walk

GROEBNER_WALK — Operator

The operator `groebner_walk` computes a `lex` basis from a given `graded` (or `weighted`) one.

`groebner_walk`(g)

where g is a `graded` basis (or `weighted` basis with a weight vector with one repeated element) of the polynomial ideal. `Groebner_walk` computes a sequence of monomial bases, each time lifting the full system to a complete basis. `Groebner_walk` should be called only in cases, where a normal `kex` computation would take too much computer time.

The operator `torder` has to be called before in order to define the variable sequence and the term order mode of g.

The variable `gvarslast` is not set.

Do not call `groebner_walk` with `on groebopt`.

`Groebner_walk` includes some overhead (such as e. g. computation with division). On the other hand, sometimes `groebner_walk` is faster than a direct `lex` computation.

14.21 groebopt

GROEBOPT Switch

If `groebopt` is set ON, the sequence of variables is optimized with respect to execution speed of `groebner` calculations; note that the final list of variables is available in `gvarslast`. By default `groebopt` is off, conserving the original variable sequence.

An explicitly declared dependency using the `depend` declaration supersedes the variable optimization.

Examples

`depend a, x, y;` $\Rightarrow$

guarantees that a will be placed in front of x and y.

14.22 gvarslast

GVARSLAST Variable

After a groebner or groebnerf calculation the actual variable sequence is stored in the variable gvarslast. If groebopt is on gvarslast shows the variable sequence after reordering.

14.23 groebprereduce

GROEBPREREDUCE Switch

If `groebprereduce` set ON, `groebner` and `groebnerf` try to simplify the input expressions: if the head term of an input expression is a multiple of the head term of another expression, it can be reduced; these reductions are done cyclicly as long as possible in order to shorten the main part of the algorithm.

By default `groebprereduce` is off.

14.24 groebfullreduction

GROEBFULLREDUCTION — Switch

If `groebfullreduction` set off, the polynomial reduction steps during `groebner` and `groebnerf` are limited to the pure head term reduction; subsequent terms are reduced otherwise.

By default `groebfullreduction` is on.

14.25 gltbasis

GLTBASIS — Switch

If `gltbasis` set on, the leading terms of the result basis of a `groebner` or `groebnerf` calculation are extracted. They are collected as a basis of monomials, which is available as value of the global variable `gltb`.

14.26 gltb

GLTB Variable

See `gltbasis`

14.27 glterms

GLTERMS Variable

If the expressions in a `groebner` or `groebnerf` call contain parameters (symbols which are not member of the variable list), the share variable `glterms` is set to a list of expression which during the calculation were assumed to be nonzero. The calculated bases are valid only under the assumption that all these expressions do not vanish.

14.28 groebstat

GROEBSTAT Switch

if `groebstat` is on, a summary of the `groebner` or `groebnerf` computation is printed at the end including the computing time, the number of intermediate H polynomials and the counters for the criteria hits.

14.29 trgroeb

TRGROEB Switch

if `trgroeb` is on, intermediate H polynomials are printed during a `groebner` or `groebnerf` calculation.

14.30 trgroebs

TRGROEBS Switch

if `trgroebs` is on, intermediate H and S polynomials are printed during a `groebner` or `groebnerf` calculation.

14.31 gzerodim?

GZERODIM? Operator

`gzerodim!?`(*basis*)

where *bas* is a Groebner basis in the current `term order` with the actual setting (see `ideal parameters`).

`gzerodim!?` tests whether the ideal spanned by the given basis has dimension zero. If yes, the number of zeros is returned, `nil` otherwise.

14.32 gdimension

GDIMENSION Operator

`gdimension(`*bas*`)`

where *bas* is a `groebner` basis in the current term order (see `ideal parameters`). `gdimension` computes the dimension of the ideal spanned by the given basis and returns the dimension as an integer number. The Kredel-Weispfenning algorithm is used: the dimension is the length of the longest independent variable set, see `gindependent_sets`

14.33 gindependent_sets

GINDEPENDENT_SETS Operator

`gindependent_sets`(*bas*)

where *bas* is a `groebner` basis in any `term order` (which must be the current `term order`) with the specified variables (see `ideal parameters`).

`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. The Kredel-Weispfenning algorithm is used.

14.34 dd_groebner

DD_GROEBNER Operator

For a homogeneous system of polynomials under `graded term order`, `gradlex term order`, `revgradlex term order` or `weighted term order` a Groebner Base can be computed with limiting the grade of the intermediate S polynomials:

`dd_groebner`(*d1*, *d2*, *plist*)

where *d1* is a non negative integer and *d2* is an integer or "infinity". A pair of polynomials is considered only if the grade of the lcm of their head terms is between *d1* and *d2*. For the term orders `graded` or `weighted` the (first) weight vector is used for the grade computation. Otherwise the total degree of a term is used.

14.35 glexconvert

GLEXCONVERT Operator

`glexconvert`$(bas[, vars][, MAXDEG = mx][, NEWVARS = nv])$

where *bas* is a `groebner` basis in the current term order, *mx* (optional) is a positive integer and *nvl* (optional) is a list of variables (see `ideal parameters`).

The operator `glexconvert` converts the basis of a zero-dimensional ideal (finite number of isolated solutions) from arbitrary ordering into a basis under `lex term order`.

The parameter *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.

If *newvars* is a list with one element, the minimal `univariate polynomial` is computed.

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.

Comments

During the call the `term order` of the input basis must be active.

14.36 greduce

GREDUCE Operator

`greduce`$(exp, \{exp1, exp2, \ldots, expm\})$

where exp is an expression, and {exp1, exp2, ... , expm} is a list of expressions or equations.

`greduce` is functionally equivalent with a call to `groebner` and then a call to `preduce`.

14.37 preduce

PREDUCE Operator

`preduce`$(p, \{exp, \ldots\})$

where p is an expression, and $\{exp, \ldots\}$ is a list of expressions or equations.

`Preduce` computes the remainder of `exp` modulo the given set of polynomials resp. equations. This result is unique (canonical) only if the given set is a `groebner` basis under the current `term order`

see also: `preducet` operator.

14.38 idealquotient

IDEALQUOTIENT — Operator

`idealquotient`$(\{exp, ...\}, d)$

where {*exp*,...} is a list of expressions or equations, d is a single expression or equation.

`Idealquotient` computes the ideal quotient: ideal spanned by the expressions {*exp*,...} divided by the single polynomial/expression f. The result is the `groebner` basis of the quotient ideal.

14.39 hilbertpolynomial

HILBERTPOLYNOMIAL Operator

hilbertpolynomial(*bas*)

where *bas* is a `groebner` basis in the current `term order`.

The degree of the `Hilbert polynomial` is the dimension of the ideal spanned by the basis. For an ideal of dimension zero the Hilbert polynomial is a constant which is the number of common zeros of the ideal (including eventual multiplicities). The `Hollmann algorithm` is used.

14.40 saturation

SATURATION Operator

`saturation`($\{exp, ...\}, p$)

where $\{exp$,...$\}$ is a list of expressions or equations, p is a single polynomial.

`Saturation` computes the quotient of the polynomial p and a power (with unknown but finite exponent) of the ideal built from $\{exp$, ...$\}$. The result is the computed quotient. `Saturation` calls `idealquotient` several times until the result does not change any more.

14.41 Factorizing Groebner bases

14.42 groebnerf

GROEBNERF Operator

`groebnerf`$(\{exp, ...\}[, \{\}, \{nz, ...\}]);$

where $\{exp, ...\}$ is a list of expressions or equations, and $\{nz, ...\}$ is an optional list of polynomials to be considered as non zero for this calculation. An empty list must be passed as second argument if the non-zero list is specified.

`groebnerf` tries to separate polynomials into individual factors and to branch the computation in a recursive manner (factorization tree). The result is a list of partial Groebner bases. 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 third parameter of `groebnerf` declares some polynomials nonzero. If any of these is found in a branch of the calculation the branch is canceled.

Example

```
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,x});

       {{Y - 3,X},

                          2
    {2*Y + 2*X - 1,2*X  - 5*X - 5}}
```

Related information

`groebresmax` **variable**

`groebmonfac` **variable**

`groebrestriction` **variable**

`groebner` **operator**

`gvarslast` **variable**

`groebopt` **switch**

`groebprereduce` **switch**

`groebfullreduction` **switch**

`gltbasis` **switch**

`gltb` **variable**

`glterms` **variable**

`groebstat` **switch**

`trgroeb` **switch**

`trgroebs` **switch**

`groebnert` **operator**

14.43 groebmonfac

GROEBMONFAC Variable

The variable `groebmonfac` is connected to the handling of monomial factors. A monomial factor is a product of variable powers as a factor, e.g. x**2*y in x**3*y - 2*x**2*y**2. A monomial factor represents a solution of the type x = 0 or y = 0 with a certain multiplicity. With `groebnerf` the multiplicity of monomial factors is lowered to the value of the shared variable `groebmonfac` which by default is 1 (= monomial factors remain present, but their multiplicity is brought down). With `groebmonfac`:= 0 the monomial factors are suppressed completely.

14.44 groebresmax

GROEBRESMAX Variable

The variable `groebresmax` controls during `groebnerf` calculations the number of partial results. Its default value is 300. If more partial results are calculated, the calculation is terminated.

14.45 groebrestriction

GROEBRESTRICTION Variable

During `groebnerf` calculations irrelevant branches can be excluded by setting the variable `groebrestriction`. The following restrictions are implemented:

`groebrestriction` := `nonnegative`
`groebrestriction` := `positive`
`groebrestriction` := `zeropoint`

With `nonnegative` branches are excluded where one polynomial has no nonnegative real zeros; with `positive` the restriction is sharpened to positive zeros only. The restriction `zeropoint` excludes all branches which do not have the origin (0,0,...0) in their solution set.

14.46 Tracing Groebner bases

14.47 groebprot

GROEBPROT Switch

If `groebprot` is `ON` the computation steps during `preduce`, `greduce` and `groebner` are collected in a list which is assigned to the variable `groebprotfile`.

14.48 groebprotfile

GROEBPROTFILE Variable

See `groebprot` switch.

14.49 groebnert

GROEBNERT Operator

groebnert($\{v = exp, ...\}$)

where v are `kernels` (simple or indexed variables), *exp* are polynomials.

`groebnert` is functionally equivalent to a `groebner` call for $\{exp,...\}$, but the result is a set of equations where the left-hand sides are the basis elements while the right-hand sides are the same values expressed as combinations of the input formulas, expressed in terms of the names v

Example

```
 groebnert({p1=2*x**2+4*y**2-100,p2=2*x-y+1});

GB1 := {2*X - Y + 1=P2,

           2
      9*Y  - 2*Y - 199= - 2*X*P2 - Y*P2 + 2*P1 + P2}
```

14.50 preducet

PREDUCET Operator

`preduce`$(p, \{v = exp...\})$

where p is an expression, v are kernels (simple or indexed variables), `exp` are polynomials.

`preducet` computes the remainder of p modulo $\{exp,...\}$ similar to `preduce`, but the result is an equation which expresses the remainder as combination of the polynomials.

Example

```
   GB2 := {G1=2*X - Y + 1,G2=9*Y**2  - 2*Y - 199}
   preducet(q=x**2,gb2);

 - 16*Y + 208= - 18*X*G1 - 9*Y*G1 + 36*Q + 9*G1 - G2
```

14.51 Groebner Bases for Modules

14.52 Module

MODULE Concept

Given a polynomial ring, e.g. R=Z[x,y,...] and an integer n¿1. The vectors with n elements of R form a free MODULE under elementwise addition and multiplication with elements of R.

For a submodule given by a finite basis a Groebner basis can be computed, and the facilities of the GROEBNER package are available except the operators `groebnerf` and `groesolve`. The vectors are encoded using auxiliary variables which represent the unit vectors in the module. These are declared in the share variable `gmodule`.

14.53 gmodule

GMODULE Variable

The vectors of a free module over a polynomial ring R are encoded as linear combinations with unit vectors of M which are represented by auxiliary variables. These must be collected in the variable gmodule before any call to an operator of the Groebner package.

```
torder({x,y,v1,v2,v3})$
gmodule := {v1,v2,v3}$
g:=groebner({x^2*v1 + y*v2,x*y*v1 - v3,2y*v1 + y*v3});
```

compute the Groebner basis of the submodule

```
([x^2,y,0],[xy,0,-1],[0,2y,y])
```

The members of the list gmodule are automatically appended to the end of the variable list, if they are not yet members there. They take part in the actual term ordering.

14.54 Computing with distributive polynomials

14.55 gsort

GSORT Operator

gsort(p)

where p is a polynomial or a list of polynomials.

The polynomials are reordered and sorted corresponding to the current `term order`.

Examples

```
torder lex;

gsort(x**2+2x*y+y**2,{y,x});
```

$\Rightarrow$ `y**2+2y*x+x**2`

14.56 gsplit

GSPLIT Operator

`gsplit`$(p[, vars])$;

where p is a polynomial or a list of polynomials.

The polynomial is reordered corresponding to the the current `term order` and then separated into leading term and reductum. Result is a list with the leading term as first and the reductum as second element.

Examples

```
torder lex;

gsplit(x**2+2x*y+y**2,{y,x});
```

$\Rightarrow$ `{y**2,2y*x+x**2}`

14.57 gspoly

GSPOLY Operator

`gspoly`(*p1*, *p2*);

where *p1* and *p2* are polynomials.

The `subtraction` polynomial of p1 and p2 is computed corresponding to the method of the Buchberger algorithm for computing `groebner bases`: p1 and p2 are multiplied with terms such that when subtracting them the leading terms cancel each other.

15 High Energy Physics

15.1 HEPHYS

HEPHYS **Introduction**

The High-energy Physics package is historic for REDUCE, since REDUCE originated as a program to aid in computations with Dirac expressions. The commutation algebra of the gamma matrices is independent of their representation, and is a natural subject for symbolic mathematics. Dirac theory is applied to β decay and the computation of cross-sections and scattering. The high-energy physics operators are available in the REDUCE main program, rather than as a module which must be loaded.

15.2 HE-dot

. **Operator**

The . operator is used to denote the scalar product of two Lorentz four-vectors.

vector . *vector*

vector must be an identifier declared to be of type `vector` to have the scalar product definition. When applied to arguments that are not vectors, the `cons` operator is used, whose symbol is also "dot."

Examples

```
vector aa,bb,cc;

let aa.bb = 0;

aa.bb;           ⇒   0

aa.cc;           ⇒   AA.CC

q := aa.cc;      ⇒   Q := AA.CC

q;               ⇒   AA.CC
```

Comments

Since vectors are special high-energy physics entities that do not contain values, the . product will not return a true scalar product. You can assign a scalar identifier to the result of a . operation, or assign a . operation to have the value of the scalar you supply, as shown above. Note that the result of a . operation is a scalar, not a vector.

The metric tensor g(u,v) can be represented by `u.v`. If contraction over the indices is required, `u` and `v` should be declared to be of type `index`.

The dot operator has the highest precedence of the infix operators, so expressions involving . and other operators have the scalar product evaluated first before other operations are done.

15.3 EPS

EPS — Operator

The `eps` operator denotes the completely antisymmetric tensor of order 4 and its contraction with Lorentz four-vectors, as used in high-energy physics calculations.

$$\texttt{eps}(vector - expr, vector - expr, vector - expr, vector - expr)$$

vector-expr must be a valid vector expression, and may be an index.

Examples

```
vector g0,g1,g2,g3;

eps(g1,g0,g2,g3);    ⇒   - EPS(G0,G1,G2,G3);

eps(g1,g2,g0,g3);    ⇒   EPS(G0,G1,G2,G3);

eps(g1,g2,g3,g1);    ⇒   0
```

Comments

Vector identifiers are ordered alphabetically by REDUCE. When an odd number of transpositions is required to restore the canonical order to the four arguments of `eps`, the term is ordered and carries a minus sign. When an even number of transpositions is required, the term is returned ordered and positive. When one of the arguments is repeated, the value 0 is returned. A contraction of the form $\epsilon_{ij\mu\nu}p_\mu q_\nu$ is represented by `eps(i,j,p,q)` when `i` and `j` have been declared to be of type `index`.

15.4 G

G Operator

`g` is an n-ary operator used to denote a product of gamma matrices contracted with Lorentz four-vectors, in high-energy physics.

$$\mathtt{g}(\mathit{identifier}, \mathit{vector} - \mathit{expr}\{, \mathit{vector} - \mathit{expr}\}*)$$

identifier is a scalar identifier representing a fermion line identifier, *vector-expr* can be any valid vector expression, representing a vector or a gamma matrix.

Examples

```
vector aa,bb,cc;

vector a;

g(line1,aa,bb);          ⇒   AA.BB

g(line2,aa,a);           ⇒   0

g(id,aa,bb,cc);          ⇒   0

g(li1,aa,bb) + k;        ⇒   AA.BB + K

let aa.bb = m*k;

g(ln1,aa)*g(ln1,bb);     ⇒   K*M

g(ln1,aa)*g(ln2,bb);     ⇒   0
```

Comments

The vector `A` is reserved in arguments of `g` to denote the special gamma matrix γ_5. It must be declared to be a vector before you use it.

Gamma matrix expressions are associated with fermion lines in a Feynman diagram. If more than one line occurs in an expression, the gamma matrices involved are separate (operating in independent spin space), as shown in the last two example lines above. A product of gamma matrices associated with a single line can be entered either as a single `g` command with several vector arguments, or as products of separate `g` commands each with a single argument.

While the product of vectors is not defined, the product, sum and difference of several gamma expressions are defined, as is the product of a gamma expression with a scalar. If an expression involving gamma matrices includes a scalar, the scalar is treated as if it were the product of itself with a unit 4×4 matrix.

Dirac expressions are evaluated by computing the trace of the expression using the commutation algebra of gamma matrices. The algorithms used are described in articles by J. S. R. Chisholm in *Il Nuovo Cimento X,* Vol. 30, p. 426, 1963, and J. Kahane, *Journal of Mathematical Physics*, Vol. 9, p. 1732, 1968. The trace is then divided by 4 to distinguish between the trace of a scalar and the trace of an expression that is the product of a scalar with a unit 4×4 matrix.

Trace calculations may be prevented over any line identifier by declaring it to be `nospur`. If it is later desired to evaluate these traces, the declaration can be undone with the `spur` declaration.

The notation of Bjorken and Drell, *Relativistic Quantum Mechanics,* 1964, is assumed in all operations involving gamma matrices. For an example of the use of `g` in a calculation, see the *REDUCE User's Manual*.

15.5 INDEX

INDEX — Declaration

The declaration `index` flags a four-vector as an index for subsequent high-energy physics calculations.

`index` *vector-id*{,*vector-id*}*

vector-id must have been declared of type `vector`.

Examples

```
vector aa,bb,cc;

index uu;

let aa.bb = 0;

(aa.uu)*(bb.uu);  ⇒  0

(aa.uu)*(cc.uu);  ⇒  AA.CC
```

Comments

Index variables are used to represent contraction over components of vectors when scalar products are taken by the `.` operator, as well as indicating contraction for the `eps` operator or metric tensor.

The special status of a vector as an index can be revoked with the declaration `remind`. The object remains a vector, however.

15.6 MASS

MASS — Command

The `mass` command associates a scalar variable as a mass with the corresponding vector variable, in high-energy physics calculations.

`mass` *vector-var*=*scalar-var* {,*vector-var*=*scalar-var*}*

vector-var can be a declared vector variable; `mass` will declare it to be of type `vector` if it is not. This may override an existing matrix variable by that name. *scalar-var* must be a scalar variable.

Examples

```
vector bb,cc;

mass cc=m;

mshell cc;
                       2
cc.cc;          ⇒     M
```

Comments

Once a mass has been attached to a vector with a `mass` declaration, the `mshell` declaration puts the associated particle "on the mass shell." Subsequent scalar (.) products of the vector with itself will be replaced by the square of the mass expression.

15.7 MSHELL

MSHELL Command

The `mshell` command puts particles on the mass shell in high-energy physics calculations.

`mshell` *vector-var*{,*vector-var*}∗

vector-var must have had a mass attached to it by a `mass` declaration.

Examples

```
vector v1,v2;

mass v1=m,v2=q;

mshell v1;
                         2
v1.v1;          ⇒     M

v2.v2;          ⇒     V2.V2

mshell v2;
                         2  2
v1.v1*v2.v2;    ⇒     M *Q
```

Comments

Even though a mass is attached to a vector variable representing a particle, the replacement does not take place until the `mshell` declaration is given for that vector variable.

15.8 NOSPUR

NOSPUR Declaration

The `nospur` declaration prevents the trace calculation over the given line identifiers in high-energy physics calculations.

`nospur` *line-id*{,*line-id*}∗

line-id is a scalar identifier that will be used as a line identifier.

Examples

```
vector a1,b1,c1;

g(line1,a1,b1)*g(line2,b1,c1);

                    ⇒  A1.B1*B1.C1

nospur line2;

g(line1,a1,b1)*g(line2,b1,c1);

                    ⇒  A1.B1*G(LINE2,B1,C1)
```

Comments

Nospur declarations can be removed by making the declaration `spur`.

15.9 REMIND

REMIND Declaration

The `remind` declaration removes the special status of its arguments as indices, which was set in the `index` declaration, in high-energy physics calculations.

`remind` *identifier*{,*identifier*}∗

identifier must have been declared to be of type `index`.

15.10 SPUR

SPUR Declaration

The `spur` declaration removes the special exemption from trace calculations that was declared by `nospur`, in high-energy physics calculations.

`spur` *line-id*{,*line-id*}*

line-id must be a line-identifier that has previously been declared `nospur`.

15.11 VECDIM

VECDIM Command

The command `vecdim` changes the vector dimension from 4 to an arbitrary integer or symbol. Used in high-energy physics calculations.

`vecdim` *dimension*

dimension must be either an integer or a valid scalar identifier that does not have a floating-point value.

Comments

The `eps` operator and the γ_5 symbol (`A`) are not properly defined in anything except four dimensions and will print an error message if you use them that way. The other high-energy physics operators should work without problem.

15.12 VECTOR

VECTOR — Declaration

The `vector` declaration declares that its arguments are of type `vector`.

`vector` *identifier*{,*identifier*}*

identifier must be a valid REDUCE identifier. It may have already been used for a matrix, array, operator or scalar variable. After an identifier has been declared to be a vector, it may not be used as a scalar variable.

Comments

Vectors are special entities for high-energy physics calculations. You cannot put values into their coordinates; they do not have coordinates. They are legal arguments for the high-energy physics operators `eps`, `g` and `.` (dot). Vector variables are used to represent gamma matrices and gamma matrices contracted with Lorentz 4-vectors, since there are no Dirac variables per se in the system. Vectors do follow the usual vector rules for arithmetic operations: `+` and `-` operate upon two or more vectors, producing a vector; `*` and `/` cannot be used between vectors; the scalar product is represented by the `.` operator; and the product of a scalar and vector expression is well defined, and is a vector.

You can represent components of vectors by including representations of unit vectors in your system. For instance, letting `E0` represent the unit vector (1,0,0,0), the command

```
V1.E0 := 0;
```

would set up the substitution of zero for the first component of the vector `V1`.

Identifiers that are declared by the `index` and `mass` declarations are automatically declared to be vectors.

The following errors can occur in calculations using the high energy physics package:

`A represents only gamma5 in vector expressions`

> You have tried to use A in some way other than gamma5 in a high-energy physics expression.

`Gamma5 not allowed unless vecdim is 4`

You have used γ_5 in a high-energy physics computation involving a vector dimension other than 4.

ID `has no mass`

One of the arguments to `mshell` has had no mass assigned to it, in high-energy physics calculations.

`Missing arguments for G operator`

A line symbol is missing in a gamma matrix expression in high-energy physics calculations.

`Unmatched index` *list*

The parser has found unmatched indices during the evaluation of a gamma matrix expression in high-energy physics calculations.

16 Numeric Package

16.1 Numeric Package

NUMERIC PACKAGE — Introduction

The numeric package supplies algorithms based on approximation techniques of numerical mathematics. The algorithms use the `rounded` mode arithmetic of REDUCE, including the variable precision feature which is exploited in some algorithms in an adaptive manner in order to reach the desired accuracy.

16.2 Interval

INTERVAL Type

Intervals are generally coded as lower bound and upper bound connected by the operator `..`, usually associated to a variable in an equation.

$$var = (low..high)$$

where *var* is a `kernel` and *low*, *high* are numbers or expression which evaluate to numbers with *low*¡=*high*.

Examples

`x= (2.5 .. 3.5)` ⇒

means that the variable x is taken in the range from 2.5 up to 3.5.

16.3 numeric accuracy

NUMERIC ACCURACY Concept

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.

16.4 TRNUMERIC

TRNUMERIC — Switch

Normally the algorithms produce only a minimum of printed output during their operation. In cases of an unsuccessful or unexpected long operation a `trace of the iteration` can be printed by setting `trnumeric on`.

16.5 num_min

NUM_MIN Operator

The Fletcher Reeves version of the `steepest descent` algorithms is used to find the `minimum` of a function of one or more variables. The function 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.

`num_min`$(exp, var[=\ val][, var[=\ val]...[, accuracy\ =\ a][, iterations\ =\ i])$

or

`num_min`$(exp, \{var[=val][, var[=val]...]\}[, accuracy = a][, iterations = i])$

where *exp* is a function expression, *var* are the variables in *exp* and *val* are the (optional) start values. For a and i see `numeric accuracy`.

`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}}`

16.6 num_solve

NUM_SOLVE — Operator

An adaptively damped Newton iteration is used to find an approximative root of a function (function vector) or the solution of an `equation` (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. When 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 `complex` on, solutions with imaginary parts can be found, if either the expression(s) or the starting point contain a nonzero imaginary part.

`num_solve`$(exp, var[= val][, accuracy = a][, iterations = i])$

or

`num_solve`$(\{exp, ..., exp\}, var[= val], ..., var[= val][, accuracy = a][, iterations = i])$

or

`num_solve`$(\{exp, ..., exp\}, \{var[= val], ..., var[= val]\}[, accuracy = a][, iterations = i])$

where *exp* are function expressions, *var* are the variables, *val* are optional start values. For a and i see `numeric accuracy`.

`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 side effect the shared variable `jacobian`.

Examples

```
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   ]
```

16.7 num_int

NUM_INT Operator

For the numerical evaluation of univariate integrals over a finite interval the following strategy is used: If `int` finds a formal antiderivative which is bounded in the integration interval, this is evaluated and the end points and the difference is returned. Otherwise a `Chebyshev fit` is computed, starting with order 20, eventually up to order 80. If that is recognized as sufficiently convergent it is used for computing the integral by directly integrating the coefficient sequence. If none of these methods is successful, an adaptive multilevel quadrature algorithm is used.

For multivariate integrals only the adaptive quadrature is used. This algorithm tolerates isolated singularities. The value `iterations` here limits the number of local interval intersection levels. a is a measure for the relative total discretization error (comparison of order 1 and order 2 approximations).

`num_int`$(exp, var = (l \,..\, u)[, var = (l \,..\, u), ...][, accuracy = a][, iterations = i])$

where *exp* is the function to be integrated, *var* are the integration variables, l are the lower bounds, u are the upper bounds.

Result is the value of the integral.

Examples

```
num_int(sin x,x=(0 .. 3.1415926));
```

$\Rightarrow$ `2.0000010334`

16.8 num_odesolve

NUM_ODESOLVE Operator

The `Runge-Kutta` method of order 3 finds an approximate graph for the solution of real `ODE initial value problem`.

`num_odesolve`$(exp, depvar = start, indep = (from\ ..\ to)[, accuracy = a][, iterations = i])$

or

`num_odesolve`$(\{exp, exp, ...\}, \{depvar = start, depvar = start, ...\} indep = (from\ ..\ to)[, accuracy = a][, iterations = i])$

where *depvar* and *start* specify the dependent variable(s) and the starting point value (vector), *indep*, *from* and *to* specify the independent variable and the integration interval (starting point and end point), *exp* are equations or expressions which contain the first derivative of the independent variable with respect to the dependent variable.

The ODEs are converted to 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 neighborhood 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.

Examples

```
depend(y,x);

num_odesolve(df(y,x)=y,y=1,x=(0 .. 1), iterations=5);
```

$\Rightarrow$

```
{0.0,1.0},{0.2,1.2214},{0.4,1.49181796},{0.6,1.8221064563},  {0.8,2.2255208258},{1.0,2
```

In most cases you must declare the dependency relation between the variables explicitly using `depend`; otherwise the formal derivative might be converted to zero.

The operator `solve` is used to convert the form into an explicit ODE. If that process fails or if it has no unique result, the evaluation is stopped with an error message.

16.9 bounds

BOUNDS Operator

Upper and lower bounds of a real valued function over an `interval` or a rectangular multivariate domain are computed by the operator `bounds`. The algorithmic basis is the computation with inequalities: starting from the interval(s) of the variables, the bounds are propagated in the expression using the rules for inequality computation. Some knowledge about the behavior 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.

`bounds`$(exp, var = (l\ ..\ u)[, var = (l\ ..\ u)...])$

or

`bounds`$(exp, \{var = (l\ ..\ u)[, var = (l\ ..\ u)...]\})$

where *exp* is the function to be investigated, *var* are the variables of *exp*, *l* and *u* specify the area as set of `intervals`.

`bounds` computes upper and lower bounds for the expression in the given area. An `interval` is returned.

Examples

```
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
```

16.10 Chebyshev fit

CHEBYSHEV FIT — Concept

The operator family `Chebyshev_...` implements approximation and evaluation of functions by the Chebyshev method. Let `T(n,a,b,x)` be the Chebyshev polynomial of order `n` transformed to the interval `(a,b)`. Then a function `f(x)` can be approximated in `(a,b)` by a series

```
for i := 0:n sum c(i)*T(i,a,b,x)
```

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. `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. `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 target function), *var* is the variable of *fcn*, *lo* and *hi* are numerical real values which describe an `interval` *lo* ¡ *hi*, the integer *n* is the approximation order (set to 20 if missing), *pt* is a number in the interval and *coeffs* is a series of Chebyshev coefficients.

Examples

```
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);
```

$\Rightarrow$ `0.4111`

16.11 num_fit

NUM_FIT Operator

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.

`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.

Examples

```
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);
```

$\Rightarrow$

```
                   2
{14.571428571*X   - 61.428571429*X + 54.6,{54.6,
     - 61.428571429,14.571428571}}
```

17 Roots Package

17.1 Roots Package

ROOTS PACKAGE — Introduction

The root finding package is designed so that it can be used to find some or all of the roots of univariate polynomials with real or complex coefficients, to the accuracy specified by the user.

Not all operators of `roots package` are described here. For using the operators

`isolater` (intervals isolating real roots)

`rlrootno` (number of real roots in an interval)

`rootsat-prec` (roots at system precision)

`rootval` (result in equation form)

`firstroot` (computing only one root)

`getroot` (selecting roots from a collection)

please consult the full documentation of the package.

17.2 MKPOLY

MKPOLY Operator

Given a roots list as returned by `roots`, the operator `mkpoly` constructs a polynomial which has these numbers as roots.

`mkpoly` *rl*

where *rl* is a `list` with equations, which all have the same `kernel` on their left-hand sides and numbers as right-hand sides.

Examples

`mkpoly{x=1,x=-2,x=i,x=-i};` ⇒ `x**4 + x**3 - x**2 + x - 2`

Note that this polynomial is unique only up to a numeric factor.

17.3 NEARESTROOT

NEARESTROOT — Operator

The operator `nearestroot` finds one root of a polynomial with an iteration using a given starting point.

`nearestroot`$(p\ pt)$

where p is a univariate polynomial and pt is a number.

Examples

`nearestroot(x^2+2,2);` ⇒ `{x=1.41421*i}`

The minimal accuracy of the result values is controlled by `rootacc`.

17.4 REALROOTS

REALROOTS   Operator

The operator `realroots` finds that real roots of a polynomial to an accuracy that is sufficient to separate them and which is a minimum of 6 decimal places.

`realroots`(p) or
`realroots`(p *from*, *to*)

where p is a univariate polynomial. The optional parameters *from* and *to* classify an interval: if given, exactly the real roots in this interval will be returned. *from* and *to* can also take the values `infinity` or `-infinity`. If omitted all real roots will be returned. Result is a `list` of equations which represent the roots of the polynomial at the given accuracy.

Examples

```
realroots(x^5-2);          ⇒  {x=1.1487}

realroots(x^3-104*x^2+403*x-300,2,infinity);
                           ⇒  {x=3.0,x=100.0}

realroots(x^3-104*x^2+403*x-300,-infinity,2);
                           ⇒  {x=1}
```

The minimal accuracy of the result values is controlled by `rootacc`.

17.5 ROOTACC

ROOTACC — Operator

The operator `rootacc` allows you to set the accuracy up to which the roots package computes its results.

`rootacc`(n)

Here n is an integer value. The internal accuracy of the `roots` package is adjusted to a value of `max(6,n)`. The default value is `6`.

17.6 ROOTS

ROOTS Operator

The operator `roots` 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.

`roots`(p)

where p is a univariate polynomial. Result is a `list` of equations which represent the roots of the polynomial at the given accuracy. In addition, `roots` stores separate lists of real roots and complex roots in the global variables `rootsreal` and `rootscomplex`.

Examples

```
roots(x^5-2);  ⇒  {x=-0.929316 + 0.675188*i,
                    x=-0.929316 - 0.675188*i,
                    x=0.354967 + 1.09248*i,
                    x=0.354967 - 1.09248*i,
                    x=1.1487}
```

The minimal accuracy of the result values is controlled by `rootacc`.

17.7 ROOT_VAL

ROOT_VAL — Operator

The operator `root_val` computes the roots of a univariate polynomial at system precision (or greater if required for root separation) and presents its result as a list of numbers.

`roots`(p)

where p is a univariate polynomial.

Examples

```
root_val(x^5-2);  ⇒  {-0.929316490603 + 0.6751879524*i,
                      -0.929316490603 - 0.6751879524*i,
                      0.354967313105 + 1.09247705578*i,
                      0.354967313105 - 1.09247705578*i,
                      1.148698355}
```

17.8 ROOTSCOMPLEX

ROOTSCOMPLEX — Variable

When the operator `roots` is called the complex roots are collected in the global variable `rootscomplex` as `list`.

17.9 ROOTSREAL

ROOTSREAL Variable

When the operator `roots` is called the real roots are collected in the global variable `rootreal` as `list`.

18 Special Functions

18.1 Special Function Package

SPECIAL FUNCTION PACKAGE Introduction

The REDUCE `Special Function Package` supplies extended algebraic and numeric support for a wide class of objects. This package was released together with REDUCE 3.5 (October 1993) for the first time, a major update is released with REDUCE 3.6.

The functions included in this package are in most cases (unless otherwise stated) defined and named like in the book by Abramowitz and Stegun: Handbook of Mathematical Functions, Dover Publications.

The aim is to collect as much information on the special functions and simplification capabilities as possible, i.e. algebraic simplifications and numeric (rounded mode) code, limits of the functions together with the definitions of the functions, which are in most cases a power series, a (definite) integral and/or a differential equation.

What can be found: Some famous constants, a variety of Bessel functions, special polynomials, the Gamma function, the (Riemann) Zeta function, Elliptic Functions, Elliptic Integrals, 3J symbols (Clebsch-Gordan coefficients) and integral functions.

What is missing: Mathieu functions, LerchPhi, etc.. The information about the special functions which solve certain differential equation is very limited. In several cases numerical approximation is restricted to real arguments or is missing completely.

The implementation of this package uses REDUCE rule sets to a large extent, which guarantees a high 'readability' of the functions definitions in the source file directory. It makes extensions to the special functions code easy in most cases too.

To look at these rules it may be convenient to use the showrules operator e.g.

`showrules` Besseli;
.

Some evaluations are improved if the special function package is loaded, e.g. some (infinite) sums and products leading to expressions including special functions are known in this case.

Note: The special function package has to be loaded explicitly by calling

```
load_package specfn;
```

The functions `MeijerG` and `hypergeometric` require additionally

```
load_package specfn2;
```

18.2 Constants

CONSTANTS — Concept

There are a few constants known to the special function package, namely

- `Euler's constant` (which can be computed as `-Psi`(1)) and
- `Khinchin's constant` (which is defined in Khinchin's book "Continued Fractions") and
- `Golden_Ratio` (which can be computed as (1 + sqrt 5)/2) and
- `Catalan's constant` (which is known as an infinite sum of reciprocal powers)

Examples

```
on rounded; Euler_Gamma;   ⇒   0.577215664902
Khinchin;                  ⇒   2.68545200107
Catalan                    ⇒   0.915965594177
Golden_Ratio               ⇒   1.61803398875
```

18.3 Bernoulli Euler Zeta

18.4 BERNOULLI

BERNOULLI Operator

The `bernoulli` operator returns the nth Bernoulli number.

`Bernoulli`(*integer*)

Examples

`bernoulli 20;` ⇒ `- 174611 / 330`

`bernoulli 17;` ⇒ `0`

Comments

All Bernoulli numbers with odd indices except for 1 are zero.

18.5 BERNOULLIP

BERNOULLIP Operator

The `BernoulliP` operator returns the nth Bernoulli Polynomial evaluated at x.

`BernoulliP`(*integer*, *expression*)

Examples

```
                         2
BernoulliP(3,z);   ⇒   z*(2*z  - 3*z + 1)/2

BernoulliP(10,3);  ⇒   338585 / 66
```

Comments

The value of the nth Bernoulli Polynomial at 0 is the nth Bernoulli number.

18.6 EULER

EULER Operator

The `EULER` operator returns the nth Euler number.

`Euler`(*integer*)

Examples

`Euler 20;` ⇒ `370371188237525`

`Euler 0;` ⇒ `1`

Comments

The `Euler` numbers are evaluated by a recursive algorithm which makes it hard to compute Euler numbers above say 200.

Euler numbers appear in the coefficients of the power series representation of 1/cos(z).

18.7 EULERP

EULERP Operator

The `EulerP` operator returns the nth Euler Polynomial.

`EulerP`(*integer*, *expression*)

Examples

`EulerP(2,xx);` $\Rightarrow$ `xx*(xx - 1)`

`EulerP(10,3);` $\Rightarrow$ `2046`

Comments

The Euler numbers are the values of the Euler Polynomials at 1/2 multiplied by 2**n.

18.8 ZETA

ZETA Operator

The `Zeta` operator returns Riemann's Zeta function,

Zeta (z) := sum(1/(k**z),k,1,infinity)

`Zeta(`*expression*`)`

Examples

```
                       2
Zeta(2);      ⇒    pi   / 6

on rounded;

Zeta 1.01;    ⇒    100.577943338
```

Comments

Numerical computation for the Zeta function for arguments close to 1 are tedious, because the series is converging very slowly. In this case a formula (e.g. found in Bender/Orzag: Advanced Mathematical Methods for Scientists and Engineers, McGraw-Hill) is used.

No numerical approximation for complex arguments is done.

18.9 Bessel Functions

18.10 BESSELJ

BESSELJ Operator

The `BesselJ` operator returns the Bessel function of the first kind.

`BesselJ`(*order*, *argument*)

Examples

```
BesselJ(1/2,pi);  ⇒  0

on rounded;

BesselJ(0,1);     ⇒  0.765197686558
```

18.11 BESSELY

BESSELY Operator

The `BesselY` operator returns the Bessel function of the second kind.

`BesselY`(*order*, *argument*)

Examples

```
BesselY (1/2,pi);  ⇒   - sqrt(2) / pi

on rounded;

BesselY (1,3);     ⇒   0.324674424792
```

Comments

The operator `BesselY` is also called Weber's function.

18.12 HANKEL1

HANKEL1 Operator

The `Hankel1` operator returns the Hankel function of the first kind.

`Hankel1`(*order*, *argument*)

Examples

```
on complex;

Hankel1 (1/2,pi);  ⇒   - i * sqrt(2) / pi

Hankel1 (1,pi);    ⇒   besselj(1,pi) + i*bessely(1,pi)
```

Comments

The operator `Hankel1` is also called Bessel function of the third kind. There is currently no numeric evaluation of Hankel functions.

18.13 HANKEL2

HANKEL2 Operator

The `Hankel2` operator returns the Hankel function of the second kind.

`Hankel2`(*order*, *argument*)

Examples

```
on complex;

Hankel2 (1/2,pi);  ⇒   - i * sqrt(2) / pi

Hankel2 (1,pi);    ⇒   besselj(1,pi) - i*bessely(1,pi)
```

Comments

The operator `Hankel2` is also called Bessel function of the third kind. There is currently no numeric evaluation of Hankel functions.

18.14 BESSELI

BESSELI Operator

The `BesselI` operator returns the modified Bessel function I.

`BesselI`(*order*, *argument*)

Examples

```
on rounded;

Besseli (1,1);  ⇒  0.565159103992
```

Comments

The knowledge about the operator `BesselI` is currently fairly limited.

18.15 BESSELK

BESSELK Operator

The `BesselK` operator returns the modified Bessel function K.

`BesselK`(*order*, *argument*)

Examples

```
df(besselk(0,x),x);  ⇒   - besselk(1,x)
```

Comments

There is currently no numeric support for the operator `BesselK`.

18.16 StruveH

STRUVEH Operator

The `StruveH` operator returns Struve's H function.

`StruveH`(*order*, *argument*)

Examples

`struveh(-3/2,x);` ⇒ `- besselj(3/2,x) / i`

18.17 StruveL

STRUVEL Operator

The `StruveL` operator returns the modified Struve L function .

`StruveL`(*order*, *argument*)

Examples

`struvel(-3/2,x);` $\Rightarrow$ `besseli(3/2,x)`

18.18 KummerM

KUMMERM Operator

The `KummerM` operator returns Kummer's M function.

`KummerM`(*parameter*, *parameter*, *argument*)

Examples

```
kummerm(1,1,x);    ⇒   e^x

on rounded;

kummerm(1,3,1.3);  ⇒   1.62046942914
```

Comments

Kummer's M function is one of the Confluent Hypergeometric functions. For reference see the `hypergeometric` operator.

18.19 KummerU

KUMMERU Operator

The `KummerU` operator returns Kummer's U function.

`KummerU`(*parameter*, *parameter*, *argument*)

Examples

`df(kummeru(1,1,x),x)` $\Rightarrow$ `- kummeru(2,2,x)`

Comments

Kummer's U function is one of the Confluent Hypergeometric functions. For reference see the `hypergeometric` operator.

18.20 WhittakerW

WHITTAKERW Operator

The `WhittakerW` operator returns Whittaker's W function.

`WhittakerW`(*parameter*, *parameter*, *argument*)

Examples

```
                                       1
                        4*sqrt(2)*kummeru(---,5,2)
                                       2
WhittakerW(2,2,2);  ⇒  ----------------------------
                                    e
```

Comments

Whittaker's W function is one of the Confluent Hypergeometric functions. For reference see the `hypergeometric` operator.

18.21 Airy Functions

18.22 Airy_Ai

AIRY_AI — Operator

The Airy_Ai operator returns the Airy Ai function for a given argument.

Airy_Ai(*argument*)

Examples

```
on complex; on rounded; Airy_Ai(0);
                         ⇒  0.355028053888
Airy_Ai(3.45 + 17.97i);  ⇒
       - 5.5561528511e+9 - 8.80397899932e+9*i
```

18.23 Airy_Bi

AIRY_BI — Operator

The `Airy_Bi` operator returns the Airy Bi function for a given argument.

`Airy_Bi`(*argument*)

Examples

```
Airy_Bi(0);              ⇒   0.614926627446

Airy_Bi(3.45 + 17.97i);  ⇒

       8.80397899932e+9 - 5.5561528511e+9*i
```

18.24 Airy_Aiprime

AIRY_AIPRIME — Operator

The `Airy_Aiprime` operator returns the Airy Aiprime function for a given argument.

`Airy_Aiprime`(*argument*)

Examples

```
Airy_Aiprime(0);            ⇒   - 0.258819403793

Airy_Aiprime(3.45+17.97i);  ⇒

       - 3.83386421824e+19 + 2.16608828136e+19*i
```

18.25 Airy_Biprime

AIRY_BIPRIME Operator

The `Airy_Biprime` operator returns the Airy Biprime function for a given argument.

`Airy_Biprime`(*argument*)

Examples

```
Airy_Biprime(0);          ⇒
Airy_Biprime(3.45 + 17.97i);
                          ⇒
      3.84251916792e+19 - 2.18006297399e+19*i
```

18.26 Jacobi's Elliptic Functions and Elliptic Integrals

18.27 JacobiSN

JACOBISN Operator

The `Jacobisn` operator returns the Jacobi Elliptic function sn.

`Jacobisn`(*expression*, *integer*)

Examples

`Jacobisn(0.672, 0.36)`	$\Rightarrow$	`0.609519691792`
`Jacobisn(1,0.9)`	$\Rightarrow$	`0.770085724907881`

18.28 JacobiCN

JACOBICN Operator

The `Jacobicn` operator returns the Jacobi Elliptic function cn.

`Jacobicn`(*expression*, *integer*)

Examples

`Jacobicn(7.2, 0.6)` ⇒ `0.837288298482018`

`Jacobicn(0.11, 19)` ⇒

`0.994403862690043 - 1.6219006985556e-16*i`

18.29 JacobiDN

JACOBIDN Operator

The `Jacobidn` operator returns the Jacobi Elliptic function dn.

`Jacobidn`(*expression*, *integer*)

Examples

`Jacobidn(15, 0.683)` ⇒ `0.640574162024592`

`Jacobidn(0,0)` ⇒ `1`

18.30 JacobiCD

JACOBICD Operator

The `Jacobicd` operator returns the Jacobi Elliptic function cd.

`Jacobicd`(*expression*, *integer*)

Examples

`Jacobicd(1, 0.34)` $\Rightarrow$ `0.657683337805273`

`Jacobicd(0.8,0.8)` $\Rightarrow$ `0.925587311582301`

18.31 JacobiSD

JACOBISD Operator

The `Jacobisd` operator returns the Jacobi Elliptic function sd.

`Jacobisd`(*expression*, *integer*)

Examples

`Jacobisd(12, 0.4)` ⇒ `0.357189729437272`

`Jacobisd(0.35,1)` ⇒ `- 1.17713873203043`

18.32 JacobiND

JACOBIND Operator

The `Jacobind` operator returns the Jacobi Elliptic function nd.

`Jacobind`(*expression*, *integer*)

Examples

`Jacobind(0.2, 17)` ⇒

`1.46553203037507 + 0.0000000000334032759313703*i`

`Jacobind(30, 0.001)` ⇒ `1.00048958438`

18.33 JacobiDC

JACOBIDC Operator

The `Jacobidc` operator returns the Jacobi Elliptic function dc.

`Jacobidc`(*expression*, *integer*)

Examples

`Jacobidc(0.003,1)` $\Rightarrow$ `1`

`Jacobidc(2, 0.75)` $\Rightarrow$ `6.43472885111`

18.34 JacobiNC

JACOBINC Operator

The `Jacobinc` operator returns the Jacobi Elliptic function nc.

`Jacobinc`(*expression*, *integer*)

Examples

`Jacobinc(1,0)`	$\Rightarrow$	`1.85081571768093`
`Jacobinc(56, 0.4387)`	$\Rightarrow$	`39.304842663512`

18.35 JacobiSC

JACOBISC Operator

The `Jacobisc` operator returns the Jacobi Elliptic function sc.

`Jacobisc`(*expression*, *integer*)

Examples

`Jacobisc(9, 0.88)` ⇒ `- 1.16417697982095`

`Jacobisc(0.34, 7)` ⇒

`0.305851938390775 - 9.8768100944891e-12*i`

18.36 JacobiNS

JACOBINS — Operator

The `Jacobins` operator returns the Jacobi Elliptic function ns.

`Jacobins`(*expression*, *integer*)

Examples

`Jacobins(3, 0.9)` ⇒ `1.00945801599785`

`Jacobins(0.887, 15)` ⇒

`0.683578280513975 - 0.85023411082469*i`

18.37 JacobiDS

JACOBIDS Operator

The `Jacobisn` operator returns the Jacobi Elliptic function ds.

`Jacobids`(*expression*, *integer*)

Examples

`Jacobids(98,0.223)` ⇒ `- 1.061253961477`

`Jacobids(0.36,0.6)` ⇒ `2.76693172243692`

18.38 JacobiCS

JACOBICS Operator

The `Jacobics` operator returns the Jacobi Elliptic function cs.

`Jacobics`(*expression*, *integer*)

Examples

`Jacobics(0, 0.767)` ⇒ `infinity`

`Jacobics(1.43, 0)` ⇒ `0.141734127352112`

18.39 JacobiAMPLITUDE

JACOBIAMPLITUDE Operator

The `JacobiAmplitude` operator returns the amplitude of u.

`JacobiAmplitude`(*expression*, *integer*)

Examples

```
JacobiAmplitude(7.239, 0.427)
                          ⇒   0.0520978301448978
JacobiAmplitude(0,0.1)    ⇒   0
```

Comments

Amplitude u = asin(`Jacobisn(u,m)`)

18.40 AGM_FUNCTION

AGM_FUNCTION Operator

The `AGM_function` operator returns a list of (N, AGM, list of aNtoa0, list of bNtob0, list of cNtoc0) where a0, b0 and c0 are the initial values; N is the index number of the last term used to generate the AGM. AGM is the Arithmetic Geometric Mean.

`AGM_function`(*integer*, *integer*, *integer*)

Examples

`AGM_function(1,1,1)` ⇒ `1,1,1,1,1,1,0,1`

`AGM_function(1, 0.1, 1.3)` ⇒

```
{6,
 2.27985615996629,
 {2.27985615996629, 2.27985615996629,
  2.2798561599706, 2.2798624278857,
  2.28742283656583, 2.55, 1},
 {2.27985615996629, 2.27985615996629,
  2.27985615996198, 2.2798498920555,
  2.27230201920557, 2.02484567313166, 4.1},
 {0, 4.30803136219904e-12, 0.0000062679151007581,
  0.00756040868012758, 0.262577163434171, - 1.55, 5.9}}
```

Comments

The other Jacobi functions use this function with initial values a0=1, b0=sqrt(1-m), c0=sqrt(m).

18.41 LANDENTRANS

LANDENTRANS Operator

The `landentrans` operator generates the descending landen transformation of the given imput values, returning a list of these values; initial to final in each case.

`landentrans`(*expression*, *integer*)

Examples

```
landentrans(0,0.1)  ⇒

       {{0,0,0,0,0},{0.1,0.0025041751943776,
                    ⇒
       0.00000156772498954046,6.1444078 9914461e-13,0}}
```

Comments

The first list ascends in value, and the second descends in value.

18.42 EllipticF

ELLIPTICF Operator

The `EllipticF` operator returns the Elliptic Integral of the First Kind.

`EllitpicF`(*expression*, *integer*)

Examples

`EllipticF(0.3, 8.222)` ⇒ `0.3`

`EllipticF(7.396, 0.1)` ⇒ `7.58123216114307`

Comments

The Complete Elliptic Integral of the First Kind can be found by putting the first argument to pi/2 or by using `EllipticK` and the second argument.

18.43 EllipticK

ELLIPTICK Operator

The `EllipticK` operator returns the Elliptic value K.

`EllipticK`(*integer*)

Examples

```
EllipticK(0.2)       ⇒   1.65962359861053
EllipticK(4.3)       ⇒
       0.808442364282734 - 1.05562492399206*i
EllipticK(0.000481)  ⇒   1.57098526617635
```

Comments

The `EllipticK` function is the Complete Elliptic Integral of the First Kind.

18.44 EllipticKprime

ELLIPTICKPRIME — Operator

The `EllipticK'` operator returns the Elliptic value K(m).

`EllipticKprime`(*integer*)

Examples

`EllipticKprime(0.2)`	⇒	`2.25720532682085`
`EllipticKprime(4.3)`	⇒	`1.05562492399206`
`EllipticKprime(0.000481)`	⇒	`5.206621921966`

Comments

The `EllipticKprime` function is the Complete Elliptic Integral of the First Kind of (1-m).

18.45 EllipticE

ELLIPTICE Operator

The `EllipticE` operator used with two arguments returns the Elliptic Integral of the Second Kind.

`EllipticE`(*expression*, *integer*)

Examples

`EllipticE(1.2,0.22)`	⇒	`1.15094019180949`
`EllipticE(0,4.35)`	⇒	`0`
`EllipticE(9,0.00719)`	⇒	`8.98312465929145`

Comments

The Complete Elliptic Integral of the Second Kind can be obtained by using just the second argument, or by using pi/2 as the first argument.

The `EllipticE` operator used with one argument returns the Elliptic value E.

`EllipticE`(*integer*)

Examples

`EllipticE(0.22)`	⇒	`1.48046637439519`
`EllipticE(pi/2, 0.22)`	⇒	`1.48046637439519`

18.46 EllipticTHETA

ELLIPTICTHETA Operator

The `EllipticTheta` operator returns one of the four Theta functions. It cannot except any number other than 1,2,3 or 4 as its first argument.

`EllipticTheta`(*integer*, *expression*, *integer*)

Examples

```
EllipticTheta(1, 1.4, 0.72)  ⇒  0.91634775373
EllipticTheta(2, 3.9, 6.1 )  ⇒  -48.0202736969 + 20.9881034377 i
EllipticTheta(3, 0.67, 0.2)  ⇒  1.0083077448
EllipticTheta(4, 8, 0.75)    ⇒  0.894963369304
EllipticTheta(5, 1, 0.1)     ⇒
       ***** In EllipticTheta(a,u,m); a = 1,2,3 or 4.
```

Comments

Theta functions are important because every one of the Jacobian Elliptic functions can be expressed as the ratio of two theta functions.

18.47 JacobiZETA

JACOBIZETA Operator

The `JacobiZeta` operator returns the Jacobian function Zeta.

`JacobiZeta`(*expression*, *integer*)

Examples

`JacobiZeta(3.2, 0.8)` $\Rightarrow$ `- 0.254536403439`

`JacobiZeta(0.2, 1.6)` $\Rightarrow$

`0.171766095970451 - 0.0717028569800147*i`

Comments

The Jacobian function Zeta is related to the Jacobian function Theta. But it is significantly different from Riemann's Zeta Function `Zeta`.

18.48 Gamma and Related Functions

18.49 POCHHAMMER

POCHHAMMER Operator

The `Pochhammer` operator implements the Pochhammer notation (shifted factorial).

`Pochhammer`(*expression*, *expression*)

Examples

```
pochhammer(17,4);   ⇒   116280
                           factorial(2*z)
pochhammer(1/2,z);  ⇒   ---------------------
                          2*z
                        (2   *factorial(z))
```

Comments

A number of complex rules for `Pochhammer` are inactive, because they cause a huge system load in algebraic mode. If one wants to use more rules for the simplification of Pochhammer's notation, one can do:
let special!*pochhammer!*rules;

18.50 GAMMA

GAMMA — Operator

The `Gamma` operator returns the Gamma function.

`Gamma`(*expression*)

Examples

`gamma(10);` $\Rightarrow$ `362880`

`gamma(1/2);` $\Rightarrow$ `sqrt(pi)`

18.51 BETA

BETA Operator

The `Beta` operator returns the Beta function defined by

Beta (z,w) := defint(t**(z-1)* (1 - t)**(w-1),t,0,1) .

`Beta`(*expression*, *expression*)

Examples

`Beta(2,2);` ⇒ `1 / 6`

`Beta(x,y);` ⇒ `gamma(x)*gamma(y) / gamma(x + y)`

Comments

The operator `Beta` is simplified towards the `GAMMA` operator.

18.52 PSI

PSI Operator

The `Psi` operator returns the Psi (or DiGamma) function.

Psi(x) := df(Gamma(z),z)/ Gamma (z)

`Gamma`(*expression*)

Examples

`Psi(3);` ⇒

```
(2*log(2) + psi(1/2) + psi(1) + 3)/2
```

`on rounded;`

`- Psi(1);` ⇒ `0.577215664902`

Comments

Euler's constant can be found as - Psi(1).

18.53 POLYGAMMA

POLYGAMMA — Operator

The `Polygamma` operator returns the Polygamma function.

Polygamma(n,x) := df(Psi(z),z,n);

`Polygamma`(*integer*, *expression*)

Examples

```
                                 2
Polygamma(1,2);      ⇒    (pi  - 6) / 6

on rounded;

Polygamma(1,2.35);  ⇒    0.52849689109
```

Comments

The Polygamma function is used for simplification of the `ZETA` function for some arguments.

18.54 Miscellaneous Functions

18.55 DILOG extended

DILOG EXTENDED — Operator

The package `specfn` supplies an extended support for the `dilog` operator which implements the `dilogarithm function`.

dilog(x) := - defint(log(t)/(t - 1),t,1,x);

`Dilog`(*order*, *expression*)

Examples

```
defint(log(t)/(t - 1),t,1,x);
                           ⇒   - dilog (x)
                                   2
dilog 2;                   ⇒   - pi  /12

on rounded;

Dilog 20;                  ⇒   - 5.92783972438
```

Comments

The operator `Dilog` is sometimes called Spence's Integral for n = 2.

18.56 Lambert_W function

LAMBERT_W FUNCTION Operator

Lambert's W function is the inverse of the function w * e**w. It is used in the `solve` package for equations containing exponentials and logarithms.

`Lambert_W`(z)

Examples

```
Lambert_W(-1/e);        ⇒   -1

solve(w + log(w),w);    ⇒   w=lambert_w(1)

on rounded;

Lambert_W(-0.05);       ⇒   - 0.0527059835515
```

Comments

The current implementation will compute the principal branch in rounded mode only.

18.57 Orthogonal Polynomials

18.58 ChebyshevT

CHEBYSHEVT — Operator

The `ChebyshevT` operator computes the nth Chebyshev T Polynomial (of the first kind).

`ChebyshevT`(*integer*, *expression*)

Examples

```
                          2
ChebyshevT(3,xx);  ⇒  xx*(4*xx  - 3)

ChebyshevT(3,4);   ⇒  244
```

Comments

Chebyshev's T polynomials are computed using the recurrence relation:

ChebyshevT(n,x) := 2x*ChebyshevT(n-1,x) - ChebyshevT(n-2,x) with ChebyshevT(0,x) := 0 and ChebyshevT(1,x) := x

18.59 ChebyshevU

CHEBYSHEVU Operator

The `ChebyshevU` operator returns the nth Chebyshev U Polynomial (of the second kind).

`ChebyshevU`(*integer*, *expression*)

Examples

```
                          2
ChebyshevU(3,xx);  ⇒   4*x*(2*x  - 1)

ChebyshevU(3,4);   ⇒   496
```

Comments

Chebyshev's U polynomials are computed using the recurrence relation:

ChebyshevU(n,x) := 2x*ChebyshevU(n-1,x) - ChebyshevU(n-2,x) with ChebyshevU(0,x) := 0 and ChebyshevU(1,x) := 2x

18.60 HermiteP

HERMITEP   Operator

The `HermiteP` operator returns the nth Hermite Polynomial.

`HermiteP`(*integer*, *expression*)

Examples

```
                                 2
HermiteP(3,xx);  ⇒   4*xx*(2*xx  - 3)

HermiteP(3,4);   ⇒   464
```

Comments

Hermite polynomials are computed using the recurrence relation:
HermiteP(n,x) := 2x*HermiteP(n-1,x) - 2*(n-1)*HermiteP(n-2,x) with
HermiteP(0,x) := 1 and HermiteP(1,x) := 2x

18.61 LaguerreP

LAGUERREP Operator

The `LaguerreP` operator computes the nth Laguerre Polynomial. The two argument call of LaguerreP is a (common) abbreviation of LaguerreP(n,0,x).

`LaguerreP`(*integer*, *expression*) or
`LaguerreP`(*integer*, *expression*, *expression*)

Examples

```
                           3        2
LaguerreP(3,xx);   ⇒   (- xx  + 9*xx  - 18*xx + 6)/6

LaguerreP(2,3,4);  ⇒   -2
```

Comments

Laguerre polynomials are computed using the recurrence relation:

LaguerreP(n,a,x) := (2n+a-1-x)/n*LaguerreP(n-1,a,x) - (n+a-1) * LaguerreP(n-2,a,x) with

LaguerreP(0,a,x) := 1 and LaguerreP(2,a,x) := -x+1+a

18.62 LegendreP

LEGENDREP Operator

The binary `LegendreP` operator computes the nth Legendre Polynomial which is a special case of the nth Jacobi Polynomial with

LegendreP(n,x) := JacobiP(n,0,0,x)

The ternary form returns the associated Legendre Polynomial (see below).

`LegendreP`(*integer*, *expression*) or
`LegendreP`(*integer*, *expression*, *expression*)

Examples

```
                                 2
                       xx*(5*xx  - 3)
LegendreP(3,xx);    ⇒  ----------------
                              2
                                    2
LegendreP(3,2,xx);  ⇒  15*xx*( - xx  + 1)
```

Comments

The ternary form of the operator `LegendreP` is the associated Legendre Polynomial defined as

P(n,m,x) = (-1)**m * (1-x**2)**(m/2) * df(LegendreP(n,x),x,m)

18.63 JacobiP

JACOBIP Operator

The `JacobiP` operator computes the nth Jacobi Polynomial.

`JacobiP`(*integer*, *expression*, *expression*, *expression*)

Examples

```
                                  3         2
                        7*(65*xx  - 13*xx  - 13*xx + 1)
JacobiP(3,4,5,xx);  ⇒  ---------------------------------
                                       8
JacobiP(3,4,5,6);   ⇒  94465/8
```

18.64 GegenbauerP

GEGENBAUERP — Operator

The `GegenbauerP` operator computes Gegenbauer's (ultraspherical) polynomials.

`GegenbauerP`(*integer*, *expression*, *expression*)

Examples

```
                                      2
GegenbauerP(3,2,xx);  ⇒   4*xx*(8*xx  - 3)

GegenbauerP(3,2,4);   ⇒   2000
```

18.65 SolidHarmonicY

SOLIDHARMONICY Operator

The `SolidHarmonicY` operator computes Solid harmonic (Laplace) polynomials.

`SolidHarmonicY`(*integer*, *integer*, *expression*, *expression*, *expression*, *expression*)

Examples

```
SolidHarmonicY(3,-2,x,y,z,r2);
```

```
                                             2    2
         sqrt(105)*z*(-2*i*x*y + x  - y )
⇒      ---------------------------------
                4*sqrt(pi)*sqrt(2)
```

18.66 SphericalHarmonicY

SPHERICALHARMONICY Operator

The `SphericalHarmonicY` operator computes Spherical harmonic (Laplace) polynomials. These are special cases of the solid harmonic polynomials, `SolidHarmonicY`.

`SphericalHarmonicY`(*integer*, *integer*, *expression*, *expression*)

Examples

```
SphericalHarmonicY(3,2,theta,phi);
```

$\Rightarrow$

```
                                   2           2                                2
 sqrt(105)*cos(theta)*sin(theta) *(cos(phi) +2*cos(phi)*sin(phi)*i- sin(phi) )
-------------------------------------------------------------------------------
                              4*sqrt(pi)*sqrt(2)
```

18.67 Integral Functions

18.68 Si

SI Operator

The `Si` operator returns the Sine Integral function.

`Si`(*expression*)

Examples

`limit(Si(x),x,infinity);` ⇒ `pi / 2`

`on rounded;`

`Si(0.35);` ⇒ `0.347626790989`

Comments

The numeric values for the operator `Si` are computed via the power series representation, which limits the argument range.

18.69 Shi

SHI — Operator

The `Shi` operator returns the hyperbolic Sine Integral function.

`Shi`(*expression*)

Examples

```
df(shi(x),x);  ⇒  sinh(x) / x

on rounded;

Shi(0.35);     ⇒  0.352390716351
```

Comments

The numeric values for the operator `Shi` are computed via the power series representation, which limits the argument range.

18.70 s_i

S_I Operator

The `s_i` operator returns the Sine Integral function si.

`s_i`(*expression*)

Examples

```
s_i(xx);          ⇒   (2*Si(xx) - pi) / 2

df(s_i(x),x);     ⇒   sin(x) / x
```

Comments

The operator name `s_i` is simplified towards `SI`. Since REDUCE is not case sensitive by default the name "si" can't be used.

18.71 Ci

CI Operator

The `Ci` operator returns the Cosine Integral function.

`Ci`(*expression*)

Examples

```
defint(cos(t)/t,t,x,infinity);
                        ⇒  - ci (x)
on rounded;
Ci(0.35);               ⇒  - 0.50307556932
```

Comments

The numeric values for the operator `Ci` are computed via the power series representation, which limits the argument range.

18.72 Chi

CHI Operator

The `Chi` operator returns the Hyperbolic Cosine Integral function.

`Chi`(*expression*)

Examples

```
defint((cosh(t)-1)/t,t,0,x);

                        ⇒   - log(x) + psi(1) + chi(x)

on rounded;

Chi(0.35);              ⇒   - 0.44182471827
```

Comments

The numeric values for the operator `Chi` are computed via the power series representation, which limits the argument range.

18.73 ERF extended

ERF EXTENDED — Operator

The special function package supplies an extended support for the `erf` operator which implements the `error function`

defint(e**(-x**2),x,0,infinity) * 2/sqrt(pi)

.

`erf`(*expression*)

Examples

```
erf(-x);      ⇒   - erf(x)

on rounded;

erf(0.35);    ⇒   0.379382053562
```

Comments

The numeric values for the operator `erf` are computed via the power series representation, which limits the argument range.

18.74 erfc

ERFC Operator

The `erfc` operator returns the complementary Error function

1 - defint(e**(-x**2),x,0,infinity) * 2/sqrt(pi)
.

`erfc`(*expression*)

Examples

`erfc(xx);` ⇒ `- erf(xx) + 1`

Comments

The operator `erfc` is simplified towards the `erf` operator.

18.75 Ei

EI Operator

The `Ei` operator returns the Exponential Integral function.

`Ei`(*expression*)

Examples

```
                   x
                  e
df(ei(x),x);  ⇒  ----
                   x
```

```
on rounded;
```

```
Ei(0.35);     ⇒  - 0.0894340019184
```

Comments

The numeric values for the operator `Ei` are computed via the power series representation, which limits the argument range.

18.76 Fresnel_C

FRESNEL_C Operator

The `Fresnel_C` operator represents Fresnel's Cosine function.

`Fresnel_C`(*expression*)

Examples

`int(cos(t^2*pi/2),t,0,x);`	⇒	`fresnel_c(x)`
`on rounded;`		
`fresnel_c(2.1);`	⇒	`0.581564135061`

Comments

The operator `Fresnel_C` has a limited numeric evaluation of large values of its argument.

18.77 Fresnel_S

FRESNEL_S Operator

The `Fresnel_S` operator represents Fresnel's Sine Integral function.

`Fresnel_S`(*expression*)

Examples

```
int(sin(t^2*pi/2),t,0,x);  ⇒  fresnel_s(x)

on rounded;

fresnel_s(2.1);            ⇒  0.374273359378
```

Comments

The operator `Fresnel_S` has a limited numeric evaluation of large values of its argument.

18.78 Combinatorial Operators

18.79 BINOMIAL

BINOMIAL Operator

The `Binomial` operator returns the Binomial coefficient if both parameter are integer and expressions involving the Gamma function otherwise.

`Binomial`(*integer*, *integer*)

Examples

```
Binomial(49,6);  ⇒   13983816
                       gamma(n + 1)
Binomial(n,3);   ⇒   ----------------
                      6*gamma(n - 2)
```

Comments

The operator `Binomial` evaluates the Binomial coefficients from the explicit form and therefore it is not the best algorithm if you want to compute many binomial coefficients with big indices in which case a recursive algorithm is preferable.

18.80 STIRLING1

STIRLING1 Operator

The `Stirling1` operator returns the Stirling Numbers S(n,m) of the first kind, i.e. the number of permutations of n symbols which have exactly m cycles (divided by (-1)**(n-m)).

`Stirling1`(*integer*, *integer*)

Examples

```
Stirling1 (17,4);    ⇒   -87077748875904
                           -gamma(n+1)
Stirling1 (n,n-1);   ⇒   --------------
                          2*gamma(n-1)
```

Comments

The operator `Stirling1` evaluates the Stirling numbers of the first kind by rulesets for special cases or by a computing the closed form, which is a series involving the operators `BINOMIAL` and `STIRLING2`.

18.81 STIRLING2

STIRLING2 Operator

The `Stirling1` operator returns the Stirling Numbers S(n,m) of the second kind, i.e. the number of ways of partitioning a set of n elements into m non-empty subsets.

`Stirling2`(*integer*, *integer*)

Examples

```
Stirling2 (17,4);   ⇒   694337290
                          gamma(n+1)
Stirling2 (n,n-1);  ⇒   --------------
                         2*gamma(n-1)
```

Comments

The operator `Stirling2` evaluates the Stirling numbers of the second kind by rulesets for special cases or by a computing the closed form.

18.82 3j and 6j symbols

18.83 ThreejSymbol

THREEJSYMBOL Operator

The `ThreejSymbol` operator implements the 3j symbol.

`ThreejSymbol`(*listofj1*, *m1*, *listofj2*, *m2*, *listofj3*, *m3*)

Examples

```
ThreejSymbol({j+1,m},{j+1,-m},{1,0});
```

$\Rightarrow$

```
         j
  ( - 1)  *(abs(j - m + 1) - abs(j + m + 1))
--------------------------------------------
                3      2                  m
   2*sqrt(2*j  + 9*j  + 13*j + 6)*( - 1)
```

18.84 Clebsch_Gordan

CLEBSCH_GORDAN Operator

The `Clebsch_Gordan` operator implements the Clebsch_Gordan coefficients. This is closely related to the `Threejsymbol`.

`Clebsch_Gordan`(*listofj1*, *m1*, *listofj2*, *m2*, *listofj3*, *m3*)

Examples

```
Clebsch_Gordan({2,0},{2,0},{2,0});
```

```
                              -2
                   ⇒      ----------
                           sqrt(14)
```

18.85 SixjSymbol

SIXJSYMBOL Operator

The `SixjSymbol` operator implements the 6j symbol.

`SixjSymbol`(*listofj1*, *j2*, *j3*, *listofl1*, *l2*, *l3*)

Examples

```
SixjSymbol({7,6,3},{2,4,6});
```

```
                 1
    ⇒    --------------
          14*sqrt(858)
```

Comments

The operator `SixjSymbol` uses the `ineq` package in order to find minima and maxima for the summation index.

18.86 Miscellaneous

18.87 HYPERGEOMETRIC

HYPERGEOMETRIC Operator

The `Hypergeometric` operator provides simplifications for the generalized hypergeometric functions. The `Hypergeometric` operator is included in the package specfn2.

`hypergeometric`(*listofparameters*, *listofparameters*, *argument*)

Examples

```
load specfn2;

hypergeometric ({1/2,1},{3/2},-x^2);
                                    atan(x)
                              ⇒    ---------
                                        x
                                     z
hypergeometric ({},{},z);    ⇒    e
```

Comments

The special case where the length of the first list is equal to 2 and the length of the second list is equal to 1 is often called "the hypergeometric function" (notated as 2F1(a1,a2,b;x)).

18.88 MeijerG

MEIJERG Operator

The `MeijerG` operator provides simplifications for Meijer's G function. The simplifications are performed towards polynomials, elementary or special functions or (generalized) `hypergeometric` functions.

The `MeijerG` operator is included in the package specfn2.

`MeijerG`(*listofparameters*, *listofparameters*, *argument*)

The first element of the lists has to be the list containing the first group (mostly called "m" and "n") of parameters. This passes the four parameters of a Meijer's G function implicitly via the length of the lists.

Examples

```
load specfn2;

MeijerG({{},1},{{0}},x);  ⇒  heaviside(-x+1)

MeijerG({{}},{{1+1/4},1-1/4},(x^2)/4) * sqrt pi;
                                          2
                               sqrt(2)*sin(x)*x
                          ⇒   ------------------
                                  4*sqrt(x)
```

Comments

Many well-known functions can be written as G functions, e.g. exponentials, logarithms, trigonometric functions, Bessel functions and hypergeometric functions. The formulae can be found e.g. in

A.P.Prudnikov, Yu.A.Brychkov, O.I.Marichev: Integrals and Series, Volume 3: More special functions, Gordon and Breach Science Publishers (1990).

18.89 Heaviside

HEAVISIDE Operator

The `Heaviside` operator returns the Heaviside function.

Heaviside(w) =¿ if (w ¡ 0) then 0 else 1
when numberp w;

`Heaviside`(*argument*)

Comments

This operator is often included in the result of the simplification of a generalized `hypergeometric` function or a `MeijerG` function.

No simplification is done for this function.

18.90 erfi

ERFI Operator

The `erfi` operator returns the error function of an imaginary argument.

erfi(x) =¿ 2/sqrt(pi) * defint(e**(t**2),t,0,x);

`erfi`(*argument*)

Comments

This operator is sometimes included in the result of the simplification of a generalized `hypergeometric` function or a `MeijerG` function.

No simplification is done for this function.

19 Taylor series

19.1 TAYLOR

TAYLOR Introduction

This short note describes a package of REDUCE procedures that allow Taylor expansion in one or more 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 expansion can be performed as well.

19.2 taylor

TAYLOR   Operator

The `taylor` operator is used for expanding an expression into a Taylor series.

`taylor`(*expression* , *var* , *expression* , *number*
{ , *var* , *expression* , *number*}∗)

expression can be any valid REDUCE algebraic expression. *var* must be a `kernel`, and is the expansion variable. The *expression* following it denotes the point about which the expansion is to take place. *number* must be a non-negative integer and denotes the maximum expansion order. If more than one triple is specified `taylor` will expand its first argument independently with respect to all the variables. 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 the maximum expansion order. If the expansion point evaluates to the special identifier `infinity`, `taylor` tries to expand 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`.

Examples

```
taylor(e^(x^2+y^2),x,0,2,y,0,2);
```

```
                                  2     2     2  2        2  2
                        ⇒   1 + Y  + X  + Y *X  + O(X ,Y )
```

```
taylor(e^(x^2+y^2),{x,y},0,2);
```

```
                                  2     2          2  2
                        ⇒   1 + Y  + X  + O({X ,Y })
```

The following example shows the case of a non-analytical function.

```
taylor(x*y/(x+y),x,0,2,y,0,2);
```

⇒

```
       ***** Not a unit in argument to QUOTTAYLOR
```

Comments

Note that it is not generally possible to apply the standard reduce operators to a Taylor kernel. For example, `part`, `coeff`, or `coeffn` cannot be used. Instead, the expression at hand has to be converted to standard form first using the `taylortostandard` operator.

Differentiation of a Taylor expression is possible. If you differentiate with respect to one of the Taylor variables the order will decrease by one.

Substitution is a bit restricted: Taylor variables can only be replaced by other kernels. There is one exception to this rule: you 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: an example is sin(acos(4)).

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.

19.3 taylorautocombine

TAYLORAUTOCOMBINE Switch

If you set `taylorautocombine` 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`.

19.4 taylorautoexpand

TAYLORAUTOEXPAND Switch

`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`.

19.5 taylorcombine

TAYLORCOMBINE — Operator

This operator tries to combine all Taylor kernels found in its argument into one. Operations currently possible are:

- Addition, subtraction, multiplication, and division.
- Roots, exponentials, and logarithms.
- Trigonometric and hyperbolic functions and their inverses.

Examples

```
hugo := taylor(exp(x),x,0,2);

                                    1   2      3
                  ⇒  HUGO := 1 + X + ---*X  + O(X )
                                    2
                                3
taylorcombine log hugo;   ⇒  X + O(X )

                                1   2      3
taylorcombine(hugo + x);  ⇒  (1 + X + ---*X  + O(X )) + X
                                2

on taylorautoexpand;

                                 1   2      3
taylorcombine(hugo + x);  ⇒  1 + 2*X + ---*X  + O(X )
                                 2
```

Comments

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 its argument 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`.

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.

19.6 taylorkeeporiginal

TAYLORKEEPORIGINAL — Switch

`taylorkeeporiginal`, if set to `on`, forces the `taylor` and all Taylor kernel manipulation operators 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`.

19.7 taylororiginal

TAYLORORIGINAL   Operator

Recovers the original expression (the one that was expanded) from the Taylor kernel that is given as its argument.

`taylororiginal`(*expression*) or `taylororiginal` *simple_expression*

Examples

```
hugo := taylor(exp(x),x,0,2);
```

```
                              1   2      3
   ⇒   HUGO := 1 + X + ---*X  + O(X )
                              2
```

```
taylororiginal hugo;   ⇒
```

```
  ***** Taylor kernel doesn't have an original part in TAYLORORIGINAL
```

```
on taylorkeeporiginal;
```

```
hugo := taylor(exp(x),x,0,2);
```

```
                              1   2      3
   ⇒   HUGO := 1 + X + ---*X  + O(X )
                              2
```

```
                         X
taylororiginal hugo;   ⇒ E
```

Comments

An error is signalled if the argument is not a Taylor kernel or if the original expression was not kept, i.e. if `taylorkeeporiginal` was set `off` during expansion.

19.8 taylorprintorder

TAYLORPRINTORDER — Switch

`taylorprintorder`, if set to `on`, causes the remainder to be printed in big-O notation. Otherwise, three dots are printed. Default is `on`.

19.9 taylorprintterms

TAYLORPRINTTERMS Variable

Only a certain number of (non-zero) coefficients are printed. 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.

Examples

```
taylor(e^(x^2+y^2),x,0,4,y,0,4);
```

⇒

```
          2    1   4    2    2  2                    5   5
     1 + Y  + ---*Y  + X  + Y *X  + (4 terms) + O(X ,Y )
               2
```

```
taylorprintterms := all;  ⇒   TAYLORPRINTTERMS := ALL
```

```
taylor(e^(x^2+y^2),x,0,4,y,0,4);
```

⇒

```
          2    1   4    2    2  2    1   4  2    1   4    1   2  4
     1 + Y  + ---*Y  + X  + Y *X  + ---*Y *X  + ---*X  + ---*Y *X
               2                     2           2        2

         1   4  4      5   5
      + ---*Y *X  + O(X ,Y )
         4
```

19.10 taylorrevert

TAYLORREVERT Operator

`taylorrevert` allows reversion of 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.

`taylorrevert`(*expression*, *var*, *var*)

The first argument must evaluate to a Taylor kernel with the second argument being one of its expansion variables.

Examples

```
                                      2      6
taylor(u - u**2,u,0,5);  ⇒   U - U  + O(U )

taylorrevert (ws,u,x);   ⇒
             2      3      4       5      6
       X + X  + 2*X  + 5*X  + 14*X  + O(X )
```

19.11 taylorseriesp

TAYLORSERIESP   Operator

This operator may be used to determine if its argument is a Taylor kernel.

`taylorseriesp`(*expression*) or `taylorseriesp` *simple_expression*

Examples

```
hugo := taylor(exp(x),x,0,2);
```

```
                              1   2      3
    ⇒   HUGO := 1 + X + ---*X  + O(X )
                              2
```

```
if taylorseriesp hugo then OK;
```

```
    ⇒   OK
```

```
if taylorseriesp(hugo + y) then OK else NO;
```

```
    ⇒   NO
```

Comments

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.

19.12 taylortemplate

TAYLORTEMPLATE Operator

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`(*expression*) or `taylortemplate` *simple_expression*

This returns a list of lists with the three elements (VAR,VAR0,ORDER). An error is signalled if the argument is not a Taylor kernel.

Examples

```
hugo := taylor(exp(x),x,0,2);
```

$\Rightarrow$

```
                  1   2       3
HUGO := 1 + X + ---*X  + O(X )
                  2
```

`taylortemplate hugo;` $\Rightarrow$ `{{X,0,2}}`

19.13 taylortostandard

TAYLORTOSTANDARD — Operator

This operator converts all Taylor kernels in its argument into standard form and resimplifies the result.

`taylortostandard`(*expression*) or `taylortostandard` *simple_expression*

Examples

```
hugo := taylor(exp(x),x,0,2);
```

```
                                   1   2        3
                  ⇒  HUGO := 1 + X + ---*X  + O(X )
                                   2
```

```
                        2
                       X  + 2*X + 2
taylortostandard hugo; ⇒  --------------
                            2
```

Index