Using Yacas, function reference

by the Yacas team [1]

Yacas version: 1.2.1
generated on September 27, 2007

This document describes the functions that are useful in the context of using Yacas as an end user.

Contents

9

Chapter 1

Introduction

Yacas (Yet Another Computer Algebra System) is a small and highly flexible general-purpose computer algebra system and programming language. The language has a familiar, C-like infix-operator syntax. The distribution contains a small library of mathematical functions, but its real strength is in the language in which you can easily write your own symbolic manipulation algorithms. The core engine supports arbitrary precision arithmetic, and is able to execute symbolic manipulations on various mathematical objects by following user-defined rules.

You can use Yacas directly from the web site as a Java applet (it runs inside your browser without needing a connection to a remote server). This means that no additional software needs to be installed (other than a free Java environment if this is not installed yet).

Yacas can also be downloaded. The download contains everything needed to create this entire web site. It also contains the files needed to build an off-line version of Yacas, a native executable that can run on your computer. This is discussed further in another part of the documentation.

This document describes the functions that are useful in the context of using Yacas as an end user. It is recommended to first read the online interactive tutorial to get acquainted with the basic language constructs first. This document expands on the tutorial by explaining the usage of the functions that are useful when doing calculations.

Chapter 2

Arithmetic and other operations on numbers

Besides the usual arithmetical operations, Yacas defines some more advanced operations on numbers. Many of them also work on polynomials.

+ — arithmetic addition

(standard library)

Calling format:

```
x+y
+x
```

Precedence: 70

Parameters:

`x` and `y` – objects for which arithmetic addition is defined

Description:

The addition operators can work on integers, rational numbers, complex numbers, vectors, matrices and lists.

These operators are implemented in the standard math library (as opposed to being built-in). This means that they can be extended by the user.

Examples:

```
In> 2+3
Out> 5;
```

- — arithmetic subtraction or negation

(standard library)

Calling format:

```
x-y
```

Precedence: left-side: 70 , right-side: 40

```
-x
```

Parameters:

`x` and `y` – objects for which subtraction is defined

Description:

The subtraction operators can work on integers, rational numbers, complex numbers, vectors, matrices and lists.

These operators are implemented in the standard math library (as opposed to being built-in). This means that they can be extended by the user.

Examples:

```
In> 2-3
Out> -1;
In> - 3
Out> -3;
```

* — arithmetic multiplication

(standard library)

Calling format:

```
x*y
```

Precedence: 40

Parameters:

`x` and `y` – objects for which arithmetic multiplication is defined

Description:

The multiplication operator can work on integers, rational numbers, complex numbers, vectors, matrices and lists.

This operator is implemented in the standard math library (as opposed to being built-in). This means that they can be extended by the user.

Examples:

```
In> 2*3
Out> 6;
```

/ — arithmetic division

(standard library)

Calling format:

```
x/y
```

Precedence: 30

Parameters:

`x` and `y` – objects for which arithmetic division is defined

Description:

The division operator can work on integers, rational numbers, complex numbers, vectors, matrices and lists.

This operator is implemented in the standard math library (as opposed to being built-in). This means that they can be extended by the user.

Examples:

```
In> 6/2
Out> 3;
```

`^` — arithmetic power

(standard library)

Calling format:

```
x^y
```

Precedence: 20

Parameters:

`x` and `y` – objects for which arithmetic operations are defined

Description:

These are the basic arithmetic operations. They can work on integers, rational numbers, complex numbers, vectors, matrices and lists.

These operators are implemented in the standard math library (as opposed to being built-in). This means that they can be extended by the user.

Examples:

```
In> 2^3
Out> 8;
```

`Div` — Determine divisor of two mathematical objects

`Mod` — Determine remainder of two mathematical objects after dividing one by the other

(standard library)

Calling format:

```
Div(x,y)
Mod(x,y)
```

Parameters:

`x`, `y` – integers or univariate polynomials

Description:

`Div` performs integer division and `Mod` returns the remainder after division. `Div` and `Mod` are also defined for polynomials.

If `Div(x,y)` returns "a" and `Mod(x,y)` equals "b", then these numbers satisfy $x = ay + b$ and $0 \leq b < y$.

Examples:

```
In> Div(5,3)
Out> 1;
In> Mod(5,3)
Out> 2;
```

See also: `Gcd`, `Lcm`

`Gcd` — greatest common divisor

(standard library)

Calling format:

```
Gcd(n,m)
Gcd(list)
```

Parameters:

`n`, `m` – integers or Gaussian integers or univariate polynomials

`list` – a list of all integers or all univariate polynomials

Description:

This function returns the greatest common divisor of "n" and "m". The gcd is the largest number that divides "n" and "m". It is also known as the highest common factor (hcf). The library code calls `MathGcd`, which is an internal function. This function implements the "binary Euclidean algorithm" for determining the greatest common divisor:

Routine for calculating `Gcd(n,m)`

1. if $n = m$ then return n
2. if both n and m are even then return $2\mathrm{Gcd}\left(\frac{n}{2}, \frac{m}{2}\right)$
3. if exactly one of n or m (say n) is even then return $\mathrm{Gcd}\left(\frac{n}{2}, m\right)$
4. if both n and m are odd and, say, $n > m$ then return $\mathrm{Gcd}\left(\frac{n-m}{2}, m\right)$

This is a rather fast algorithm on computers that can efficiently shift integers. When factoring Gaussian integers, a slower recursive algorithm is used.

If the second calling form is used, `Gcd` will return the greatest common divisor of all the integers or polynomials in "list". It uses the identity

$$\mathrm{Gcd}\left(a, b, c\right) = \mathrm{Gcd}\left(\mathrm{Gcd}\left(a, b\right), c\right).$$

Examples:

```
In> Gcd(55,10)
Out> 5;
In> Gcd({60,24,120})
Out> 12;
In> Gcd( 7300 + 12*I, 2700 + 100*I)
Out> Complex(-4,4);
```

See also: `Lcm`

`Lcm` — least common multiple

(standard library)

Calling format:

```
Lcm(n,m)
Lcm(list)
```

Parameters:

`n`, `m` – integers or univariate polynomials `list` – list of integers

Description:

This command returns the least common multiple of “n” and “m” or all of the integers in the list `list`. The least common multiple of two numbers “n” and “m” is the lowest number which is an integer multiple of both “n” and “m”. It is calculated with the formula

$$\mathrm{Lcm}\left(n,m\right)=\mathrm{Div}\left(nm,\mathrm{Gcd}\left(n,m\right)\right).$$

This means it also works on polynomials, since `Div`, `Gcd` and multiplication are also defined for them.

Examples:

```
In> Lcm(60,24)
Out> 120;
In> Lcm({3,5,7,9})
Out> 315;
```

See also: `Gcd`

`<<` — binary shift left operator

`>>` — binary shift right operator

(standard library)

Calling format:

```
n<<m
n>>m
```

Parameters:

`n`, `m` – integers

Description:

These operators shift integers to the left or to the right. They are similar to the C shift operators. These are sign-extended shifts, so they act as multiplication or division by powers of 2.

Examples:

```
In> 1 << 10
Out> 1024;
In> -1024 >> 10
Out> -1;
```

`FromBase` — conversion of a number from non-decimal base to decimal base

`ToBase` — conversion of a number in decimal base to non-decimal base

(Yacas internal)

Calling format:

```
FromBase(base,"string")
ToBase(base, number)
```

Parameters:

`base` – integer, base to convert to/from

`number` – integer, number to write out in a different base

`"string"` – string representing a number in a different base

Description:

In Yacas, all numbers are written in decimal notation (base 10). The two functions `FromBase`, `ToBase` convert numbers between base 10 and a different base. Numbers in non-decimal notation are represented by strings.

`FromBase` converts an integer, written as a string in base `base`, to base 10. `ToBase` converts `number`, written in base 10, to base `base`.

Non-integer arguments are not supported.

Examples:

Write the binary number `111111` as a decimal number:

```
In> FromBase(2,"111111")
Out> 63;
```

Write the (decimal) number `255` in hexadecimal notation:

```
In> ToBase(16,255)
Out> "ff";
```

See also: `PAdicExpand`

`N` — try determine numerical approximation of expression

(standard library)

Calling format:

```
N(expression)
N(expression, precision)
```

Parameters:

`expression` – expression to evaluate

`precision` – integer, precision to use

Description:

The function N instructs Yacas to try to coerce an expression in to a numerical approximation to the expression `expr`, using `prec` digits precision if the second calling sequence is used, and the default precision otherwise. This overrides the normal behaviour, in which expressions are kept in symbolic form (eg. `Sqrt(2)` instead of `1.41421`).

Application of the `N` operator will make Yacas calculate floating point representations of functions whenever possible. In addition, the variable `Pi` is bound to the value of π calculated at the current precision. (This value is a "cached constant", so it is not recalculated each time `N` is used, unless the precision is increased.)

`N` is a macro. Its argument `expr` will only be evaluated after switching to numeric mode.

Examples:

```
In> 1/2
Out> 1/2;
In> N(1/2)
Out> 0.5;
In> Sin(1)
Out> Sin(1);
In> N(Sin(1),10)
Out> 0.8414709848;
In> Pi
Out> Pi;
In> N(Pi,20)
Out> 3.14159265358979323846;
```

See also: `Pi`

`Rationalize` — convert floating point numbers to fractions

(standard library)

Calling format:

```
Rationalize(expr)
```

Parameters:

`expr` – an expression containing real numbers

Description:

This command converts every real number in the expression "expr" into a rational number. This is useful when a calculation needs to be done on floating point numbers and the algorithm is unstable. Converting the floating point numbers to rational numbers will force calculations to be done with infinite precision (by using rational numbers as representations).

It does this by finding the smallest integer n such that multiplying the number with 10^n is an integer. Then it divides by 10^n again, depending on the internal gcd calculation to reduce the resulting division of integers.

Examples:

```
In> {1.2,3.123,4.5}
Out> {1.2,3.123,4.5};
In> Rationalize(%)
Out> {6/5,3123/1000,9/2};
```

See also: `IsRational`

`ContFrac` — continued fraction expansion

(standard library)

Calling format:

```
ContFrac(x)
ContFrac(x, depth)
```

Parameters:

`x` – number or polynomial to expand in continued fractions

`depth` – integer, maximum required depth of result

Description:

This command returns the continued fraction expansion of `x`, which should be either a floating point number or a polynomial. If `depth` is not specified, it defaults to 6. The remainder is denoted by `rest`.

This is especially useful for polynomials, since series expansions that converge slowly will typically converge a lot faster if calculated using a continued fraction expansion.

Examples:

```
In> PrettyForm(ContFrac(N(Pi)))

             1
--------------------------- + 3
           1
----------------------- + 7
        1
------------------ + 15
      1
-------------- + 1
   1
-------- + 292
rest + 1

Out> True;
In> PrettyForm(ContFrac(x^2+x+1, 3))

       x
---------------- + 1
         x
1 - ------------
       x
    -------- + 1
    rest + 1

Out> True;
```

See also: `PAdicExpand`, `N`

`Decimal` — decimal representation of a rational

(standard library)

Calling format:

```
Decimal(frac)
```

Parameters:

`frac` – a rational number

Description:

This function returns the infinite decimal representation of a rational number `frac`. It returns a list, with the first element being the number before the decimal point and the last element the sequence of digits that will repeat forever. All the intermediate list elements are the initial digits before the period sets in.

Examples:

```
In> Decimal(1/22)
Out> {0,0,{4,5}};
In> N(1/22,30)
Out> 0.045454545454545454545454545454;
```

See also: `N`

`Floor` — round a number downwards

(standard library)

Calling format:

```
Floor(x)
```

Parameters:

`x` – a number

Description:

This function returns $\lfloor x \rfloor$, the largest integer smaller than or equal to x.

Examples:

```
In> Floor(1.1)
Out> 1;
In> Floor(-1.1)
Out> -2;
```

See also: `Ceil`, `Round`

`Ceil` — round a number upwards

(standard library)

Calling format:

```
Ceil(x)
```

Parameters:

`x` – a number

Description:

This function returns $\lceil x \rceil$, the smallest integer larger than or equal to x.

Examples:

```
In> Ceil(1.1)
Out> 2;
In> Ceil(-1.1)
Out> -1;
```

See also: `Floor`, `Round`

`Round` — round a number to the nearest integer

(standard library)

Calling format:

```
Round(x)
```

Parameters:

`x` – a number

Description:

This function returns the integer closest to x. Half-integers (i.e. numbers of the form $n + 0.5$, with n an integer) are rounded upwards.

Examples:

```
In> Round(1.49)
Out> 1;
In> Round(1.51)
Out> 2;
In> Round(-1.49)
Out> -1;
In> Round(-1.51)
Out> -2;
```

See also: `Floor`, `Ceil`

`Min` — minimum of a number of values

(standard library)

Calling format:

```
Min(x,y)
Min(list)
```

Parameters:

`x`, `y` – pair of values to determine the minimum of
`list` – list of values from which the minimum is sought

Description:

This function returns the minimum value of its argument(s). If the first calling sequence is used, the smaller of "x" and "y" is returned. If one uses the second form, the smallest of the entries in "list" is returned. In both cases, this function can only be used with numerical values and not with symbolic arguments.

Examples:

```
In> Min(2,3);
Out> 2;
In> Min({5,8,4});
Out> 4;
```

See also: `Max`, `Sum`

Max — maximum of a number of values

(standard library)

Calling format:

```
Max(x,y)
Max(list)
```

Parameters:

x, y – pair of values to determine the maximum of
list – list of values from which the maximum is sought

Description:

This function returns the maximum value of its argument(s). If the first calling sequence is used, the larger of "x" and "y" is returned. If one uses the second form, the largest of the entries in "list" is returned. In both cases, this function can only be used with numerical values and not with symbolic arguments.

Examples:

```
In> Max(2,3);
Out> 3;
In> Max({5,8,4});
Out> 8;
```

See also: Min, Sum

Numer — numerator of an expression

(standard library)

Calling format:

```
Numer(expr)
```

Parameters:

expr – expression to determine numerator of

Description:

This function determines the numerator of the rational expression "expr" and returns it. As a special case, if its argument is numeric but not rational, it returns this number. If "expr" is neither rational nor numeric, the function returns unevaluated.

Examples:

```
In> Numer(2/7)
Out> 2;
In> Numer(a / x^2)
Out> a;
In> Numer(5)
Out> 5;
```

See also: Denom, IsRational, IsNumber

Denom — denominator of an expression

(standard library)

Calling format:

```
Denom(expr)
```

Parameters:

expr – expression to determine denominator of

Description:

This function determines the denominator of the rational expression "expr" and returns it. As a special case, if its argument is numeric but not rational, it returns 1. If "expr" is neither rational nor numeric, the function returns unevaluated.

Examples:

```
In> Denom(2/7)
Out> 7;
In> Denom(a / x^2)
Out> x^2;
In> Denom(5)
Out> 1;
```

See also: Numer, IsRational, IsNumber

Pslq — search for integer relations between reals

(standard library)

Calling format:

```
Pslq(xlist,precision)
```

Parameters:

xlist – list of numbers
precision – required number of digits precision of calculation

Description:

This function is an integer relation detection algorithm. This means that, given the numbers x_i in the list "xlist", it tries to find integer coefficients a_i such that $a_1 x_1 + ... + a_n x_n = 0$. The list of integer coefficients is returned.

The numbers in "xlist" must evaluate to floating point numbers if the N operator is applied on them.

Example:

```
In> Pslq({ 2*Pi+3*Exp(1), Pi, Exp(1) },20)
Out> {1,-2,-3};
```

Note: in this example the system detects correctly that $1(2\pi + 3e) + (-2)\pi + (-3)e = 0$.

See also: N

Chapter 3

Predicates relating to numbers

< — test for "less than"

(standard library)

Calling format:

```
e1 < e2
```

Precedence: 90

Parameters:

`e1`, `e2` – expressions to be compared

Description:

The two expression are evaluated. If both results are numeric, they are compared. If the first expression is smaller than the second one, the result is `True` and it is `False` otherwise. If either of the expression is not numeric, after evaluation, the expression is returned with evaluated arguments.

The word "numeric" in the previous paragraph has the following meaning. An expression is numeric if it is either a number (i.e. `IsNumber` returns `True`), or the quotient of two numbers, or an infinity (i.e. `IsInfinity` returns `True`). Yacas will try to coerce the arguments passed to this comparison operator to a real value before making the comparison.

Examples:

```
In> 2 < 5;
Out> True;
In> Cos(1) < 5;
Out> True;
```

See also: `IsNumber`, `IsInfinity`, `N`

> — test for "greater than"

(standard library)

Calling format:

```
e1 > e2
```

Precedence: 90

Parameters:

`e1`, `e2` – expressions to be compared

Description:

The two expression are evaluated. If both results are numeric, they are compared. If the first expression is larger than the second one, the result is `True` and it is `False` otherwise. If either of the expression is not numeric, after evaluation, the expression is returned with evaluated arguments.

The word "numeric" in the previous paragraph has the following meaning. An expression is numeric if it is either a number (i.e. `IsNumber` returns `True`), or the quotient of two numbers, or an infinity (i.e. `IsInfinity` returns `True`). Yacas will try to coerce the arguments passed to this comparison operator to a real value before making the comparison.

Examples:

```
In> 2 > 5;
Out> False;
In> Cos(1) > 5;
Out> False
```

See also: `IsNumber`, `IsInfinity`, `N`

<= — test for "less or equal"

(standard library)

Calling format:

```
e1 <= e2
```

Precedence: 90

Parameters:

`e1`, `e2` – expressions to be compared

Description:

The two expression are evaluated. If both results are numeric, they are compared. If the first expression is smaller than or equals the second one, the result is `True` and it is `False` otherwise. If either of the expression is not numeric, after evaluation, the expression is returned with evaluated arguments.

The word "numeric" in the previous paragraph has the following meaning. An expression is numeric if it is either a number (i.e. `IsNumber` returns `True`), or the quotient of two numbers, or an infinity (i.e. `IsInfinity` returns `True`). Yacas will try to coerce the arguments passed to this comparison operator to a real value before making the comparison.

Examples:

```
In> 2 <= 5;
Out> True;
In> Cos(1) <= 5;
Out> True
```

See also: `IsNumber`, `IsInfinity`, `N`

`>=` — test for “greater or equal”

(standard library)

Calling format:

```
e1 >= e2
```

Precedence: 90

Parameters:

`e1`, `e2` – expressions to be compared

Description:

The two expression are evaluated. If both results are numeric, they are compared. If the first expression is larger than or equals the second one, the result is `True` and it is `False` otherwise. If either of the expression is not numeric, after evaluation, the expression is returned with evaluated arguments.

The word “numeric” in the previous paragraph has the following meaning. An expression is numeric if it is either a number (i.e. `IsNumber` returns `True`), or the quotient of two numbers, or an infinity (i.e. `IsInfinity` returns `True`). Yacas will try to coerce the arguments passed to this comparison operator to a real value before making the comparison.

Examples:

```
In> 2 >= 5;
Out> False;
In> Cos(1) >= 5;
Out> False
```

See also: `IsNumber`, `IsInfinity`, `N`

`IsZero` — test whether argument is zero

(standard library)

Calling format:

```
IsZero(n)
```

Parameters:

`n` – number to test

Description:

`IsZero(n)` evaluates to `True` if “n” is zero. In case “n” is not a number, the function returns `False`.

Examples:

```
In> IsZero(3.25)
Out> False;
In> IsZero(0)
Out> True;
In> IsZero(x)
Out> False;
```

See also: `IsNumber`, `IsNotZero`

`IsRational` — test whether argument is a rational

(standard library)

Calling format:

```
IsRational(expr)
```

Parameters:

`expr` – expression to test

Description:

This commands tests whether the expression “expr” is a rational number, i.e. an integer or a fraction of integers.

Examples:

```
In> IsRational(5)
Out> False;
In> IsRational(2/7)
Out> True;
In> IsRational(0.5)
Out> False;
In> IsRational(a/b)
Out> False;
In> IsRational(x + 1/x)
Out> False;
```

See also: `Numer`, `Denom`

Chapter 4

Calculus and elementary functions

In this chapter, some facilities for doing calculus are described. These include functions implementing differentiation, integration, standard mathematical functions, and solving of equations.

Sin — trigonometric sine function

(standard library)

Calling format:

```
Sin(x)
```

Parameters:

x – argument to the function, in radians

Description:

This function represents the trigonometric function sine. Yacas leaves expressions alone even if x is a number, trying to keep the result as exact as possible. The floating point approximations of these functions can be forced by using the N function.

Yacas knows some trigonometric identities, so it can simplify to exact results even if N is not used. This is the case, for instance, when the argument is a multiple of $\pi/6$ or $\pi/4$.

These functions are threaded, meaning that if the argument x is a list, the function is applied to all entries in the list.

Examples:

```
In> Sin(1)
Out> Sin(1);
In> N(Sin(1),20)
Out> 0.84147098480789650665;
In> Sin(Pi/4)
Out> Sqrt(2)/2;
```

See also: Cos, Tan, ArcSin, ArcCos, ArcTan, N, Pi

Cos — trigonometric cosine function

(standard library)

Calling format:

```
Cos(x)
```

Parameters:

x – argument to the function, in radians

Description:

This function represents the trigonometric function cosine. Yacas leaves expressions alone even if x is a number, trying to keep the result as exact as possible. The floating point approximations of these functions can be forced by using the N function.

Yacas knows some trigonometric identities, so it can simplify to exact results even if N is not used. This is the case, for instance, when the argument is a multiple of $\pi/6$ or $\pi/4$.

These functions are threaded, meaning that if the argument x is a list, the function is applied to all entries in the list.

Examples:

```
In> Cos(1)
Out> Cos(1);
In> N(Cos(1),20)
Out> 0.5403023058681397174;
In> Cos(Pi/4)
Out> Sqrt(1/2);
```

See also: Sin, Tan, ArcSin, ArcCos, ArcTan, N, Pi

Tan — trigonometric tangent function

(standard library)

Calling format:

```
Tan(x)
```

Parameters:

x – argument to the function, in radians

Description:

This function represents the trigonometric function tangent. Yacas leaves expressions alone even if x is a number, trying to keep the result as exact as possible. The floating point approximations of these functions can be forced by using the N function.

Yacas knows some trigonometric identities, so it can simplify to exact results even if N is not used. This is the case, for instance, when the argument is a multiple of $\pi/6$ or $\pi/4$.

These functions are threaded, meaning that if the argument x is a list, the function is applied to all entries in the list.

Examples:

```
In> Tan(1)
Out> Tan(1);
In> N(Tan(1),20)
Out> 1.5574077246549022305;
In> Tan(Pi/4)
Out> 1;
```

See also: Sin, Cos, ArcSin, ArcCos, ArcTan, N, Pi

ArcSin — inverse trigonometric function arc-sine

(standard library)

Calling format:

```
ArcSin(x)
```

Parameters:

x – argument to the function

Description:

This function represents the inverse trigonometric function arc-sine. For instance, the value of $\arcsin x$ is a number y such that $\sin y$ equals x.

Note that the number y is not unique. For instance, $\sin 0$ and $\sin \pi$ both equal 0, so what should $\arcsin 0$ be? In Yacas, it is agreed that the value of $\arcsin x$ should be in the interval [-π/2,π/2].

Usually, Yacas leaves this function alone unless it is forced to do a numerical evaluation by the N function. If the argument is -1, 0, or 1 however, Yacas will simplify the expression. If the argument is complex, the expression will be rewritten using the Ln function.

This function is threaded, meaning that if the argument x is a list, the function is applied to all entries in the list.

Examples:

```
In> ArcSin(1)
Out> Pi/2;

In> ArcSin(1/3)
Out> ArcSin(1/3);
In> Sin(ArcSin(1/3))
Out> 1/3;

In> x:=N(ArcSin(0.75))
Out> 0.848062;
In> N(Sin(x))
Out> 0.7499999477;
```

See also: Sin, Cos, Tan, N, Pi, Ln, ArcCos, ArcTan

ArcCos — inverse trigonometric function arc-cosine

(standard library)

Calling format:

```
ArcCos(x)
```

Parameters:

x – argument to the function

Description:

This function represents the inverse trigonometric function arc-cosine. For instance, the value of $\arccos x$ is a number y such that $\cos y$ equals x.

Note that the number y is not unique. For instance, $\cos \frac{\pi}{2}$ and $\cos 3\frac{\pi}{2}$ both equal 0, so what should $\arccos 0$ be? In Yacas, it is agreed that the value of $\arccos x$ should be in the interval [0,π] .

Usually, Yacas leaves this function alone unless it is forced to do a numerical evaluation by the N function. If the argument is -1, 0, or 1 however, Yacas will simplify the expression. If the argument is complex, the expression will be rewritten using the Ln function.

This function is threaded, meaning that if the argument x is a list, the function is applied to all entries in the list.

Examples:

```
In> ArcCos(0)
Out> Pi/2

In> ArcCos(1/3)
Out> ArcCos(1/3)
In> Cos(ArcCos(1/3))
Out> 1/3

In> x:=N(ArcCos(0.75))
Out> 0.7227342478
In> N(Cos(x))
Out> 0.75
```

See also: Sin, Cos, Tan, N, Pi, Ln, ArcSin, ArcTan

ArcTan — inverse trigonometric function arc-tangent

(standard library)

Calling format:

```
ArcTan(x)
```

Parameters:

x – argument to the function

Description:

This function represents the inverse trigonometric function arc-tangent. For instance, the value of $\arctan x$ is a number y such that $\tan y$ equals x.

Note that the number y is not unique. For instance, $\tan 0$ and $\tan 2\pi$ both equal 0, so what should $\arctan 0$ be? In Yacas, it is agreed that the value of $\arctan x$ should be in the interval [-π/2,π/2].

Usually, Yacas leaves this function alone unless it is forced to do a numerical evaluation by the N function. Yacas will try to simplify as much as possible while keeping the result exact. If the argument is complex, the expression will be rewritten using the Ln function.

This function is threaded, meaning that if the argument x is a list, the function is applied to all entries in the list.

Examples:

```
In> ArcTan(1)
Out> Pi/4
```

```
In> ArcTan(1/3)
Out> ArcTan(1/3)
In> Tan(ArcTan(1/3))
Out> 1/3
```

```
In> x:=N(ArcTan(0.75))
Out> 0.643501108793285592213351264945231378078460693359375
In> N(Tan(x))
Out> 0.75
```

See also: Sin, Cos, Tan, N, Pi, Ln, ArcSin, ArcCos

Exp — exponential function

(standard library)

Calling format:

```
Exp(x)
```

Parameters:

x – argument to the function

Description:

This function calculates e raised to the power x, where e is the mathematic constant 2.71828... One can use Exp(1) to represent e.

This function is threaded, meaning that if the argument x is a list, the function is applied to all entries in the list.

Examples:

```
In> Exp(0)
Out> 1;
In> Exp(I*Pi)
Out> -1;
In> N(Exp(1))
Out> 2.7182818284;
```

See also: Ln, Sin, Cos, Tan, N

Ln — natural logarithm

(standard library)

Calling format:

```
Ln(x)
```

Parameters:

x – argument to the function

Description:

This function calculates the natural logarithm of "x". This is the inverse function of the exponential function, Exp, i.e. $\ln x = y$ implies that $\exp(y) = x$. For complex arguments, the imaginary part of the logarithm is in the interval (-π,π]. This is compatible with the branch cut of Arg.

This function is threaded, meaning that if the argument x is a list, the function is applied to all entries in the list.

Examples:

```
In> Ln(1)
Out> 0;
In> Ln(Exp(x))
Out> x;
In> D(x) Ln(x)
Out> 1/x;
```

See also: Exp, Arg

Sqrt — square root

(standard library)

Calling format:

```
Sqrt(x)
```

Parameters:

x – argument to the function

Description:

This function calculates the square root of "x". If the result is not rational, the call is returned unevaluated unless a numerical approximation is forced with the N function. This function can also handle negative and complex arguments.

This function is threaded, meaning that if the argument x is a list, the function is applied to all entries in the list.

Examples:

```
In> Sqrt(16)
Out> 4;
In> Sqrt(15)
Out> Sqrt(15);
In> N(Sqrt(15))
Out> 3.8729833462;
In> Sqrt(4/9)
Out> 2/3;
In> Sqrt(-1)
Out> Complex(0,1);
```

See also: Exp, ^, N

Abs — absolute value or modulus of complex number

(standard library)

Calling format:

```
Abs(x)
```

Parameters:

x – argument to the function

Description:

This function returns the absolute value (also called the modulus) of "x". If "x" is positive, the absolute value is "x" itself; if "x" is negative, the absolute value is "-x". For complex "x", the modulus is the "r" in the polar decomposition $x = r \exp(\imath \phi)$.

This function is connected to the `Sign` function by the identity "Abs(x) * Sign(x) = x" for real "x".

This function is threaded, meaning that if the argument `x` is a list, the function is applied to all entries in the list.

Examples:

```
In> Abs(2);
Out> 2;
In> Abs(-1/2);
Out> 1/2;
In> Abs(3+4*I);
Out> 5;
```

See also: `Sign`, `Arg`

`Sign` — sign of a number

(standard library)

Calling format:

```
Sign(x)
```

Parameters:

`x` – argument to the function

Description:

This function returns the sign of the real number x. It is "1" for positive numbers and "-1" for negative numbers. Somewhat arbitrarily, `Sign(0)` is defined to be 1.

This function is connected to the `Abs` function by the identity $|x| \operatorname{Sign}(x) = x$ for real x.

This function is threaded, meaning that if the argument `x` is a list, the function is applied to all entries in the list.

Examples:

```
In> Sign(2)
Out> 1;
In> Sign(-3)
Out> -1;
In> Sign(0)
Out> 1;
In> Sign(-3) * Abs(-3)
Out> -3;
```

See also: `Arg`, `Abs`

`D` — take derivative of expression with respect to variable

(standard library)

Calling format:

```
D(variable) expression
D(list) expression
D(variable,n) expression
```

Parameters:

`variable` – variable
`list` – a list of variables
`expression` – expression to take derivatives of
`n` – order of derivative

Description:

This function calculates the derivative of the expression `expr` with respect to the variable `var` and returns it. If the third calling format is used, the `n`-th derivative is determined. Yacas knows how the differentiate standard functions such as `Ln` and `Sin`.

The `D` operator is threaded in both `var` and `expr`. This means that if either of them is a list, the function is applied to each entry in the list. The results are collected in another list which is returned. If both `var` and `expr` are a list, their lengths should be equal. In this case, the first entry in the list `expr` is differentiated with respect to the first entry in the list `var`, the second entry in `expr` is differentiated with respect to the second entry in `var`, and so on.

The `D` operator returns the original function if $n = 0$, a common mathematical idiom that simplifies many formulae.

Examples:

```
In> D(x)Sin(x*y)
Out> y*Cos(x*y);
In> D({x,y,z})Sin(x*y)
Out> {y*Cos(x*y),x*Cos(x*y),0};
In> D(x,2)Sin(x*y)
Out> -Sin(x*y)*y^2;
In> D(x){Sin(x),Cos(x)}
Out> {Cos(x),-Sin(x)};
```

See also: `Integrate`, `Taylor`, `Diverge`, `Curl`

`Curl` — curl of a vector field

(standard library)

Calling format:

```
Curl(vector, basis)
```

Parameters:

`vector` – vector field to take the curl of
`basis` – list of variables forming the basis

Description:

This function takes the curl of the vector field "vector" with respect to the variables "basis". The curl is defined in the usual way,

```
Curl(f,x) = {
    D(x[2]) f[3] - D(x[3]) f[2],
    D(x[3]) f[1] - D(x[1]) f[3],
    D(x[1]) f[2] - D(x[2]) f[1]
}
```

Both "vector" and "basis" should be lists of length 3.

Example:

```
In> Curl({x*y,x*y,x*y},{x,y,z})
Out> {x,-y,y-x};
```

See also: `D`, `Diverge`

Diverge — divergence of a vector field

(standard library)

Calling format:

```
Diverge(vector, basis)
```

Parameters:

`vector` – vector field to calculate the divergence of
`basis` – list of variables forming the basis

Description:

This function calculates the divergence of the vector field "vector" with respect to the variables "basis". The divergence is defined as

```
Diverge(f,x) = D(x[1]) f[1] + ...
    + D(x[n]) f[n],
```

where `n` is the length of the lists "vector" and "basis". These lists should have equal length.

Example:

```
In> Diverge({x*y,x*y,x*y},{x,y,z})
Out> y+x;
```

See also: `D`, `Curl`

Integrate — integration

(standard library)

Calling format:

```
Integrate(var, x1, x2) expr
Integrate(var) expr
```

Parameters:

`var` – atom, variable to integrate over
`x1` – first point of definite integration
`x2` – second point of definite integration
`expr` – expression to integrate

Description:

This function integrates the expression `expr` with respect to the variable `var`. The first calling format is used to perform definite integration: the integration is carried out from var $= x_1$ to var $= x_2$". The second form is for indefinite integration.

Some simple integration rules have currently been implemented. Polynomials, some quotients of polynomials, trigonometric functions and their inverses, hyperbolic functions and their inverses, `Exp`, and `Ln`, and products of these functions with polynomials can be integrated.

Examples:

```
In> Integrate(x,a,b) Cos(x)
Out> Sin(b)-Sin(a);
In> Integrate(x) Cos(x)
Out> Sin(x);
```

See also: `D`, `UniqueConstant`

Limit — limit of an expression

(standard library)

Calling format:

```
Limit(var, val) expr
Limit(var, val, dir) expr
```

Parameters:

`var` – a variable
`val` – a number
`dir` – a direction (`Left` or `Right`)
`expr` – an expression

Description:

This command tries to determine the value that the expression "expr" converges to when the variable "var" approaches "val". One may use `Infinity` or `-Infinity` for "val". The result of `Limit` may be one of the symbols `Undefined` (meaning that the limit does not exist), `Infinity`, or `-Infinity`.

The second calling sequence is used for unidirectional limits. If one gives "dir" the value `Left`, the limit is taken as "var" approaches "val" from the positive infinity; and `Right` will take the limit from the negative infinity.

Examples:

```
In> Limit(x,0) Sin(x)/x
Out> 1;
In> Limit(x,0) (Sin(x)-Tan(x))/(x^3)
Out> -1/2;
In> Limit(x,0) 1/x
Out> Undefined;
In> Limit(x,0,Left) 1/x
Out> -Infinity;
In> Limit(x,0,Right) 1/x
Out> Infinity;
```

Chapter 5

Random numbers

Random, RandomSeed — (pseudo-) random number generator

(standard library)

Calling format:

```
Random()
RandomSeed(init)
```

*PARAMS `init` – positive integer, initial random seed

Description:

The function `Random` returns a random number, uniformly distributed in the interval between 0 and 1. The same sequence of random numbers is generated in every Yacas session.

The random number generator can be initialized by calling `RandomSeed` with an integer value. Each seed value will result in the same sequence of pseudo-random numbers.

See also: `RandomInteger`, `RandomPoly`, `Rng`

RngCreate — manipulate random number generators as objects

RngSeed — manipulate random number generators as objects

Rng — manipulate random number generators as objects

(standard library)

Calling format:

```
RngCreate()
RngCreate(init)
RngCreate(option==value,...)
RngSeed(r, init)
Rng(r)
```

Parameters:

`init` – integer, initial seed value
`option` – atom, option name
`value` – atom, option value
`r` – a list, RNG object

Description:

These commands are an object-oriented interface to (pseudo-)random number generators (RNGs).

`RngCreate` returns a list which is a well-formed RNG object. Its value should be saved in a variable and used to call `Rng` and `RngSeed`.

`Rng(r)` returns a floating-point random number between 0 and 1 and updates the RNG object `r`. (Currently, the Gaussian option makes a RNG return a *complex* random number instead of a real random number.)

`RngSeed(r,init)` re-initializes the RNG object `r` with the seed value `init`. The seed value should be a positive integer.

The `RngCreate` function accepts several options as arguments. Currently the following options are available:

- `seed` – specify initial seed value, must be a positive integer
- `dist` – specify the distribution of the random number; currently `flat` and `gauss` are implemented, and the default is the flat (uniform) distribution
- `engine` – specify the RNG engine; currently `default` and `advanced` are available ("advanced" is slower but has much longer period)

If the initial seed is not specified, the value of 76544321 will be used.

The `gauss` option will create a RNG object that generates pairs of Gaussian distributed random numbers as a complex random number. The real and the imaginary parts of this number are independent random numbers taken from a Gaussian (i.e. "normal") distribution with unit variance.

For the Gaussian distribution, the Box-Muller transform method is used. A good description of this method, along with the proof that the method generates normally distributed random numbers, can be found in Knuth, "The Art of Computer Programming", Volume 2 (Seminumerical algorithms, third edition), section 3.4.1

Note that unlike the global `Random` function, the RNG objects created with `RngCreate` are independent RNGs and do not affect each other. They generate independent streams of pseudo-random numbers. However, the `Random` function is slightly faster.

Examples:

```
In> r1:=RngCreate(seed=1,dist=gauss)
Out> {"GaussianRNGDist","RNGEngine'LCG'2",{1}}
In> Rng(r1)
Out> Complex(-1.6668466417,0.228904004);
In> Rng(r1);
Out> Complex(0.0279296109,-0.5382405341);
```

The second RNG gives a uniform distribution (default option) but uses a more complicated algorithm:

```
In> [r2:=RngCreate(engine=advanced);Rng(r2);]
Out> 0.3653615377;
```

The generator `r1` can be re-initialized with seed 1 again to obtain the same sequence:

```
In> RngSeed(r1, 1)
Out> True;
In> Rng(r1)
Out> Complex(-1.6668466417,0.228904004);
```

See also: `Random`

`RandomIntegerMatrix` — generate a matrix of random integers

(standard library)

Calling format:

```
RandomIntegerMatrix(rows,cols,from,to)
```

Parameters:

`rows` – number of rows in matrix
`cols` – number of cols in matrix
`from` – lower bound
`to` – upper bound

Description:

This function generates a `rows x cols` matrix of random integers. All entries lie between “from” and “to”, including the boundaries, and are uniformly distributed in this interval.

Examples:

```
In> PrettyForm( RandomIntegerMatrix(5,5,-2^10,2^10) )
/                                                  \
| ( -506 ) ( 749 )  ( -574 ) ( -674 ) ( -106 )  |
|                                               |
| ( 301 )  ( 151 )  ( -326 ) ( -56 )  ( -277 )  |
|                                               |
| ( 777 )  ( -761 ) ( -161 ) ( -918 ) ( -417 )  |
|                                               |
| ( -518 ) ( 127 )  ( 136 )  ( 797 )  ( -406 )  |
|                                               |
| ( 679 )  ( 854 )  ( -78 )  ( 503 )  ( 772 )   |
\                                                  /
```

See also: `RandomIntegerVector`, `RandomPoly`

`RandomIntegerVector` — generate a vector of random integers

(standard library)

Calling format:

```
RandomIntegerVector(nr, from, to)
```

Parameters:

`nr` – number of integers to generate
`from` – lower bound
`to` – upper bound

Description:

This function generates a list with “nr” random integers. All entries lie between “from” and “to”, including the boundaries, and are uniformly distributed in this interval.

Examples:

```
In> RandomIntegerVector(4,-3,3)
Out> {0,3,2,-2};
```

See also: `Random`, `RandomPoly`

`RandomPoly` — construct a random polynomial

(standard library)

Calling format:

```
RandomPoly(var,deg,coefmin,coefmax)
```

Parameters:

`var` – free variable for resulting univariate polynomial
`deg` – degree of resulting univariate polynomial
`coefmin` – minimum value for coefficients
`coefmax` – maximum value for coefficients

Description:

RandomPoly generates a random polynomial in variable “var”, of degree “deg”, with integer coefficients ranging from “coefmin” to “coefmax” (inclusive). The coefficients are uniformly distributed in this interval, and are independent of each other.

Examples:

```
In> RandomPoly(x,3,-10,10)
Out> 3*x^3+10*x^2-4*x-6;
In> RandomPoly(x,3,-10,10)
Out> -2*x^3-8*x^2+8;
```

See also: `Random`, `RandomIntegerVector`

Chapter 6

Series

Add — find sum of a list of values

(standard library)

Calling format:

```
Add(val1, val2, ...)
Add({list})
```

Parameters:

`val1`, `val2` – expressions
`{list}` – list of expressions to add

Description:

This function adds all its arguments and returns their sum. It accepts any number of arguments. The arguments can be also passed as a list.

Examples:

```
In> Add(1,4,9);
Out> 14;
In> Add(1 .. 10);
Out> 55;
```

Sum — find sum of a sequence

(standard library)

Calling format:

```
Sum(var, from, to, body)
```

Parameters:

`var` – variable to iterate over
`from` – integer value to iterate from
`to` – integer value to iterate up to
`body` – expression to evaluate for each iteration

Description:

The command finds the sum of the sequence generated by an iterative formula. The expression "body" is evaluated while the variable "var" ranges over all integers from "from" up to "to", and the sum of all the results is returned. Obviously, "to" should be greater than or equal to "from".

Warning: `Sum` does not evaluate its arguments `var` and `body` until the actual loop is run.

Examples:

```
In> Sum(i, 1, 3, i^2);
Out> 14;
```

See also: `Factorize`

Factorize — product of a list of values

(standard library)

Calling format:

```
Factorize(list)
Factorize(var, from, to, body)
```

Parameters:

`list` – list of values to multiply
`var` – variable to iterate over
`from` – integer value to iterate from
`to` – integer value to iterate up to
`body` – expression to evaluate for each iteration

Description:

The first form of the `Factorize` command simply multiplies all the entries in "list" and returns their product.

If the second calling sequence is used, the expression "body" is evaluated while the variable "var" ranges over all integers from "from" up to "to", and the product of all the results is returned. Obviously, "to" should be greater than or equal to "from".

Examples:

```
In> Factorize({1,2,3,4});
Out> 24;
In> Factorize(i, 1, 4, i);
Out> 24;
```

See also: `Sum`, `Apply`

Taylor — univariate Taylor series expansion

(standard library)

Calling format:

```
Taylor(var, at, order) expr
```

Parameters:

`var` – variable
`at` – point to get Taylor series around
`order` – order of approximation
`expr` – expression to get Taylor series for

Description:

This function returns the Taylor series expansion of the expression "expr" with respect to the variable "var" around "at" up to order "order". This is a polynomial which agrees with "expr" at the point "var = at", and furthermore the first "order" derivatives of the polynomial at this point agree with "expr". Taylor expansions around removable singularities are correctly handled by taking the limit as "var" approaches "at".

Examples:

```
In> PrettyForm(Taylor(x,0,9) Sin(x))

     3     5       7          9
    x     x       x          x
x - -- + --- - ---- + ------
    6     120    5040    362880

Out> True;
```

See also: D, InverseTaylor, ReversePoly, BigOh

InverseTaylor — Taylor expansion of inverse

(standard library)

Calling format:

```
InverseTaylor(var, at, order) expr
```

Parameters:

var – variable
at – point to get inverse Taylor series around
order – order of approximation
expr – expression to get inverse Taylor series for

Description:

This function builds the Taylor series expansion of the inverse of the expression "expr" with respect to the variable "var" around "at" up to order "order". It uses the function ReversePoly to perform the task.

Examples:

```
In> PrettyPrinter'Set("PrettyForm")

True

In> exp1 := Taylor(x,0,7) Sin(x)

     3     5       7
    x     x       x
x - -- + --- - ----
    6     120    5040

In> exp2 := InverseTaylor(x,0,7) ArcSin(x)

 5       7      3
x       x      x
--- - ---- - -- + x
120    5040   6

In> Simplify(exp1-exp2)

0
```

See also: ReversePoly, Taylor, BigOh

ReversePoly — solve $h(f(x)) = g(x) + O(x^n)$ for h

(standard library)

Calling format:

```
ReversePoly(f, g, var, newvar, degree)
```

Parameters:

f, g – functions of "var"
var – a variable
newvar – a new variable to express the result in
degree – the degree of the required solution

Description:

This function returns a polynomial in "newvar", say "h(newvar)", with the property that "h(f(var))" equals "g(var)" up to order "degree". The degree of the result will be at most "degree-1". The only requirement is that the first derivative of "f" should not be zero.

This function is used to determine the Taylor series expansion of the inverse of a function "f": if we take "g(var)=var", then "h(f(var))=var" (up to order "degree"), so "h" will be the inverse of "f".

Examples:

```
In> f(x):=Eval(Expand((1+x)^4))
Out> True;
In> g(x) := x^2
Out> True;
In> h(y):=Eval(ReversePoly(f(x),g(x),x,y,8))
Out> True;
In> BigOh(h(f(x)),x,8)
Out> x^2;
In> h(x)
Out> (-2695*(x-1)^7)/131072+(791*(x-1)^6)
/32768 +(-119*(x-1)^5)/4096+(37*(x-1)^4)
/1024+(-3*(x-1)^3)/64+(x-1)^2/16;
```

See also: InverseTaylor, Taylor, BigOh

BigOh — drop all terms of a certain order in a polynomial

(standard library)

Calling format:

```
BigOh(poly, var, degree)
```

Parameters:

poly – a univariate polynomial
var – a free variable
degree – positive integer

Description:

This function drops all terms of order "degree" or higher in "poly", which is a polynomial in the variable "var".

Examples:

```
In> BigOh(1+x+x^2+x^3,x,2)
Out> x+1;
```

See also: Taylor, InverseTaylor

`LagrangeInterpolant` — polynomial interpolation

(standard library)

Calling format:

```
LagrangeInterpolant(xlist, ylist, var)
```

Parameters:

`xlist` – list of argument values
`ylist` – list of function values
`var` – free variable for resulting polynomial

Description:

This function returns a polynomial in the variable "var" which interpolates the points "(xlist, ylist)". Specifically, the value of the resulting polynomial at "xlist[1]" is "ylist[1]", the value at "xlist[2]" is "ylist[2]", etc. The degree of the polynomial is not greater than the length of "xlist".

The lists "xlist" and "ylist" should be of equal length. Furthermore, the entries of "xlist" should be all distinct to ensure that there is one and only one solution.

This routine uses the Lagrange interpolant formula to build up the polynomial.

Examples:

```
In> f := LagrangeInterpolant({0,1,2}, \
  {0,1,1}, x);
Out> (x*(x-1))/2-x*(x-2);
In> Eval(Subst(x,0) f);
Out> 0;
In> Eval(Subst(x,1) f);
Out> 1;
In> Eval(Subst(x,2) f);
Out> 1;

In> PrettyPrinter'Set("PrettyForm");

True

In> LagrangeInterpolant({x1,x2,x3}, {y1,y2,y3}, x)

y1 * ( x - x2 ) * ( x - x3 )
----------------------------
 ( x1 - x2 ) * ( x1 - x3 )

  y2 * ( x - x1 ) * ( x - x3 )
+ ----------------------------
   ( x2 - x1 ) * ( x2 - x3 )

  y3 * ( x - x1 ) * ( x - x2 )
+ ----------------------------
   ( x3 - x1 ) * ( x3 - x2 )
```

See also: `Subst`

Chapter 7

Combinatorics

`!` — factorial

`!!` — factorial and related functions

`***` — factorial and related functions

`Subfactorial` — factorial and related functions

(standard library)

Calling format:

```
n!
n!!
a *** b
Subfactorial(m)
```

Parameters:

`m` – integer `n` – integer, half-integer, or list `a`, `b` – numbers

Description:

The factorial function `n!` calculates the factorial of integer or half-integer numbers. For nonnegative integers, $n! \equiv n(n-1)(n-2)...\cdot 1$. The factorial of half-integers is defined via Euler's Gamma function, $z! \equiv \Gamma(z+1)$. If $n=0$ the function returns 1.

The "double factorial" function `n!!` calculates $n(n-2)(n-4)...$. This product terminates either with 1 or with 2 depending on whether n is odd or even. If $n=0$ the function returns 1.

The "partial factorial" function `a *** b` calculates the product $a(a+1)...$ which is terminated at the least integer not greater than b. The arguments a and b do not have to be integers; for integer arguments, `a *** b` $= \frac{b!}{(a-1)!}$. This function is sometimes a lot faster than evaluating the two factorials, especially if a and b are close together. If $a>b$ the function returns 1.

The `Subfactorial` function can be interpreted as the number of permutations of `m` objects in which no object appears in its natural place, also called "derangements."

The factorial functions are threaded, meaning that if the argument `n` is a list, the function will be applied to each element of the list.

Note: For reasons of Yacas syntax, the factorial sign `!` cannot precede other non-letter symbols such as `+` or `*`. Therefore, you should enter a space after `!` in expressions such as `x! +1`.

The factorial functions terminate and print an error message if the arguments are too large (currently the limit is $n<65535$) because exact factorials of such large numbers are computationally expensive and most probably not useful. One can call `Internal'LnGammaNum()` to evaluate logarithms of such factorials to desired precision.

Examples:

```
In> 5!
Out> 120;
In> 1 * 2 * 3 * 4 * 5
Out> 120;
In> (1/2)!
Out> Sqrt(Pi)/2;
In> 7!!;
Out> 105;
In> 1/3 *** 10;
Out> 17041024000/59049;
In> Subfactorial(10)
Out> 1334961;
```

See also: `Bin`, `Factorize`, `Gamma`, `!!`, `***`, `Subfactorial`

`Bin` — binomial coefficients

(standard library)

Calling format:

```
Bin(n, m)
```

Parameters:

`n`, `m` – integers

Description:

This function calculates the binomial coefficient "n" above "m", which equals

$$\frac{n!}{m!\,(n-m)!}$$

This is equal to the number of ways to choose "m" objects out of a total of "n" objects if order is not taken into account. The binomial coefficient is defined to be zero if "m" is negative or greater than "n"; `Bin(0,0)`=1.

Examples:

```
In> Bin(10, 4)
Out> 210;
In> 10! / (4! * 6!)
Out> 210;
```

See also: `!`, `Eulerian`

`Eulerian` — Eulerian numbers

(standard library)

Calling format:

```
Eulerian(n,m)
```

Parameters:

`n`, `m` — integers

Description:

The Eulerian numbers can be viewed as a generalization of the binomial coefficients, and are given explicitly by

$$\sum_{j=0}^{k+1} (-1)^j \binom{n+1}{j} (k-j+1)^n$$

.

Examples:

```
In> Eulerian(6,2)
Out> 302;
In> Eulerian(10,9)
Out> 1;
```

See also: `Bin`

`LeviCivita` — totally anti-symmetric Levi-Civita symbol

(standard library)

Calling format:

```
LeviCivita(list)
```

Parameters:

`list` – a list of integers 1 .. n in some order

Description:

`LeviCivita` implements the Levi-Civita symbol. This is generally useful for tensor calculus. `list` should be a list of integers, and this function returns 1 if the integers are in successive order, eg. `LeviCivita( {1,2,3,...} )` would return 1. Swapping two elements of this list would return -1. So, `LeviCivita( {2,1,3} )` would evaluate to -1.

Examples:

```
In> LeviCivita({1,2,3})
Out> 1;
In> LeviCivita({2,1,3})
Out> -1;
In> LeviCivita({2,2,3})
Out> 0;
```

See also: `Permutations`

`Permutations` — get all permutations of a list

(standard library)

Calling format:

```
Permutations(list)
```

Parameters:

`list` – a list of elements

Description:

Permutations returns a list with all the permutations of the original list.

Examples:

```
In> Permutations({a,b,c})
Out> {{a,b,c},{a,c,b},{c,a,b},{b,a,c},
{b,c,a},{c,b,a}};
```

See also: `LeviCivita`

Chapter 8

Special functions

In this chapter, special and transcendental mathematical functions are described.

Gamma — Euler's Gamma function

(standard library)

Calling format:

```
Gamma(x)
```

Parameters:

x – expression
number – expression that can be evaluated to a number

Description:

Gamma(x) is an interface to Euler's Gamma function $\Gamma(x)$. It returns exact values on integer and half-integer arguments. N(Gamma(x) takes a numeric parameter and always returns a floating-point number in the current precision.

Note that Euler's constant $\gamma \approx 0.57722$ is the lowercase gamma in Yacas.

Examples:

```
In> Gamma(1.3)
Out> Gamma(1.3);
In> N(Gamma(1.3),30)
Out> 0.897470696306277188493754954771;
In> Gamma(1.5)
Out> Sqrt(Pi)/2;
In> N(Gamma(1.5),30);
Out> 0.88622692545275801364908374167;
```

See also: !, N, gamma

Zeta — Riemann's Zeta function

(standard library)

Calling format:

```
Zeta(x)
```

Parameters:

x – expression
number – expression that can be evaluated to a number

Description:

Zeta(x) is an interface to Riemann's Zeta function $\zeta(s)$. It returns exact values on integer and half-integer arguments. N(Zeta(x) takes a numeric parameter and always returns a floating-point number in the current precision.

Examples:

```
In> Precision(30)
Out> True;
In> Zeta(1)
Out> Infinity;
In> Zeta(1.3)
Out> Zeta(1.3);
In> N(Zeta(1.3))
Out> 3.93194921180954422697490751058798;
In> Zeta(2)
Out> Pi^2/6;
In> N(Zeta(2));
Out> 1.644934066848226436472415166646025;
```

See also: !, N

Bernoulli — Bernoulli numbers and polynomials

(standard library)

Calling format:

```
Bernoulli(index)
Bernoulli(index, x)
```

Parameters:

x – expression that will be the variable in the polynomial
index – expression that can be evaluated to an integer

Description:

Bernoulli(n) evaluates the n-th Bernoulli number. Bernoulli(n, x) returns the n-th Bernoulli polynomial in the variable x. The polynomial is returned in the Horner form.

Example:

```
In> Bernoulli(20);
Out> -174611/330;
In> Bernoulli(4, x);
Out> ((x-2)*x+1)*x^2-1/30;
```

See also: Gamma, Zeta

Euler — Euler numbers and polynomials

(standard library)

Calling format:

```
Euler(index)
Euler(index,x)
```

Parameters:

x – expression that will be the variable in the polynomial
index – expression that can be evaluated to an integer

Description:

Euler(n) evaluates the n-th Euler number. Euler(n,x) returns the n-th Euler polynomial in the variable x.

Examples:

```
In> Euler(6)
Out> -61;
In> A:=Euler(5,x)
Out> (x-1/2)^5+(-10*(x-1/2)^3)/4+(25*(x-1/2))/16;
In> Simplify(A)
Out> (2*x^5-5*x^4+5*x^2-1)/2;
```

See also: Bin

LambertW — Lambert's W function

(standard library)

Calling format:

```
LambertW(x)
```

Parameters:

x – expression, argument of the function

Description:

Lambert's W function is (a multiple-valued, complex function) defined for any (complex) z by

$$W(z)\exp(W(z)) = z.$$

This function is sometimes useful to represent solutions of transcendental equations. For example, the equation $\ln x = 3x$ can be "solved" by writing $x = -3W\left(-\frac{1}{3}\right)$. It is also possible to take a derivative or integrate this function "explicitly".

For real arguments x, $W(x)$ is real if $x \geq -\exp(-1)$.

To compute the numeric value of the principal branch of Lambert's W function for real arguments $x \geq -\exp(-1)$ to current precision, one can call N(LambertW(x)) (where the function N tries to approximate its argument with a real value).

Examples:

```
In> LambertW(0)
Out> 0;
In> N(LambertW(-0.24/Sqrt(3*Pi)))
Out> -0.0851224014;
```

See also: Exp

Chapter 9

Complex numbers

Yacas understands the concept of a complex number, and has a few functions that allow manipulation of complex numbers.

Complex — construct a complex number

(standard library)

Calling format:

```
Complex(r, c)
```

Parameters:

r – real part
c – imaginary part

Description:

This function represents the complex number "r + I*c", where "I" is the imaginary unit. It is the standard representation used in Yacas to represent complex numbers. Both "r" and "c" are supposed to be real.

Note that, at the moment, many functions in Yacas assume that all numbers are real unless it is obvious that it is a complex number. Hence `Im(Sqrt(x))` evaluates to `0` which is only true for nonnegative "x".

Examples:

```
In> I
Out> Complex(0,1);
In> 3+4*I
Out> Complex(3,4);
In> Complex(-2,0)
Out> -2;
```

See also: `Re`, `Im`, `I`, `Abs`, `Arg`

Re — real part of a complex number

(standard library)

Calling format:

```
Re(x)
```

Parameters:

x – argument to the function

Description:

This function returns the real part of the complex number "x".

Examples:

```
In> Re(5)
Out> 5;
In> Re(I)
Out> 0;
In> Re(Complex(3,4))
Out> 3;
```

See also: `Complex`, `Im`

Im — imaginary part of a complex number

(standard library)

Calling format:

```
Im(x)
```

Parameters:

x – argument to the function

Description:

This function returns the imaginary part of the complex number "x".

Examples:

```
In> Im(5)
Out> 0;
In> Im(I)
Out> 1;
In> Im(Complex(3,4))
Out> 4;
```

See also: `Complex`, `Re`

I — imaginary unit

(standard library)

Calling format:

```
I
```

Description:

This symbol represents the imaginary unit, which equals the square root of -1. It evaluates to `Complex(0,1)`.

Examples:

```
In> I
Out> Complex(0,1);
In> I = Sqrt(-1)
Out> True;
```

See also: `Complex`

`Conjugate` — complex conjugate

(standard library)

Calling format:

```
Conjugate(x)
```

Parameters:

`x` – argument to the function

Description:

This function returns the complex conjugate of "x". The complex conjugate of $a + \imath b$ is $a - \imath b$. This function assumes that all unbound variables are real.

Examples:

```
In> Conjugate(2)
Out> 2;
In> Conjugate(Complex(a,b))
Out> Complex(a,-b);
```

See also: `Complex`, `Re`, `Im`

`Arg` — argument of a complex number

(standard library)

Calling format:

```
Arg(x)
```

Parameters:

`x` – argument to the function

Description:

This function returns the argument of "x". The argument is the angle with the positive real axis in the Argand diagram, or the angle "phi" in the polar representation $r \exp(\imath \phi)$ of "x". The result is in the range $(-\pi, \pi]$, that is, excluding $-\pi$ but including π. The argument of 0 is `Undefined`.

Examples:

```
In> Arg(2)
Out> 0;
In> Arg(-1)
Out> Pi;
In> Arg(1+I)
Out> Pi/4;
```

See also: `Abs`, `Sign`

Chapter 10

Transforms

In this chapter, some facilities for various transforms are described.

`LaplaceTransform` — Laplace Transform

(standard library)

Calling format:

```
LaplaceTransform(t,s,func)
```

Parameters:

`t` – independent variable that is being transformed
`s` – independent variable that is being transformed into
`f` – function

Description:

This function attempts to take the function `f(t)` and find the Laplace transform of it,`F(s)`, which is defined as `Integrate(t,0,Infinity) Exp(-s*t)*f`. This is also sometimes referred to the "unilateral" Laplace tranform. `LaplaceTransform` can transform most elementary functions that do not require a convolution integral, as well as any polynomial times an elementary function. If a transform cannot be found then `LaplaceTransform` will return unevaluated. This can happen for function which are not of "exponential order", which means that they grow faster than exponential functions.

Examples:

```
In> LaplaceTransform(t,s,2*t^5+ t^2/2 )
Out> 240/s^6+2/(2*s^3);
In> LaplaceTransform(t,s,t*Sin(2*t)*Exp(-3*t) )
Out> (2*(s+3))/(2*(2*(((s+3)/2)^2+1))^2);
In> LaplaceTransform(t,s, BesselJ(3,2*t) )
Out> (Sqrt((s/2)^2+1)-s/2)^3/(2*Sqrt((s/2)^2+1));
In> LaplaceTransform(t,s,Exp(t^2)); // not of exponential order
Out> LaplaceTransform(t,s,Exp(t^2));
In> LaplaceTransform(p,q,Ln(p))
Out> -(gamma+Ln(q))/q;
```

Chapter 11

Simplification of expressions

Simplification of expression is a big and non-trivial subject. Simplification implies that there is a preferred form. In practice the preferred form depends on the calculation at hand. This chapter describes the functions offered that allow simplification of expressions.

Simplify — try to simplify an expression

(standard library)

Calling format:

```
Simplify(expr)
```

Parameters:

`expr` – expression to simplify

Description:

This function tries to simplify the expression `expr` as much as possible. It does this by grouping powers within terms, and then grouping similar terms.

Examples:

```
In> a*b*a^2/b-a^3
Out> (b*a^3)/b-a^3;
In> Simplify(a*b*a^2/b-a^3)
Out> 0;
```

See also: `TrigSimpCombine`, `RadSimp`

RadSimp — simplify expression with nested radicals

(standard library)

Calling format:

```
RadSimp(expr)
```

Parameters:

`expr` – an expression containing nested radicals

Description:

This function tries to write the expression "expr" as a sum of roots of integers: $\sqrt{e_1} + \sqrt{e_2} + ...$, where e_1, e_2 and so on are natural numbers. The expression "expr" may not contain free variables.

It does this by trying all possible combinations for e_1, e_2, ... Every possibility is numerically evaluated using `N` and compared with the numerical evaluation of "expr". If the approximations are equal (up to a certain margin), this possibility is returned. Otherwise, the expression is returned unevaluated.

Note that due to the use of numerical approximations, there is a small chance that the expression returned by `RadSimp` is close but not equal to `expr`. The last example underneath illustrates this problem. Furthermore, if the numerical value of `expr` is large, the number of possibilities becomes exorbitantly big so the evaluation may take very long.

Examples:

```
In> RadSimp(Sqrt(9+4*Sqrt(2)))
Out> Sqrt(8)+1;
In> RadSimp(Sqrt(5+2*Sqrt(6)) \
  +Sqrt(5-2*Sqrt(6)))
Out> Sqrt(12);
In> RadSimp(Sqrt(14+3*Sqrt(3+2
*Sqrt(5-12*Sqrt(3-2*Sqrt(2))))))
Out> Sqrt(2)+3;
```

But this command may yield incorrect results:

```
In> RadSimp(Sqrt(1+10^(-6)))
Out> 1;
```

See also: `Simplify`, `N`

FactorialSimplify — Simplify hypergeometric expressions containing factorials

(standard library)

Calling format:

```
FactorialSimplify(expression)
```

Parameters:

`expression` – expression to simplify

Description:

`FactorialSimplify` takes an expression that may contain factorials, and tries to simplify it. An expression like $\frac{(n+1)!}{n!}$ would simplify to $n+1$.

The following steps are taken to simplify:

1. binomials are expanded into factorials
2. the expression is flattened as much as possible, to reduce it to a sum of simple rational terms
3. expressions like $\frac{p^n}{p^m}$ are reduced to p^{n-m} if $n-m$ is an integer
4. expressions like $\frac{n!}{m!}$ are simplified if $n-m$ is an integer

The function `Simplify` is used to determine if the relevant expressions $n-m$ are integers.

Example:

```
In> FactorialSimplify( (n-k+1)! / (n-k)! )
Out> n+1-k
In> FactorialSimplify(n! / Bin(n,k))
Out> k! *(n-k)!
In> FactorialSimplify(2^(n+1)/2^n)
Out> 2
```

See also: `Simplify`, `!`, `Bin`

`LnExpand` — expand a logarithmic expression using standard logarithm rules

(standard library)

Calling format:

```
LnExpand(expr)
```

Parameters:

`expr` – the logarithm of an expression

Description:

`LnExpand` takes an expression of the form $\ln \mathrm{expr}$, and applies logarithm rules to expand this into multiple `Ln` expressions where possible. An expression like $\ln ab^n$ would be expanded to $\ln a + n \ln b$.

If the logarithm of an integer is discovered, it is factorised using `Factors` and expanded as though `LnExpand` had been given the factorised form. So $\ln 18$ goes to $\ln x + 2 \ln 3$.

Example:

```
In> LnExpand(Ln(a*b^n))
Out> Ln(a)+Ln(b)*n
In> LnExpand(Ln(a^m/b^n))
Out> Ln(a)*m-Ln(b)*n
In> LnExpand(Ln(60))
Out> 2*Ln(2)+Ln(3)+Ln(5)
In> LnExpand(Ln(60/25))
Out> 2*Ln(2)+Ln(3)-Ln(5)
```

See also: `Ln`, `LnCombine`, `Factors`

`LnCombine` — combine logarithmic expressions using standard logarithm rules

(standard library)

Calling format:

```
LnCombine(expr)
```

Parameters:

`expr` – an expression possibly containing multiple `Ln` terms to be combined

Description:

`LnCombine` finds `Ln` terms in the expression it is given, and combines them using logarithm rules. It is intended to be the exact converse of `LnExpand`.

Example:

```
In> LnCombine(Ln(a)+Ln(b)*n)
Out> Ln(a*b^n)
In> LnCombine(2*Ln(2)+Ln(3)-Ln(5))
Out> Ln(12/5)
```

See also: `Ln`, `LnExpand`

`TrigSimpCombine` — combine products of trigonometric functions

(standard library)

Calling format:

```
TrigSimpCombine(expr)
```

Parameters:

`expr` – expression to simplify

Description:

This function applies the product rules of trigonometry, e.g. $\cos u \sin v = \frac{1}{2} (\sin (v-u) + \sin (v+u))$. As a result, all products of the trigonometric functions `Cos` and `Sin` disappear. The function also tries to simplify the resulting expression as much as possible by combining all similar terms.

This function is used in for instance `Integrate`, to bring down the expression into a simpler form that hopefully can be integrated easily.

Examples:

```
In> PrettyPrinter'Set("PrettyForm");

True

In> TrigSimpCombine(Cos(a)^2+Sin(a)^2)

1

In> TrigSimpCombine(Cos(a)^2-Sin(a)^2)
```

```
Cos( -2 * a )

Out>
In> TrigSimpCombine(Cos(a)^2*Sin(b))

Sin( b )   Sin( -2 * a + b )
-------- + -----------------
   2               4

    Sin( -2 * a - b )
  - -----------------
            4
```

See also: Simplify, Integrate, Expand, Sin, Cos, Tan

Chapter 12

Symbolic solvers

By solving one tries to find a mathematical object that meets certain criteria. This chapter documents the functions that are available to help find solutions to specific types of problems.

Solve — solve an equation

(standard library)

Calling format:

```
Solve(eq, var)
```

Parameters:

`eq` – equation to solve
`var` – variable to solve for

Description:

This command tries to solve an equation. If `eq` does not contain the `==` operator, it is assumed that the user wants to solve $eq == 0$. The result is a list of equations of the form `var == value`, each representing a solution of the given equation. The `Where` operator can be used to substitute this solution in another expression. If the given equation `eq` does not have any solutions, or if `Solve` is unable to find any, then an empty list is returned.

The current implementation is far from perfect. In particular, the user should keep the following points in mind:

- `Solve` cannot solve all equations. If it is given a equation it can not solve, it raises an error via `Check`. Unfortunately, this is not displayed by the inline pretty-printer; call `PrettyPrinter'Set` to change this. If an equation cannot be solved analytically, you may want to call `Newton` to get a numerical solution.
- Systems of equations are not handled yet. For linear systems, `MatrixSolve` can be used. The old version of `Solve`, with the name `OldSolve` might be able to solve nonlinear systems of equations.
- The periodicity of the trigonometric functions `Sin`, `Cos`, and `Tan` is not taken into account. The same goes for the (imaginary) periodicity of `Exp`. This causes `Solve` to miss solutions.
- It is assumed that all denominators are nonzero. Hence, a solution reported by `Solve` may in fact fail to be a solution because a denominator vanishes.
- In general, it is wise not to have blind trust in the results returned by `Solve`. A good strategy is to substitute the solutions back in the equation.

Examples:

First a simple example, where everything works as it should. The quadratic equation $x^2 + x = 0$ is solved. Then the result is checked by substituting it back in the quadratic.

```
In> quadratic := x^2+x;
Out> x^2+x;
In> Solve(quadratic, x);
Out> {x==0,x==(-1)};
In> quadratic Where %;
Out> {0,0};
```

If one tries to solve the equation $\exp(x) = \sin x$, one finds that `Solve` can not do this.

```
In> PrettyPrinter'Set("DefaultPrint");
Out> True;
In> Solve(Exp(x) == Sin(x), x);
Error: Solve'Fails: cannot solve equation Exp(x)-Sin(x) fo
Out> {};
```

The equation $\cos x = \frac{1}{2}$ has an infinite number of solutions, namely $x = \left(2k + \frac{1}{3}\right)\pi$ and $x = \left(2k - \frac{1}{3}\right)\pi$ for any integer k. However, `Solve` only reports the solutions with $k = 0$.

```
In> Solve(Cos(x) == 1/2, x);
Out> {x==Pi/3,x== -Pi/3};
```

For the equation $\frac{x}{\sin x} = 0$, a spurious solution at $x = 0$ is returned. However, the fraction is undefined at that point.

```
In> Solve(x / Sin(x) == 0, x);
Out> {x==0};
```

At first sight, the equation $\sqrt{x} = a$ seems to have the solution $x = a^2$. However, this is not true for eg. $a = -1$.

```
In> PrettyPrinter'Set("DefaultPrint");
Out> True;
In> Solve(Sqrt(x) == a, x);
Error: Solve'Fails: cannot solve equation Sqrt(x)-a for  x
Out> {};
In> Solve(Sqrt(x) == 2, x);
Out> {x==4};
In> Solve(Sqrt(x) == -1, x);
Out> {};
```

See also: `Check`, `MatrixSolve`, `Newton`, `OldSolve`, `PrettyPrinter'Set`, `PSolve`, `Where`, `==`

OldSolve — old version of Solve

(standard library)

Calling format:

```
OldSolve(eq, var)
OldSolve(eqlist, varlist)
```

Parameters:

eq – single identity equation
var – single variable
eqlist – list of identity equations
varlist – list of variables

Description:

This is an older version of Solve. It is retained for two reasons. The first one is philosophical: it is good to have multiple algorithms available. The second reason is more practical: the newer version cannot handle systems of equations, but OldSolve can.

This command tries to solve one or more equations. Use the first form to solve a single equation and the second one for systems of equations.

The first calling sequence solves the equation "eq" for the variable "var". Use the == operator to form the equation. The value of "var" which satisfies the equation, is returned. Note that only one solution is found and returned.

To solve a system of equations, the second form should be used. It solves the system of equations contained in the list "eqlist" for the variables appearing in the list "varlist". A list of results is returned, and each result is a list containing the values of the variables in "varlist". Again, at most a single solution is returned.

The task of solving a single equation is simply delegated to SuchThat. Multiple equations are solved recursively: firstly, an equation is sought in which one of the variables occurs exactly once; then this equation is solved with SuchThat; and finally the solution is substituted in the other equations by Eliminate decreasing the number of equations by one. This suffices for all linear equations and a large group of simple nonlinear equations.

Examples:

```
In> OldSolve(a+x*y==z,x)
Out> (z-a)/y;
In> OldSolve({a*x+y==0,x+z==0},{x,y})
Out> {{-z,z*a}};
```

This means that "x = (z-a)/y" is a solution of the first equation and that "x = -z", "y = z*a" is a solution of the systems of equations in the second command.

An example which OldSolve cannot solve:

```
In> OldSolve({x^2-x == y^2-y,x^2-x == y^3+y},{x,y});
Out> {};
```

See also: Solve, SuchThat, Eliminate, PSolve, ==

SuchThat — special purpose solver

(standard library)

Calling format:

```
SuchThat(expr, var)
```

Parameters:

expr – expression to make zero
var – variable (or subexpression) to solve for

Description:

This functions tries to find a value of the variable "var" which makes the expression "expr" zero. It is also possible to pass a subexpression as "var", in which case SuchThat will try to solve for that subexpression.

Basically, only expressions in which "var" occurs only once are handled; in fact, SuchThat may even give wrong results if the variables occurs more than once. This is a consequence of the implementation, which repeatedly applies the inverse of the top function until the variable "var" is reached.

Examples:

```
In> SuchThat(a+b*x, x)
Out> (-a)/b;
In> SuchThat(Cos(a)+Cos(b)^2, Cos(b))
Out> Cos(a)^(1/2);
In> A:=Expand(a*x+b*x+c, x)
Out> (a+b)*x+c;
In> SuchThat(A, x)
Out> (-c)/(a+b);
```

See also: Solve, OldSolve, Subst, Simplify

Eliminate — substitute and simplify

(standard library)

Calling format:

```
Eliminate(var, value, expr)
```

Parameters:

var – variable (or subexpression) to substitute
value – new value of "var"
expr – expression in which the substitution should take place

Description:

This function uses Subst to replace all instances of the variable (or subexpression) "var" in the expression "expr" with "value", calls Simplify to simplify the resulting expression, and returns the result.

Examples:

```
In> Subst(Cos(b), c) (Sin(a)+Cos(b)^2/c)
Out> Sin(a)+c^2/c;
In> Eliminate(Cos(b), c, Sin(a)+Cos(b)^2/c)
Out> Sin(a)+c;
```

See also: SuchThat, Subst, Simplify

PSolve — solve a polynomial equation

(standard library)

Calling format:

```
PSolve(poly, var)
```

Parameters:

`poly` – a polynomial in ”var”
`var` – a variable

Description:

This commands returns a list containing the roots of “poly”, considered as a polynomial in the variable “var”. If there is only one root, it is not returned as a one-entry list but just by itself. A double root occurs twice in the result, and similarly for roots of higher multiplicity. All polynomials of degree up to 4 are handled.

Examples:

```
In> PSolve(b*x+a,x)
Out> -a/b;
In> PSolve(c*x^2+b*x+a,x)
Out> {(Sqrt(b^2-4*c*a)-b)/(2*c),(-(b+
Sqrt(b^2-4*c*a)))/(2*c)};
```

See also: `Solve`, `Factor`

MatrixSolve — solve a system of equations

(standard library)

Calling format:

```
MatrixSolve(A,b)
```

Parameters:

`A` – coefficient matrix
`b` – row vector

Description:

`MatrixSolve` solves the matrix equations `A*x = b` using Gaussian Elimination with Backward substitution. If your matrix is triangular or diagonal, it will be recognized as such and a faster algorithm will be used.

Examples:

```
In> A:={{2,4,-2,-2},{1,2,4,-3},{-3,-3,8,-2},{-1,1,6,-3}};
Out> {{2,4,-2,-2},{1,2,4,-3},{-3,-3,8,-2},{-1,1,6,-3}};
In> b:={-4,5,7,7};
Out> {-4,5,7,7};
In> MatrixSolve(A,b);
Out> {1,2,3,4};
```

Chapter 13

Numeric solvers

Newton — solve an equation numerically with Newton's method

(standard library)

Calling format:

```
Newton(expr, var, initial, accuracy)
Newton(expr, var, initial, accuracy,min,max)
```

Parameters:

`expr` – an expression to find a zero for
`var` – free variable to adjust to find a zero
`initial` – initial value for "var" to use in the search
`accuracy` – minimum required accuracy of the result
`min` – minimum value for "var" to use in the search
`max` – maximum value for "var" to use in the search

Description:

This function tries to numerically find a zero of the expression `expr`, which should depend only on the variable `var`. It uses the value `initial` as an initial guess.

The function will iterate using Newton's method until it estimates that it has come within a distance `accuracy` of the correct solution, and then it will return its best guess. In particular, it may loop forever if the algorithm does not converge.

When `min` and `max` are supplied, the Newton iteration takes them into account by returning `Fail` if it failed to find a root in the given range. Note this doesn't mean there isn't a root, just that this algorithm failed to find it due to the trial values going outside of the bounds.

Examples:

```
In> Newton(Sin(x),x,3,0.0001)
Out> 3.1415926535;
In> Newton(x^2-1,x,2,0.0001,-5,5)
Out> 1;
In> Newton(x^2+1,x,2,0.0001,-5,5)
Out> Fail;
```

See also: `Solve`, `NewtonNum`

FindRealRoots — find the real roots of a polynomial

(standard library)

Calling format:

```
FindRealRoots(p)
```

Parameters:

`p` - a polynomial in `x`

Description:

Return a list with the real roots of p. It tries to find the real-valued roots, and thus requires numeric floating point calculations. The precision of the result can be improved by increasing the calculation precision.

Examples:

```
In> p:=Expand((x+3.1)^5*(x-6.23))
Out> x^6+9.27*x^5-0.465*x^4-300.793*x^3-
1394.2188*x^2-2590.476405*x-1783.5961073;
In> FindRealRoots(p)
Out> {-3.1,6.23};
```

See also: `SquareFree`, `NumRealRoots`, `MinimumBound`, `MaximumBound`, `Factor`

NumRealRoots — return the number of real roots of a polynomial

(standard library)

Calling format:

```
NumRealRoots(p)
```

Parameters:

`p` - a polynomial in `x`

Description:

Returns the number of real roots of a polynomial p. The polynomial must use the variable `x` and no other variables.

Examples:

```
In> NumRealRoots(x^2-1)
Out> 2;
In> NumRealRoots(x^2+1)
Out> 0;
```

See also: `FindRealRoots`, `SquareFree`, `MinimumBound`, `MaximumBound`, `Factor`

MinimumBound — return lower bounds on the absolute values of real roots of a polynomial

MaximumBound — return upper bounds on the absolute values of real roots of a polynomial

(standard library)

Calling format:

```
MinimumBound(p)
MaximumBound(p)
```

Parameters:

p - a polynomial in x

Description:

Return minimum and maximum bounds for the absolute values of the real roots of a polynomial p. The polynomial has to be converted to one with rational coefficients first, and be made square-free. The polynomial must use the variable x.

Examples:

```
In> p:=SquareFree(Rationalize((x-3.1)*(x+6.23)))
Out> (-40000*x^2-125200*x+772520)/870489;
In> MinimumBound(p)
Out> 5000000000/2275491039;
In> N(%)
Out> 2.1973279236;
In> MaximumBound(p)
Out> 10986639613/1250000000;
In> N(%)
Out> 8.7893116904;
```

See also: SquareFree, NumRealRoots, FindRealRoots, Factor

Chapter 14

Propositional logic theorem prover

CanProve — try to prove statement

(standard library)

Calling format:

```
CanProve(proposition)
```

Parameters:

`proposition` – an expression with logical operations

Description:

Yacas has a small built-in propositional logic theorem prover. It can be invoked with a call to `CanProve`.

An example of a proposition is: “if a implies b and b implies c then a implies c”. Yacas supports the following logical operations:

`Not` : negation, read as ”not”
`And` : conjunction, read as ”and”
`Or` : disjunction, read as ”or”
`=>` : implication, read as ”implies”

The abovementioned proposition would be represented by the following expression,

```
( (a=>b) And (b=>c) ) => (a=>c)
```

Yacas can prove that is correct by applying `CanProve` to it:

```
In> CanProve(( (a=>b) And (b=>c) ) => (a=>c))
Out> True;
```

It does this in the following way: in order to prove a proposition p, it suffices to prove that $\neg p$ is false. It continues to simplify $\neg p$ using the rules:

```
Not  ( Not x)      --> x
```

(eliminate double negation),

```
x=>y  -->  Not x  Or  y
```

(eliminate implication),

```
Not (x And y)  -->  Not x  Or  Not y
```

(De Morgan’s law),

```
Not (x Or y) -->  Not x  And  Not y
```

(De Morgan’s law),

```
(x And y) Or z --> (x Or z) And (y Or z)
```

(distribution),

```
x Or (y And z) --> (x Or y) And (x Or z)
```

(distribution), and the obvious other rules, such as,

```
True Or x --> True
```

etc. The above rules will translate a proposition into a form

```
(p1  Or  p2  Or  ...)  And  (q1  Or  q2
   Or  ...)  And ...
```

If any of the clauses is false, the entire expression will be false. In the next step, clauses are scanned for situations of the form:

```
(p Or Y)  And  ( Not p Or Z) --> (Y Or Z)
```

If this combination `(Y Or Z)` is empty, it is false, and thus the entire proposition is false.

As a last step, the algorithm negates the result again. This has the added advantage of simplifying the expression further.

Examples:

```
In> CanProve(a  Or   Not a)
Out> True;
In> CanProve(True  Or  a)
Out> True;
In> CanProve(False  Or  a)
Out> a;
In> CanProve(a  And   Not a)
Out> False;
In> CanProve(a  Or b Or (a And b))
Out> a Or b;
```

See also: `True`, `False`, `And`, `Or`, `Not`

Chapter 15

Differential Equations

In this chapter, some facilities for solving differential equations are described. Currently only simple equations without auxiliary conditions are supported.

OdeSolve — general ODE solver

(standard library)

Calling format:

```
OdeSolve(expr1==expr2)
```

Parameters:

`expr1,expr2` – expressions containing a function to solve for

Description:

This function currently can solve second order homogeneous linear real constant coefficient equations. The solution is returned with unique constants generated by `UniqueConstant`. The roots of the auxiliary equation are used as the arguments of exponentials. If the roots are complex conjugate pairs, then the solution returned is in the form of exponentials, sines and cosines.

First and second derivatives are entered as `y',y''`. Higher order derivatives may be entered as `y(n)`, where `n` is any integer.

Examples:

```
In> OdeSolve( y'' + y == 0 )
Out> C42*Sin(x)+C43*Cos(x);
In> OdeSolve( 2*y'' + 3*y' + 5*y == 0 )
Out> Exp(((-3)*x)/4)*(C78*Sin(Sqrt(31/16)*x)+C79*Cos(Sqrt(31/16)*x));
In> OdeSolve( y'' - 4*y == 0 )
Out> C132*Exp((-2)*x)+C136*Exp(2*x);
In> OdeSolve( y'' +2*y' + y == 0 )
Out> (C183+C184*x)*Exp(-x);
```

See also: `Solve`, `RootsWithMultiples`

OdeTest — test the solution of an ODE

(standard library)

Calling format:

```
OdeTest(eqn,testsol)
```

Parameters:

`eqn` – equation to test

`testsol` – test solution

Description:

This function automates the verification of the solution of an ODE. It can also be used to quickly see how a particular equation operates on a function.

Examples:

```
In> OdeTest(y''+y,Sin(x)+Cos(x))
Out> 0;
In> OdeTest(y''+2*y,Sin(x)+Cos(x))
Out> Sin(x)+Cos(x);
```

See also: `OdeSolve`

OdeOrder — return order of an ODE

(standard library)

Calling format:

```
OdeOrder(eqn)
```

Parameters:

`eqn` – equation

Description:

This function returns the order of the differential equation, which is order of the highest derivative. If no derivatives appear, zero is returned.

Examples:

```
In> OdeOrder(y'' + 2*y' == 0)
Out> 2;
In> OdeOrder(Sin(x)*y(5) + 2*y' == 0)
Out> 5;
In> OdeOrder(2*y + Sin(y) == 0)
Out> 0;
```

See also: `OdeSolve`

Chapter 16

Linear Algebra

This chapter describes the commands for doing linear algebra. They can be used to manipulate vectors, represented as lists, and matrices, represented as lists of lists.

Dot, . — get dot product of tensors

(standard library)

Calling format:

```
Dot(t1,t2)
t1 . t2
```

Precedence: 30

Parameters:

t1,t2 – tensor lists (currently only vectors and matrices are supported)

Description:

Dot returns the dot (aka inner) product of two tensors t1 and t2. The last index of t1 and the first index of t2 are contracted. Currently Dot works only for vectors and matrices. Dot-multiplication of two vectors, a matrix with a vector (and vice versa) or two matrices yields either a scalar, a vector or a matrix.

Examples:

```
In> Dot({1,2},{3,4})
Out> 11;
In> Dot({{1,2},{3,4}},{5,6})
Out> {17,39};
In> Dot({5,6},{{1,2},{3,4}})
Out> {23,34};
In> Dot({{1,2},{3,4}},{{5,6},{7,8}})
Out> {{19,22},{43,50}};
```

```
Or, using the "."-Operator:
```

```
In> {1,2} . {3,4}
Out> 11;
In> {{1,2},{3,4}} . {5,6}
Out> {17,39};
In> {5,6} . {{1,2},{3,4}}
Out> {23,34};
In> {{1,2},{3,4}} . {{5,6},{7,8}}
Out> {{19,22},{43,50}};
```

See also: Outer, Cross, IsScalar, IsVector, IsMatrix

InProduct — inner product of vectors (deprecated)

(standard library)

Calling format:

```
InProduct(a,b)
```

Parameters:

a, b – vectors of equal length

Description:

The inner product of the two vectors "a" and "b" is returned. The vectors need to have the same size.

This function is superceded by the . operator.

Examples:

```
In> {a,b,c} . {d,e,f};
Out> a*d+b*e+c*f;
```

See also: Dot, CrossProduct

CrossProduct — outer product of vectors

(standard library)

Calling format:

```
CrossProduct(a,b)
a X b
```

Precedence: 30

Parameters:

a, b – three-dimensional vectors

Description:

The cross product of the vectors "a" and "b" is returned. The result is perpendicular to both "a" and "b" and its length is the product of the lengths of the vectors. Both "a" and "b" have to be three-dimensional.

Examples:

```
In> {a,b,c} X {d,e,f};
Out> {b*f-c*e,c*d-a*f,a*e-b*d};
```

See also: InProduct

`Outer`, `o` — get outer tensor product

(standard library)

Calling format:

```
Outer(t1,t2)
t1 o t2
```

Precedence: 30

Parameters:

`t1,t2` – tensor lists (currently only vectors are supported)

Description:

`Outer` returns the outer product of two tensors t1 and t2. Currently `Outer` work works only for vectors, i.e. tensors of rank 1. The outer product of two vectors yields a matrix.

Examples:

```
In> Outer({1,2},{3,4,5})
Out> {{3,4,5},{6,8,10}};
In> Outer({a,b},{c,d})
Out> {{a*c,a*d},{b*c,b*d}};

Or, using the "o"-Operator:

In> {1,2} o {3,4,5}
Out> {{3,4,5},{6,8,10}};
In> {a,b} o {c,d}
Out> {{a*c,a*d},{b*c,b*d}};
```

See also: `Dot`, `Cross`

`ZeroVector` — create a vector with all zeroes

(standard library)

Calling format:

```
ZeroVector(n)
```

Parameters:

`n` – length of the vector to return

Description:

This command returns a vector of length "n", filled with zeroes.

Examples:

```
In> ZeroVector(4)
Out> {0,0,0,0};
```

See also: `BaseVector`, `ZeroMatrix`, `IsZeroVector`

`BaseVector` — base vector

(standard library)

Calling format:

```
BaseVector(k, n)
```

Parameters:

`k` – index of the base vector to construct
`n` – dimension of the vector

Description:

This command returns the "k"-th base vector of dimension "n". This is a vector of length "n" with all zeroes except for the "k"-th entry, which contains a 1.

Examples:

```
In> BaseVector(2,4)
Out> {0,1,0,0};
```

See also: `ZeroVector`, `Identity`

`Identity` — make identity matrix

(standard library)

Calling format:

```
Identity(n)
```

Parameters:

`n` – size of the matrix

Description:

This commands returns the identity matrix of size "n" by "n". This matrix has ones on the diagonal while the other entries are zero.

Examples:

```
In> Identity(3)
Out> {{1,0,0},{0,1,0},{0,0,1}};
```

See also: `BaseVector`, `ZeroMatrix`, `DiagonalMatrix`

`ZeroMatrix` — make a zero matrix

(standard library)

Calling format:

```
ZeroMatrix(n)
ZeroMatrix(n, m)
```

Parameters:

`n` – number of rows
`m` – number of columns

Description:

This command returns a matrix with `n` rows and `m` columns, completely filled with zeroes. If only given one parameter, it returns the square `n` by `n` zero matrix.

Examples:

```
In> ZeroMatrix(3,4)
Out> {{0,0,0,0},{0,0,0,0},{0,0,0,0}};
In> ZeroMatrix(3)
Out> {{0,0,0},{0,0,0},{0,0,0}};
```

See also: `ZeroVector`, `Identity`

Diagonal — extract the diagonal from a matrix

(standard library)

Calling format:

```
Diagonal(A)
```

Parameters:

`A` – matrix

Description:

This command returns a vector of the diagonal components of the matrix `A`.

Examples:

```
In> Diagonal(5*Identity(4))
Out> {5,5,5,5};
In> Diagonal(HilbertMatrix(3))
Out> {1,1/3,1/5};
```

See also: `DiagonalMatrix`, `IsDiagonal`

DiagonalMatrix — construct a diagonal matrix

(standard library)

Calling format:

```
DiagonalMatrix(d)
```

Parameters:

`d` – list of values to put on the diagonal

Description:

This command constructs a diagonal matrix, that is a square matrix whose off-diagonal entries are all zero. The elements of the vector "d" are put on the diagonal.

Examples:

```
In> DiagonalMatrix(1 .. 4)
Out> {{1,0,0,0},{0,2,0,0},{0,0,3,0},{0,0,0,4}};
```

See also: `Identity`, `ZeroMatrix`

OrthogonalBasis — create an orthogonal basis

(standard library)

Calling format:

```
OrthogonalBasis(W)
```

Parameters:

`W` - A linearly independent set of row vectors (aka a matrix)

Description:

Given a linearly independent set `W` (constructed of rows vectors), this command returns an orthogonal basis `V` for `W`, which means that span(V) = span(W) and `InProduct(V[i],V[j]) = 0` when `i != j`. This function uses the Gram-Schmidt orthogonalization process.

Examples:

```
In> OrthogonalBasis({{1,1,0},{2,0,1},{2,2,1}})
Out> {{1,1,0},{1,-1,1},{-1/3,1/3,2/3}};
```

See also: `OrthonormalBasis`, `InProduct`

OrthonormalBasis — create an orthonormal basis

(standard library)

Calling format:

```
OrthonormalBasis(W)
```

Parameters:

`W` - A linearly independent set of row vectors (aka a matrix)

Description:

Given a linearly independent set `W` (constructed of rows vectors), this command returns an orthonormal basis `V` for `W`. This is done by first using `OrthogonalBasis(W)`, then dividing each vector by its magnitude, so as the give them unit length.

Examples:

```
In> OrthonormalBasis({{1,1,0},{2,0,1},{2,2,1}})
Out> {{Sqrt(1/2),Sqrt(1/2),0},{Sqrt(1/3),-Sqrt(1/3),Sqrt(1/
{-Sqrt(1/6),Sqrt(1/6),Sqrt(2/3)}};
```

See also: `OrthogonalBasis`, `InProduct`, `Normalize`

Normalize — normalize a vector

(standard library)

Calling format:

```
Normalize(v)
```

Parameters:

`v` – a vector

Description:

Return the normalized (unit) vector parallel to `v`: a vector having the same direction but with length 1.

Examples:

```
In> v:=Normalize({3,4})
Out> {3/5,4/5};
In> v . v
Out> 1;
```

See also: `InProduct`, `CrossProduct`

Transpose — get transpose of a matrix

(standard library)

Calling format:

```
Transpose(M)
```

Parameters:

`M` – a matrix

Description:

`Transpose` returns the transpose of a matrix M. Because matrices are just lists of lists, this is a useful operation too for lists.

Examples:

```
In> Transpose({{a,b}})
Out> {{a},{b}};
```

Determinant — determinant of a matrix

(standard library)

Calling format:

```
Determinant(M)
```

Parameters:

`M` – a matrix

Description:

Returns the determinant of a matrix M.

Examples:

```
In> A:=DiagonalMatrix(1 .. 4)
Out> {{1,0,0,0},{0,2,0,0},{0,0,3,0},{0,0,0,4}};
In> Determinant(A)
Out> 24;
```

Trace — trace of a matrix

(standard library)

Calling format:

```
Trace(M)
```

Parameters:

`M` – a matrix

Description:

`Trace` returns the trace of a matrix M (defined as the sum of the elements on the diagonal of the matrix).

Examples:

```
In> A:=DiagonalMatrix(1 .. 4)
Out> {{1,0,0,0},{0,2,0,0},{0,0,3,0},{0,0,0,4}};
In> Trace(A)
Out> 10;
```

Inverse — get inverse of a matrix

(standard library)

Calling format:

```
Inverse(M)
```

Parameters:

`M` – a matrix

Description:

Inverse returns the inverse of matrix M. The determinant of M should be non-zero. Because this function uses `Determinant` for calculating the inverse of a matrix, you can supply matrices with non-numeric (symbolic) matrix elements.

Examples:

```
In> A:=DiagonalMatrix({a,b,c})
Out> {{a,0,0},{0,b,0},{0,0,c}};
In> B:=Inverse(A)
Out> {{(b*c)/(a*b*c),0,0},{0,(a*c)/(a*b*c),0},
{0,0,(a*b)/(a*b*c)}};
In> Simplify(B)
Out> {{1/a,0,0},{0,1/b,0},{0,0,1/c}};
```

See also: `Determinant`

Minor — get principal minor of a matrix

(standard library)

Calling format:

```
Minor(M,i,j)
```

Parameters:

`M` – a matrix
`i`, `j` - positive integers

Description:

Minor returns the minor of a matrix around the element (i, j). The minor is the determinant of the matrix obtained from M by deleting the i-th row and the j-th column.

Examples:

```
In> A := {{1,2,3}, {4,5,6}, {7,8,9}};
Out> {{1,2,3},{4,5,6},{7,8,9}};
In> PrettyForm(A);

/                    \
| ( 1 ) ( 2 ) ( 3 )  |
|                    |
| ( 4 ) ( 5 ) ( 6 )  |
|                    |
| ( 7 ) ( 8 ) ( 9 )  |
\                    /
Out> True;
In> Minor(A,1,2);
Out> -6;
In> Determinant({{2,3}, {8,9}});
Out> -6;
```

See also: `CoFactor`, `Determinant`, `Inverse`

`CoFactor` — cofactor of a matrix

(standard library)

Calling format:

```
CoFactor(M,i,j)
```

Parameters:

`M` – a matrix
`i`, `j` - positive integers

Description:

`CoFactor` returns the cofactor of a matrix around the element (i, j). The cofactor is the minor times $(-1)^{i+j}$.

Examples:

```
In> A := {{1,2,3}, {4,5,6}, {7,8,9}};
Out> {{1,2,3},{4,5,6},{7,8,9}};
In> PrettyForm(A);

/                    \
| ( 1 ) ( 2 ) ( 3 )  |
|                    |
| ( 4 ) ( 5 ) ( 6 )  |
|                    |
| ( 7 ) ( 8 ) ( 9 )  |
\                    /
Out> True;
In> CoFactor(A,1,2);
Out> 6;
In> Minor(A,1,2);
Out> -6;
In> Minor(A,1,2) * (-1)^(1+2);
Out> 6;
```

See also: `Minor`, `Determinant`, `Inverse`

`MatrixPower` — get nth power of a square matrix

(standard library)

Calling format:

```
MatrixPower(mat,n)
```

Parameters:

`mat` – a square matrix
`n` – an integer

Description:

`MatrixPower(mat,n)` returns the `n`th power of a square matrix `mat`. For positive `n` it evaluates dot products of `mat` with itself. For negative `n` the nth power of the inverse of `mat` is returned. For `n`=0 the identity matrix is returned.

Example:

```
In> A:={{1,2},{3,4}}
Out> {{1,2},{3,4}};
In> MatrixPower(A,0)
Out> {{1,0},{0,1}};
In> MatrixPower(A,1)
Out> {{1,2},{3,4}};
In> MatrixPower(A,3)
Out> {{37,54},{81,118}};
In> MatrixPower(A,-3)
Out> {{-59/4,27/4},{81/8,-37/8}};
```

See also: `IsSquareMatrix`, `Inverse`, `Dot`

`SolveMatrix` — solve a linear system

(standard library)

Calling format:

```
SolveMatrix(M,v)
```

Parameters:

`M` – a matrix
`v` – a vector

Description:

`SolveMatrix` returns the vector x that satisfies the equation $Mx = v$. The determinant of M should be non-zero.

Examples:

```
In> A := {{1,2}, {3,4}};
Out> {{1,2},{3,4}};
In> v := {5,6};
Out> {5,6};
In> x := SolveMatrix(A, v);
Out> {-4,9/2};
In> A * x;
Out> {5,6};
```

See also: `Inverse`, `Solve`, `PSolve`, `Determinant`

`CharacteristicEquation` — get characteristic polynomial of a matrix

(standard library)

Calling format:

```
CharacteristicEquation(matrix,var)
```

Parameters:

`matrix` – a matrix
`var` – a free variable

Description:

CharacteristicEquation returns the characteristic equation of "matrix", using "var". The zeros of this equation are the eigenvalues of the matrix, Det(matrix-I*var);

Examples:

```
In> A:=DiagonalMatrix({a,b,c})
Out> {{a,0,0},{0,b,0},{0,0,c}};
In> B:=CharacteristicEquation(A,x)
Out> (a-x)*(b-x)*(c-x);
In> Expand(B,x)
Out> (b+a+c)*x^2-x^3-((b+a)*c+a*b)*x+a*b*c;
```

See also: `EigenValues`, `EigenVectors`

EigenValues — get eigenvalues of a matrix

(standard library)

Calling format:

```
EigenValues(matrix)
```

Parameters:

`matrix` – a square matrix

Description:

EigenValues returns the eigenvalues of a matrix. The eigenvalues x of a matrix M are the numbers such that $Mv = xv$ for some vector.

It first determines the characteristic equation, and then factorizes this equation, returning the roots of the characteristic equation Det(matrix-x*identity).

Examples:

```
In> M:={{1,2},{2,1}}
Out> {{1,2},{2,1}};
In> EigenValues(M)
Out> {3,-1};
```

See also: `EigenVectors`, `CharacteristicEquation`

EigenVectors — get eigenvectors of a matrix

(standard library)

Calling format:

```
EigenVectors(A,eigenvalues)
```

Parameters:

`matrix` – a square matrix
`eigenvalues` – list of eigenvalues as returned by `EigenValues`

Description:

`EigenVectors` returns a list of the eigenvectors of a matrix. It uses the eigenvalues and the matrix to set up n equations with n unknowns for each eigenvalue, and then calls `Solve` to determine the values of each vector.

Examples:

```
In> M:={{1,2},{2,1}}
Out> {{1,2},{2,1}};
In> e:=EigenValues(M)
Out> {3,-1};
In> EigenVectors(M,e)
Out> {{-ki2/ -1,ki2},{-ki2,ki2}};
```

See also: `EigenValues`, `CharacteristicEquation`

Sparsity — get the sparsity of a matrix

(standard library)

Calling format:

```
Sparsity(matrix)
```

Parameters:

`matrix` – a matrix

Description:

The function `Sparsity` returns a number between `0` and `1` which represents the percentage of zero entries in the matrix. Although there is no definite critical value, a sparsity of `0.75` or more is almost universally considered a "sparse" matrix. These type of matrices can be handled in a different manner than "full" matrices which speedup many calculations by orders of magnitude.

Examples:

```
In> Sparsity(Identity(2))
Out> 0.5;
In> Sparsity(Identity(10))
Out> 0.9;
In> Sparsity(HankelMatrix(10))
Out> 0.45;
In> Sparsity(HankelMatrix(100))
Out> 0.495;
In> Sparsity(HilbertMatrix(10))
Out> 0;
In> Sparsity(ZeroMatrix(10,10))
Out> 1;
```

Cholesky — find the Cholesky Decomposition

(standard library)

Calling format:

```
Cholesky(A)
```

Parameters:

`A` – a square positive definite matrix

Description:

`Cholesky` returns a upper triangular matrix `R` such that `Transpose(R)*R = A`. The matrix `A` must be positive definite, `Cholesky` will notify the user if the matrix is not. Some families of positive definite matrices are all symmetric matrices, diagonal matrices with positive elements and Hilbert matrices.

Examples:

```
In> A:={{4,-2,4,2},{-2,10,-2,-7},{4,-2,8,4},{2,-7,4,7}}
Out> {{4,-2,4,2},{-2,10,-2,-7},{4,-2,8,4},{2,-7,4,7}};
In> R:=Cholesky(A);
Out> {{2,-1,2,1},{0,3,0,-2},{0,0,2,1},{0,0,0,1}};
In> Transpose(R)*R = A
Out> True;
```

```
In> Cholesky(4*Identity(5))
Out> {{2,0,0,0,0},{0,2,0,0,0},{0,0,2,0,0},{0,0,0,2,0},{0,0,0,0,2}};
In> Cholesky(HilbertMatrix(3))
Out> {{1,1/2,1/3},{0,Sqrt(1/12),Sqrt(1/12)},{0,0,Sqrt(1/180)}};
In> Cholesky(ToeplitzMatrix({1,2,3}))
In function "Check" :
CommandLine(1) : "Cholesky: Matrix is not positive definite"
```

See also: `IsSymmetric`, `IsDiagonal`, `Diagonal`

Chapter 17

Predicates related to matrices

IsScalar — test for a scalar

(standard library)

Calling format:

```
IsScalar(expr)
```

Parameters:

expr – a mathematical object

Description:

IsScalar returns True if expr is a scalar, False otherwise. Something is considered to be a scalar if it's not a list.

Examples:

```
In> IsScalar(7)
Out> True;
In> IsScalar(Sin(x)+x)
Out> True;
In> IsScalar({x,y})
Out> False;
```

See also: IsList, IsVector, IsMatrix

IsVector — test for a vector

(standard library)

Calling format:

```
IsVector(expr)
```

```
IsVector(pred,expr)
```

Parameters:

expr – expression to test
pred – predicate test (e.g. IsNumber, IsInteger, ...)

Description:

IsVector(expr) returns True if expr is a vector, False otherwise. Something is considered to be a vector if it's a list of scalars. IsVector(pred,expr) returns True if expr is a vector and if the predicate test pred returns True when applied to every element of the vector expr, False otherwise.

Examples:

```
In> IsVector({a,b,c})
Out> True;
In> IsVector({a,{b},c})
Out> False;
In> IsVector(IsInteger,{1,2,3})
Out> True;
In> IsVector(IsInteger,{1,2.5,3})
Out> False;
```

See also: IsList, IsScalar, IsMatrix

IsMatrix — test for a matrix

(standard library)

Calling format:

```
IsMatrix(expr)
```

```
IsMatrix(pred,expr)
```

Parameters:

expr – expression to test
pred – predicate test (e.g. IsNumber, IsInteger, ...)

Description:

IsMatrix(expr) returns True if expr is a matrix, False otherwise. Something is considered to be a matrix if it's a list of vectors of equal length. IsMatrix(pred,expr) returns True if expr is a matrix and if the predicate test pred returns True when applied to every element of the matrix expr, False otherwise.

Examples:

```
In> IsMatrix(1)
Out> False;
In> IsMatrix({1,2})
Out> False;
In> IsMatrix({{1,2},{3,4}})
Out> True;
In> IsMatrix(IsRational,{{1,2},{3,4}})
Out> False;
In> IsMatrix(IsRational,{{1/2,2/3},{3/4,4/5}})
Out> True;
```

See also: IsList, IsVector

IsSquareMatrix — test for a square matrix

(standard library)

Calling format:

```
IsSquareMatrix(expr)
IsSquareMatrix(pred,expr)
```

Parameters:

`expr` – expression to test
`pred` – predicate test (e.g. IsNumber, IsInteger, ...)

Description:

`IsSquareMatrix(expr)` returns `True` if `expr` is a square matrix, `False` otherwise. Something is considered to be a square matrix if it's a matrix having the same number of rows and columns. `IsMatrix(pred,expr)` returns `True` if `expr` is a square matrix and if the predicate test `pred` returns `True` when applied to every element of the matrix `expr`, `False` otherwise.

Examples:

```
In> IsSquareMatrix({{1,2},{3,4}});
Out> True;
In> IsSquareMatrix({{1,2,3},{4,5,6}});
Out> False;
In> IsSquareMatrix(IsBoolean,{{1,2},{3,4}});
Out> False;
In> IsSquareMatrix(IsBoolean,{{True,False},{False,True}});
Out> True;
```

See also: `IsMatrix`

IsHermitian — test for a Hermitian matrix

(standard library)

Calling format:

```
IsHermitian(A)
```

Parameters:

`A` – a square matrix

Description:

IsHermitian(A) returns `True` if `A` is Hermitian and `False` otherwise. A is a Hermitian matrix iff Conjugate(Transpose A)=A. If A is a real matrix, it must be symmetric to be Hermitian.

Examples:

```
In> IsHermitian({{0,I},{-I,0}})
Out> True;
In> IsHermitian({{0,I},{2,0}})
Out> False;
```

See also: `IsUnitary`

IsOrthogonal — test for an orthogonal matrix

(standard library)

Calling format:

```
IsOrthogonal(A)
```

Parameters:

`A` – square matrix

Description:

`IsOrthogonal(A)` returns `True` if `A` is orthogonal and `False` otherwise. A is orthogonal iff A*Transpose(A) = Identity, or equivalently Inverse(A) = Transpose(A).

Examples:

```
In> A := {{1,2,2},{2,1,-2},{-2,2,-1}};
Out> {{1,2,2},{2,1,-2},{-2,2,-1}};
In> PrettyForm(A/3)

/                             \
| / 1 \  / 2 \ / 2 \          |
| | - |  | - | | - |          |
| \ 3 /  \ 3 / \ 3 /          |
|                             |
| / 2 \  / 1 \ / -2 \         |
| | - |  | - | | -- |         |
| \ 3 /  \ 3 / \ 3  /         |
|                             |
| / -2 \ / 2 \ / -1 \         |
| | -- | | - | | -- |         |
| \ 3  / \ 3 / \ 3  /         |
\                             /
Out> True;
In> IsOrthogonal(A/3)
Out> True;
```

IsDiagonal — test for a diagonal matrix

(standard library)

Calling format:

```
IsDiagonal(A)
```

Parameters:

`A` – a matrix

Description:

`IsDiagonal(A)` returns `True` if `A` is a diagonal square matrix and `False` otherwise.

Examples:

```
In> IsDiagonal(Identity(5))
Out> True;
In> IsDiagonal(HilbertMatrix(5))
Out> False;
```

IsLowerTriangular — test for a lower triangular matrix

IsUpperTriangular — test for an upper triangular matrix

(standard library)

Calling format:

```
IsLowerTriangular(A)
IsUpperTriangular(A)
```

Parameters:

A – a matrix

Description:

A lower/upper triangular matrix is a square matrix which has all zero entries above/below the diagonal.

IsLowerTriangular(A) returns True if A is a lower triangular matrix and False otherwise. IsUpperTriangular(A) returns True if A is an upper triangular matrix and False otherwise.

Examples:

```
In> IsUpperTriangular(Identity(5))
Out> True;
In> IsLowerTriangular(Identity(5))
Out> True;
In> IsLowerTriangular({{1,2},{0,1}})
Out> False;
In> IsUpperTriangular({{1,2},{0,1}})
Out> True;
```

A non-square matrix cannot be triangular:

```
In> IsUpperTriangular({{1,2,3},{0,1,2}})
Out> False;
```

See also: IsDiagonal

IsSymmetric — test for a symmetric matrix

(standard library)

Calling format:

```
IsSymmetric(A)
```

Parameters:

A – a matrix

Description:

IsSymmetric(A) returns True if A is symmetric and False otherwise. A is symmetric iff Transpose $(A) = A$.

Examples:

```
In> A := {{1,0,0,0,1},{0,2,0,0,0},{0,0,3,0,0},
  {0,0,0,4,0},{1,0,0,0,5}};
In> PrettyForm(A)
/                                  \
| ( 1 ) ( 0 ) ( 0 ) ( 0 ) ( 1 )  |
|                                  |
| ( 0 ) ( 2 ) ( 0 ) ( 0 ) ( 0 )  |
|                                  |
| ( 0 ) ( 0 ) ( 3 ) ( 0 ) ( 0 )  |
|                                  |
| ( 0 ) ( 0 ) ( 0 ) ( 4 ) ( 0 )  |
|                                  |
| ( 1 ) ( 0 ) ( 0 ) ( 0 ) ( 5 )  |
\                                  /
Out> True;
In> IsSymmetric(A)
Out> True;
```

See also: IsHermitian, IsSkewSymmetric

IsSkewSymmetric — test for a skew-symmetric matrix

(standard library)

Calling format:

```
IsSkewSymmetric(A)
```

Parameters:

A – a square matrix

Description:

IsSkewSymmetric(A) returns True if A is skew symmetric and False otherwise. A is skew symmetric iff Transpose $(A) = -A$.

Examples:

```
In> A := {{0,-1},{1,0}}
Out> {{0,-1},{1,0}};
In> PrettyForm(%)
/                \
| ( 0 ) ( -1 )  |
|                |
| ( 1 ) ( 0 )   |
\                /
Out> True;
In> IsSkewSymmetric(A);
Out> True;
```

See also: IsSymmetric, IsHermitian

IsUnitary — test for a unitary matrix

(standard library)

Calling format:

```
IsUnitary(A)
```

Parameters:

A – a square matrix

Description:

This function tries to find out if A is unitary.

A matrix A is orthogonal iff A^{-1} = Transpose(Conjugate(A)). This is equivalent to the fact that the columns of A build an orthonormal system (with respect to the scalar product defined by `InProduct`).

Examples:

```
In> IsUnitary({{0,I},{-I,0}})
Out> True;
In> IsUnitary({{0,I},{2,0}})
Out> False;
```

See also: `IsHermitian`, `IsSymmetric`

`IsIdempotent` — test for an idempotent matrix

(standard library)

Calling format:

```
IsIdempotent(A)
```

Parameters:

`A` – a square matrix

Description:

`IsIdempotent(A)` returns `True` if `A` is idempotent and `False` otherwise. A is idempotent iff $A^2 = A$. Note that this also implies that A raised to any power is also equal to A.

Examples:

```
In> IsIdempotent(ZeroMatrix(10,10));
Out> True;
In> IsIdempotent(Identity(20))
Out> True;
```

Chapter 18

Special matrices

JacobianMatrix — calculate the Jacobian matrix of n functions in n variables

(standard library)

Calling format:

```
JacobianMatrix(functions,variables)
```

Parameters:

`functions` – an n-dimensional vector of functions
`variables` – an n-dimensional vector of variables

Description:

The function `JacobianMatrix` calculates the Jacobian matrix of n functions in n variables.

The (i,j)-th element of the Jacobian matrix is defined as the derivative of i-th function with respect to the j-th variable.

Examples:

```
In> JacobianMatrix( {Sin(x),Cos(y)}, {x,y} );
Out> {{Cos(x),0},{0,-Sin(y)}};
In> PrettyForm(%)
```

```
/                                \
| ( Cos( x ) ) ( 0 )             |
|                                |
| ( 0 )        ( -( Sin( y ) ) ) |
\                                /
```

VandermondeMatrix — create the Vandermonde matrix

(standard library)

Calling format:

```
VandermondeMatrix(vector)
```

Parameters:

`vector` – an n-dimensional vector

Description:

The function `VandermondeMatrix` calculates the Vandermonde matrix of a vector.

The (i,j)-th element of the Vandermonde matrix is defined as i^{j-1}.

Examples:

```
In> VandermondeMatrix({1,2,3,4})
Out> {{1,1,1,1},{1,2,3,4},{1,4,9,16},{1,8,27,64}};
In>PrettyForm(%)
```

```
/                               \
| ( 1 ) ( 1 ) ( 1 )  ( 1 )      |
|                               |
| ( 1 ) ( 2 ) ( 3 )  ( 4 )      |
|                               |
| ( 1 ) ( 4 ) ( 9 )  ( 16 )     |
|                               |
| ( 1 ) ( 8 ) ( 27 ) ( 64 )     |
\                               /
```

HessianMatrix — create the Hessian matrix

(standard library)

Calling format:

```
HessianMatrix(function,var)
```

Parameters:

`function` – a function in n variables
`var` – an n-dimensional vector of variables

Description:

The function `HessianMatrix` calculates the Hessian matrix of a vector.

If $f(x)$ is a function of an n-dimensional vector x, then the (i,j)-th element of the Hessian matrix of the function $f(x)$ is defined as $\frac{\partial}{\partial x_i}\left(\frac{d}{dx_j} f(x)\right)$. If the third order mixed partials are continuous, then the Hessian matrix is symmetric (a standard theorem of calculus).

The Hessian matrix is used in the second derivative test to discern if a critical point is a local maximum, a local minimum or a saddle point.

Examples:

```
In> HessianMatrix(3*x^2-2*x*y+y^2-8*y, {x,y} )
Out> {{6,-2},{-2,2}};
In> PrettyForm(%)
```

```
/                 \
| ( 6 )  ( -2 )   |
|                 |
| ( -2 ) ( 2 )    |
\                 /
```

HilbertMatrix — create a Hilbert matrix

(standard library)

Calling format:

```
HilbertMatrix(n)
HilbertMatrix(n,m)
```

Parameters:

n,m – positive integers

Description:

The function HilbertMatrix returns the n by m Hilbert matrix if given two arguments, and the square n by n Hilbert matrix if given only one. The Hilbert matrix is defined as A(i,j) = 1/(i+j-1). The Hilbert matrix is extremely sensitive to manipulate and invert numerically.

Examples:

```
In> PrettyForm(HilbertMatrix(4))

/                          \
| ( 1 ) / 1 \ / 1 \ / 1 \  |
|       | - | | - | | - |  |
|       \ 2 / \ 3 / \ 4 /  |
|                          |
| / 1 \ / 1 \ / 1 \ / 1 \  |
| | - | | - | | - | | - |  |
| \ 2 / \ 3 / \ 4 / \ 5 /  |
|                          |
| / 1 \ / 1 \ / 1 \ / 1 \  |
| | - | | - | | - | | - |  |
| \ 3 / \ 4 / \ 5 / \ 6 /  |
|                          |
| / 1 \ / 1 \ / 1 \ / 1 \  |
| | - | | - | | - | | - |  |
| \ 4 / \ 5 / \ 6 / \ 7 /  |
\                          /
```

See also: HilbertInverseMatrix

HilbertInverseMatrix — create a Hilbert inverse matrix

(standard library)

Calling format:

```
HilbertInverseMatrix(n)
```

Parameters:

n – positive integer

Description:

The function HilbertInverseMatrix returns the n by n inverse of the corresponding Hilbert matrix. All Hilbert inverse matrices have integer entries that grow in magnitude rapidly.

Examples:

```
In> PrettyForm(HilbertInverseMatrix(4))

/                                          \
| ( 16 )   ( -120 )  ( 240 )   ( -140 )    |
|                                          |
| ( -120 ) ( 1200 )  ( -2700 ) ( 1680 )    |
|                                          |
| ( 240 )  ( -2700 ) ( 6480 )  ( -4200 )   |
|                                          |
| ( -140 ) ( 1680 )  ( -4200 ) ( 2800 )    |
\                                          /
```

See also: HilbertMatrix

ToeplitzMatrix — create a Toeplitz matrix

(standard library)

Calling format:

```
ToeplitzMatrix(N)
```

Parameters:

N – an n-dimensional row vector

Description:

The function ToeplitzMatrix calculates the Toeplitz matrix given an n-dimensional row vector. This matrix has the same entries in all diagonal columns, from upper left to lower right.

Examples:

```
In> PrettyForm(ToeplitzMatrix({1,2,3,4,5}))

/                               \
| ( 1 ) ( 2 ) ( 3 ) ( 4 ) ( 5 ) |
|                               |
| ( 2 ) ( 1 ) ( 2 ) ( 3 ) ( 4 ) |
|                               |
| ( 3 ) ( 2 ) ( 1 ) ( 2 ) ( 3 ) |
|                               |
| ( 4 ) ( 3 ) ( 2 ) ( 1 ) ( 2 ) |
|                               |
| ( 5 ) ( 4 ) ( 3 ) ( 2 ) ( 1 ) |
\                               /
```

WronskianMatrix — create the Wronskian matrix

(standard library)

Calling format:

```
WronskianMatrix(func,var)
```

Parameters:

func – an n-dimensional vector of functions
var – a variable to differentiate with respect to

Description:

The function WronskianMatrix calculates the Wronskian matrix of n functions.

The Wronskian matrix is created by putting each function as the first element of each column, and filling in the rest of each column by the $(i-1)$-th derivative, where i is the current row.

The Wronskian matrix is used to verify that the n functions are linearly independent, usually solutions to a differential equation. If the determinant of the Wronskian matrix is zero, then the functions are dependent, otherwise they are independent.

Examples:

```
In> WronskianMatrix({Sin(x),Cos(x),x^4},x);
Out> {{Sin(x),Cos(x),x^4},{Cos(x),-Sin(x),4*x^3},
  {-Sin(x),-Cos(x),12*x^2}};
In> PrettyForm(%)
```

```
/                                                 \
| ( Sin( x ) )      ( Cos( x ) )      /  4 \      |
|                                     \ x  /      |
|                                                 |
| ( Cos( x ) )      ( -( Sin( x ) ) ) /      3 \  |
|                                     \ 4 * x  /  |
|                                                 |
| ( -( Sin( x ) ) ) ( -( Cos( x ) ) ) /       2 \ |
|                                     \ 12 * x  / |
\                                                 /
```

The last element is a linear combination of the first two, so the determinant is zero:

```
In> A:=Determinant( WronskianMatrix( {x^4,x^3,2*x^4
  + 3*x^3},x ) )
Out> x^4*3*x^2*(24*x^2+18*x)-x^4*(8*x^3+9*x^2)*6*x
  +(2*x^4+3*x^3)*4*x^3*6*x-4*x^6*(24*x^2+18*x)+x^3
  *(8*x^3+9*x^2)*12*x^2-(2*x^4+3*x^3)*3*x^2*12*x^2;
In> Simplify(A)
Out> 0;
```

SylvesterMatrix — calculate the Sylvester matrix of two polynomials

(standard library)

Calling format:

```
SylvesterMatrix(poly1,poly2,variable)
```

Parameters:

poly1 – polynomial

poly2 – polynomial

variable – variable to express the matrix for

Description:

The function SylvesterMatrix calculates the Sylvester matrix for a pair of polynomials.

The Sylvester matrix is closely related to the resultant, which is defined as the determinant of the Sylvester matrix. Two polynomials share common roots only if the resultant is zero.

Examples:

```
In> ex1:= x^2+2*x-a
Out> x^2+2*x-a;
In> ex2:= x^2+a*x-4
Out> x^2+a*x-4;
In> A:=SylvesterMatrix(ex1,ex2,x)
Out> {{1,2,-a,0},{0,1,2,-a},
  {1,a,-4,0},{0,1,a,-4}};
In> B:=Determinant(A)
Out> 16-a^2*a- -8*a-4*a+a^2- -2*a^2-16-4*a;
In> Simplify(B)
Out> 3*a^2-a^3;
```

The above example shows that the two polynomials have common zeros if $a = 3$.

See also: Determinant, Simplify, Solve, PSolve

Chapter 19

Operations on polynomials

This chapter contains commands to manipulate polynomials. This includes functions for constructing and evaluating orthogonal polynomials.

Expand — transform a polynomial to an expanded form

(standard library)

Calling format:

```
Expand(expr)
Expand(expr, var)
Expand(expr, varlist)
```

Parameters:

`expr` – a polynomial expression
`var` – a variable
`varlist` – a list of variables

Description:

This command brings a polynomial in expanded form, in which polynomials are represented in the form $c_0 + c_1 x + c_2 x^2 + ... + c_n x^n$. In this form, it is easier to test whether a polynomial is zero, namely by testing whether all coefficients are zero.

If the polynomial "expr" contains only one variable, the first calling sequence can be used. Otherwise, the second form should be used which explicitly mentions that "expr" should be considered as a polynomial in the variable "var". The third calling form can be used for multivariate polynomials. Firstly, the polynomial "expr" is expanded with respect to the first variable in "varlist". Then the coefficients are all expanded with respect to the second variable, and so on.

Examples:

```
In> PrettyPrinter'Set("PrettyForm");

True

In> Expand((1+x)^5);

 5        4         3          2
x  + 5 * x  + 10 * x  + 10 * x  + 5 * x + 1

In> Expand((1+x-y)^2, x);

 2                                 2
x  + 2 * ( 1 - y ) * x + ( 1 - y )

In> Expand((1+x-y)^2, {x,y});

 2                           2
x  + ( -2 * y + 2 ) * x + y  - 2 * y + 1
```

See also: `ExpandBrackets`

Degree — degree of a polynomial

(standard library)

Calling format:

```
Degree(expr)
Degree(expr, var)
```

Parameters:

`expr` – a polynomial
`var` – a variable occurring in "expr"

Description:

This command returns the degree of the polynomial "expr" with respect to the variable "var". The degree is the highest power of "var" occurring in the polynomial. If only one variable occurs in "expr", the first calling sequence can be used. Otherwise the user should use the second form in which the variable is explicitly mentioned.

Examples:

```
In> Degree(x^5+x-1);
Out> 5;
In> Degree(a+b*x^3, a);
Out> 1;
In> Degree(a+b*x^3, x);
Out> 3;
```

See also: `Expand`, `Coef`

Coef — coefficient of a polynomial

(standard library)

Calling format:

```
Coef(expr, var, order)
```

Parameters:

`expr` – a polynomial
`var` – a variable occurring in ”expr”
`order` – integer or list of integers

Description:

This command returns the coefficient of “var” to the power “order” in the polynomial “expr”. The parameter “order” can also be a list of integers, in which case this function returns a list of coefficients.

Examples:

```
In> e := Expand((a+x)^4,x)
Out> x^4+4*a*x^3+(a^2+(2*a)^2+a^2)*x^2+
(a^2*2*a+2*a^3)*x+a^4;
In> Coef(e,a,2)
Out> 6*x^2;
In> Coef(e,a,0 .. 4)
Out> {x^4,4*x^3,6*x^2,4*x,1};
```

See also: `Expand`, `Degree`, `LeadingCoef`

`Content` — content of a univariate polynomial

(standard library)

Calling format:

```
Content(expr)
```

Parameters:

`expr` – univariate polynomial

Description:

This command determines the content of a univariate polynomial. The content is the greatest common divisor of all the terms in the polynomial. Every polynomial can be written as the product of the content with the primitive part.

Examples:

```
In> poly := 2*x^2 + 4*x;
Out> 2*x^2+4*x;
In> c := Content(poly);
Out> 2*x;
In> pp := PrimitivePart(poly);
Out> x+2;
In> Expand(pp*c);
Out> 2*x^2+4*x;
```

See also: `PrimitivePart`, `Gcd`

`PrimitivePart` — primitive part of a univariate polynomial

(standard library)

Calling format:

```
PrimitivePart(expr)
```

Parameters:

`expr` – univariate polynomial

Description:

This command determines the primitive part of a univariate polynomial. The primitive part is what remains after the content (the greatest common divisor of all the terms) is divided out. So the product of the content and the primitive part equals the original polynomial.

Examples:

```
In> poly := 2*x^2 + 4*x;
Out> 2*x^2+4*x;
In> c := Content(poly);
Out> 2*x;
In> pp := PrimitivePart(poly);
Out> x+2;
In> Expand(pp*c);
Out> 2*x^2+4*x;
```

See also: `Content`

`LeadingCoef` — leading coefficient of a polynomial

(standard library)

Calling format:

```
LeadingCoef(poly)
LeadingCoef(poly, var)
```

Parameters:

`poly` – a polynomial
`var` – a variable

Description:

This function returns the leading coefficient of “poly”, regarded as a polynomial in the variable “var”. The leading coefficient is the coefficient of the term of highest degree. If only one variable appears in the expression “poly”, it is obvious that it should be regarded as a polynomial in this variable and the first calling sequence may be used.

Examples:

```
In> poly := 2*x^2 + 4*x;
Out> 2*x^2+4*x;
In> lc := LeadingCoef(poly);
Out> 2;
In> m := Monic(poly);
Out> x^2+2*x;
In> Expand(lc*m);
Out> 2*x^2+4*x;

In> LeadingCoef(2*a^2 + 3*a*b^2 + 5, a);
Out> 2;
In> LeadingCoef(2*a^2 + 3*a*b^2 + 5, b);
Out> 3*a;
```

See also: `Coef`, `Monic`

`Monic` — monic part of a polynomial

(standard library)

Calling format:

```
Monic(poly)
Monic(poly, var)
```

Parameters:

`poly` – a polynomial
`var` – a variable

Description:

This function returns the monic part of "poly", regarded as a polynomial in the variable "var". The monic part of a polynomial is the quotient of this polynomial by its leading coefficient. So the leading coefficient of the monic part is always one. If only one variable appears in the expression "poly", it is obvious that it should be regarded as a polynomial in this variable and the first calling sequence may be used.

Examples:

```
In> poly := 2*x^2 + 4*x;
Out> 2*x^2+4*x;
In> lc := LeadingCoef(poly);
Out> 2;
In> m := Monic(poly);
Out> x^2+2*x;
In> Expand(lc*m);
Out> 2*x^2+4*x;

In> Monic(2*a^2 + 3*a*b^2 + 5, a);
Out> a^2+(a*3*b^2)/2+5/2;
In> Monic(2*a^2 + 3*a*b^2 + 5, b);
Out> b^2+(2*a^2+5)/(3*a);
```

See also: `LeadingCoef`

`SquareFree` — return the square-free part of polynomial

(standard library)

Calling format:

```
SquareFree(p)
```

Parameters:

`p` - a polynomial in `x`

Description:

Given a polynomial

$$p = p_1^{n_1} ... p_m^{n_m}$$

with irreducible polynomials p_i, return the square-free version part (with all the factors having multiplicity 1):

$$p_1 ... p_m$$

Examples:

```
In> Expand((x+1)^5)
Out> x^5+5*x^4+10*x^3+10*x^2+5*x+1;
In> SquareFree(%)
Out> (x+1)/5;
In> Monic(%)
Out> x+1;
```

See also: `FindRealRoots`, `NumRealRoots`, `MinimumBound`, `MaximumBound`, `Factor`

`Div` and `Mod` for polynomials

(standard library)

`Div` and `Mod` are also defined for polynomials.

See also: `Div`, `Mod`

`Horner` — convert a polynomial into the Horner form

(standard library)

Calling format:

```
Horner(expr, var)
```

Parameters:

`expr` – a polynomial in "var"
`var` – a variable

Description:

This command turns the polynomial "expr", considered as a univariate polynomial in "var", into Horner form. A polynomial in normal form is an expression such as

$$c_0 + c_1 x + ... + c_n x^n.$$

If one converts this polynomial into Horner form, one gets the equivalent expression

$$(... (c_n x + c_{n-1}) x + ... + c_1) x + c_0.$$

Both expression are equal, but the latter form gives a more efficient way to evaluate the polynomial as the powers have disappeared.

Examples:

```
In> expr1:=Expand((1+x)^4)
Out> x^4+4*x^3+6*x^2+4*x+1;
In> Horner(expr1,x)
Out> (((x+4)*x+6)*x+4)*x+1;
```

See also: `Expand`, `ExpandBrackets`, `EvaluateHornerScheme`

ExpandBrackets — expand all brackets

(standard library)

Calling format:

```
ExpandBrackets(expr)
```

Parameters:

expr – an expression

Description:

This command tries to expand all the brackets by repeatedly using the distributive laws $a(b+c) = ab + ac$ and $(a+b)c = ac + bc$. It goes further than Expand, in that it expands all brackets.

Examples:

```
In> Expand((a-x)*(b-x),x)
Out> x^2-(b+a)*x+a*b;
In> Expand((a-x)*(b-x),{x,a,b})
Out> x^2-(b+a)*x+b*a;
In> ExpandBrackets((a-x)*(b-x))
Out> a*b-x*b+x^2-a*x;
```

See also: Expand

EvaluateHornerScheme — fast evaluation of polynomials

(standard library)

Calling format:

```
EvaluateHornerScheme(coeffs,x)
```

Parameters:

coeffs – a list of coefficients
x – expression

Description:

This function evaluates a polynomial given as a list of its coefficients, using the Horner scheme. The list of coefficients starts with the 0-th power.

Example:

```
In> EvaluateHornerScheme({a,b,c,d},x)
Out> a+x*(b+x*(c+x*d));
```

See also: Horner

Chapter 20

Special polynomials

OrthoP — Legendre and Jacobi orthogonal polynomials

(standard library)

Calling format:

```
OrthoP(n, x);
OrthoP(n, a, b, x);
```

Parameters:

n – degree of polynomial
x – point to evaluate polynomial at
a, b – parameters for Jacobi polynomial

Description:

The first calling format with two arguments evaluates the Legendre polynomial of degree n at the point x. The second form does the same for the Jacobi polynomial with parameters a and b, which should be both greater than -1.

The Jacobi polynomials are orthogonal with respect to the weight function $(1-x)^a (1+x)^b$ on the interval [-1,1]. They satisfy the recurrence relation

$$P(n,a,b,x) = \frac{2n+a+b-1}{2n+a+b-2} *$$

$$\frac{a^2-b^2+x(2n+a+b-2)(n+a+b)}{2n(n+a+b)} P(n-1,a,b,x)$$

$$-\frac{(n+a-1)(n+b-1)(2n+a+b)}{n(n+a+b)(2n+a+b-2)} P(n-2,a,b,x)$$

for $n>1$, with $P(0,a,b,x)=1$,

$$P(1,a,b,x) = \frac{a-b}{2} + x\left(1+\frac{a+b}{2}\right).$$

Legendre polynomials are a special case of Jacobi polynomials with the specific parameter values $a=b=0$. So they form an orthogonal system with respect to the weight function identically equal to 1 on the interval [-1,1], and they satisfy the recurrence relation

$$P(n,x) = (2n-1)\frac{x}{2n} P(n-1,x) - \frac{n-1}{n} P(n-2,x)$$

for $n>1$, with $P(0,x)=1$, $P(1,x)=x$.

Most of the work is performed by the internal function OrthoPoly.

Examples:

```
In> PrettyPrinter'Set("PrettyForm");

True

In> OrthoP(3, x);

    /      2     \
    | 5 * x    3 |
x * | ------ - - |
    \   2      2 /

In> OrthoP(3, 1, 2, x);

1       /     / 21 * x   7 \   7 \
- + x * | x * | ------ - - | - - |
2       \     \   2      2 /   2 /

In> Expand(%)

     3        2
21 * x  - 7 * x  - 7 * x + 1
----------------------------
             2

In> OrthoP(3, 1, 2, 0.5);

-0.8124999999
```

See also: OrthoPSum, OrthoG, OrthoPoly

OrthoH — Hermite orthogonal polynomials

(standard library)

Calling format:

```
OrthoH(n, x);
```

Parameters:

n – degree of polynomial
x – point to evaluate polynomial at

Description:

This function evaluates the Hermite polynomial of degree n at the point x.

The Hermite polynomials are orthogonal with respect to the weight function $\exp\left(-\frac{x^2}{2}\right)$ on the entire real axis. They satisfy the recurrence relation

$$H(n,x) = 2xH(n-1,x) - 2(n-1)H(n-2,x)$$

for $n > 1$, with $H(0,x) = 1$, $H(1,x) = 2x$.

Most of the work is performed by the internal function `OrthoPoly`.

Examples:

```
In> OrthoH(3, x);
Out> x*(8*x^2-12);
In> OrthoH(6, 0.5);
Out> 31;
```

See also: `OrthoHSum`, `OrthoPoly`

`OrthoG` — Gegenbauer orthogonal polynomials

(standard library)

Calling format:

```
OrthoG(n, a, x);
```

Parameters:

`n` – degree of polynomial
`a` – parameter
`x` – point to evaluate polynomial at

Description:

This function evaluates the Gegenbauer (or ultraspherical) polynomial with parameter `a` and degree `n` at the point `x`. The parameter `a` should be greater than -1/2.

The Gegenbauer polynomials are orthogonal with respect to the weight function $\left(1-x^2\right)^{a-\frac{1}{2}}$ on the interval [-1,1]. Hence they are connected to the Jacobi polynomials via

$$G(n,a,x) = P\left(n, a-\frac{1}{2}, a-\frac{1}{2}, x\right).$$

They satisfy the recurrence relation

$$G(n,a,x) = 2\left(1+\frac{a-1}{n}\right)xG(n-1,a,x) - \left(1+2\frac{a-2}{n}\right)G(n-2,a,x)$$

for $n > 1$, with $G(0,a,x) = 1$, $G(1,a,x) = 2x$.

Most of the work is performed by the internal function `OrthoPoly`.

Examples:

```
In> OrthoG(5, 1, x);
Out> x*((32*x^2-32)*x^2+6);
In> OrthoG(5, 2, -0.5);
Out> 2;
```

See also: `OrthoP`, `OrthoT`, `OrthoU`, `OrthoGSum`, `OrthoPoly`

`OrthoL` — Laguerre orthogonal polynomials

(standard library)

Calling format:

```
OrthoL(n, a, x);
```

Parameters:

`n` – degree of polynomial
`a` – parameter
`x` – point to evaluate polynomial at

Description:

This function evaluates the Laguerre polynomial with parameter `a` and degree `n` at the point `x`. The parameter `a` should be greater than -1.

The Laguerre polynomials are orthogonal with respect to the weight function $x^a \exp(-x)$ on the positive real axis. They satisfy the recurrence relation

$$L(n,a,x) = \left(2+\frac{a-1-x}{n}\right)L(n-1,a,x) - \left(1-\frac{a-1}{n}\right)L(n-2,a,x)$$

for $n > 1$, with $L(0,a,x) = 1$, $L(1,a,x) = a+1-x$.

Most of the work is performed by the internal function `OrthoPoly`.

Examples:

```
In> OrthoL(3, 1, x);
Out> x*(x*(2-x/6)-6)+4;
In> OrthoL(3, 1/2, 0.25);
Out> 1.2005208334;
```

See also: `OrthoLSum`, `OrthoPoly`

`OrthoT` — Chebyshev polynomials

`OrthoU` — Chebyshev polynomials

(standard library)

Calling format:

```
OrthoT(n, x);
OrthoU(n, x);
```

Parameters:

`n` – degree of polynomial
`x` – point to evaluate polynomial at

Description:

These functions evaluate the Chebyshev polynomials of the first kind $T(n,x)$ and of the second kind $U(n,x)$, of degree "n" at the point "x". (The name of this Russian mathematician is also sometimes spelled "Tschebyscheff".)

The Chebyshev polynomials are orthogonal with respect to the weight function $\left(1-x^2\right)^{-\frac{1}{2}}$. Hence they are a special case of the Gegenbauer polynomials $G(n, a, x)$, with $a = 0$. They satisfy the recurrence relations

$$T(n, x) = 2xT(n-1, x) - T(n-2, x),$$

$$U(n, x) = 2xU(n-1, x) - U(n-2, x)$$

for $n > 1$, with $T(0, x) = 1$, $T(1, x) = x$, $U(0, x) = 1$, $U(1, x) = 2x$.

Examples:

```
In> OrthoT(3, x);
Out> 2*x*(2*x^2-1)-x;
In> OrthoT(10, 0.9);
Out> -0.2007474688;
In> OrthoU(3, x);
Out> 4*x*(2*x^2-1);
In> OrthoU(10, 0.9);
Out> -2.2234571776;
```

See also: `OrthoG`, `OrthoTSum`, `OrthoUSum`, `OrthoPoly`

`OrthoPSum` — sums of series of orthogonal polynomials

`OrthoHSum` — sums of series of orthogonal polynomials

`OrthoLSum` — sums of series of orthogonal polynomials

`OrthoGSum` — sums of series of orthogonal polynomials

`OrthoTSum` — sums of series of orthogonal polynomials

`OrthoUSum` — sums of series of orthogonal polynomials

(standard library)

Calling format:

```
OrthoPSum(c, x);
OrthoPSum(c, a, b, x);
OrthoHSum(c, x);
OrthoLSum(c, a, x);
OrthoGSum(c, a, x);
OrthoTSum(c, x);
OrthoUSum(c, x);
```

Parameters:

`c` – list of coefficients
`a`, `b` – parameters of specific polynomials
`x` – point to evaluate polynomial at

Description:

These functions evaluate the sum of series of orthogonal polynomials at the point `x`, with given list of coefficients `c` of the series and fixed polynomial parameters `a`, `b` (if applicable).

The list of coefficients starts with the lowest order, so that for example OrthoLSum(c, a, x) = c[1] L[0](a,x) + c[2] L[1](a,x) + ... + c[N] L[N-1](a,x).

See pages for specific orthogonal polynomials for more details on the parameters of the polynomials.

Most of the work is performed by the internal function `OrthoPolySum`. The individual polynomials entering the series are not computed, only the sum of the series.

Examples:

```
In> Expand(OrthoPSum({1,0,0,1/7,1/8}, 3/2, \
  2/3, x));
Out> (7068985*x^4)/3981312+(1648577*x^3)/995328+
(-3502049*x^2)/4644864+(-4372969*x)/6967296
+28292143/27869184;
```

See also: `OrthoP`, `OrthoG`, `OrthoH`, `OrthoL`, `OrthoT`, `OrthoU`, `OrthoPolySum`

`OrthoPoly` — internal function for constructing orthogonal polynomials

(standard library)

Calling format:

```
OrthoPoly(name, n, par, x)
```

Parameters:

`name` – string containing name of orthogonal family
`n` – degree of the polynomial
`par` – list of values for the parameters
`x` – point to evaluate at

Description:

This function is used internally to construct orthogonal polynomials. It returns the `n`-th polynomial from the family `name` with parameters `par` at the point `x`.

All known families are stored in the association list returned by the function `KnownOrthoPoly()`. The name serves as key. At the moment the following names are known to Yacas: `"Jacobi"`, `"Gegenbauer"`, `"Laguerre"`, `"Hermite"`, `"Tscheb1"`, and `"Tscheb2"`. The value associated to the key is a pure function that takes two arguments: the order `n` and the extra parameters `p`, and returns a list of two lists: the first list contains the coefficients `A,B` of the n=1 polynomial, i.e. $A + Bx$; the second list contains the coefficients `A,B,C` in the recurrence relation, i.e. $P_n = (A + Bx) P_{n-1} + CP_{n-2}$. (There are only 3 coefficients in the second list, because none of the polynomials use $C + Dx$ instead of C in the recurrence relation. This is assumed in the implementation!)

If the argument `x` is numerical, the function `OrthoPolyNumeric` is called. Otherwise, the function `OrthoPolyCoeffs` computes a list of coefficients, and `EvaluateHornerScheme` converts this list into a polynomial expression.

See also: `OrthoP`, `OrthoG`, `OrthoH`, `OrthoL`, `OrthoT`, `OrthoU`, `OrthoPolySum`

`OrthoPolySum` — internal function for computing series of orthogonal polynomials

(standard library)

Calling format:

```
OrthoPolySum(name, c, par, x)
```

Parameters:

`name` – string containing name of orthogonal family
`c` – list of coefficients
`par` – list of values for the parameters
`x` – point to evaluate at

Description:

This function is used internally to compute series of orthogonal polynomials. It is similar to the function `OrthoPoly` and returns the result of the summation of series of polynomials from the family `name` with parameters `par` at the point `x`, where `c` is the list of coefficients of the series.

The algorithm used to compute the series without first computing the individual polynomials is the Clenshaw-Smith recurrence scheme. (See the algorithms book for explanations.)

If the argument `x` is numerical, the function `OrthoPolySumNumeric` is called. Otherwise, the function `OrthoPolySumCoeffs` computes the list of coefficients of the resulting polynomial, and `EvaluateHornerScheme` converts this list into a polynomial expression.

See also: `OrthoPSum`, `OrthoGSum`, `OrthoHSum`, `OrthoLSum`, `OrthoTSum`, `OrthoUSum`, `OrthoPoly`

Chapter 21

List operations

Most objects that can be of variable size are represented as lists (linked lists internally). Yacas does implement arrays, which are faster when the number of elements in a collection of objects doesn't change. Operations on lists have better support in the current system.

`Head` — the first element of a list

(Yacas internal)

Calling format:

```
Head(list)
```

Parameters:

`list` – a list

Description:

This function returns the first element of a list. If it is applied to a general expression, it returns the first operand. An error is returned if "list" is an atom.

Examples:

```
In> Head({a,b,c})
Out> a;
In> Head(f(a,b,c));
Out> a;
```

See also: `Tail`, `Length`

`Tail` — returns a list without its first element

(Yacas internal)

Calling format:

```
Tail(list)
```

Parameters:

`list` – a list

Description:

This function returns "list" without its first element.

Examples:

```
In> Tail({a,b,c})
Out> {b,c};
```

See also: `Head`, `Length`

`Length` — the length of a list or string

(Yacas internal)

Calling format:

```
Length(object)
```

Parameters:

`object` – a list, array or string

Description:

Length returns the length of a list. This function also works on strings and arrays.

Examples:

```
In> Length({a,b,c})
Out> 3;
In> Length("abcdef");
Out> 6;
```

See also: `Head`, `Tail`, `Nth`, `Count`

`Map` — apply an n-ary function to all entries in a list

(standard library)

Calling format:

```
Map(fn, list)
```

Parameters:

`fn` – function to apply
`list` – list of lists of arguments

Description:

This function applies "fn" to every list of arguments to be found in "list". So the first entry of "list" should be a list containing the first, second, third, ... argument to "fn", and the same goes for the other entries of "list". The function can either be given as a string or as a pure function (see Apply for more information on pure functions).

Examples:

```
In> MapSingle("Sin",{a,b,c});
Out> {Sin(a),Sin(b),Sin(c)};
In> Map("+",{{a,b},{c,d}});
Out> {a+c,b+d};
```

See also: `MapSingle`, `MapArgs`, `Apply`

MapSingle — apply a unary function to all entries in a list

(standard library)

Calling format:

```
MapSingle(fn, list)
```

Parameters:

fn – function to apply
list – list of arguments

Description:

The function "fn" is successively applied to all entries in "list", and a list containing the respective results is returned. The function can be given either as a string or as a pure function (see Apply for more information on pure functions).

The /@ operator provides a shorthand for MapSingle.

Examples:

```
In> MapSingle("Sin",{a,b,c});
Out> {Sin(a),Sin(b),Sin(c)};
In> MapSingle({{x},x^2}, {a,2,c});
Out> {a^2,4,c^2};
```

See also: Map, MapArgs, /@, Apply

MakeVector — vector of uniquely numbered variable names

(standard library)

Calling format:

```
MakeVector(var,n)
```

Parameters:

var – free variable
n – length of the vector

Description:

A list of length "n" is generated. The first entry contains the identifier "var" with the number 1 appended to it, the second entry contains "var" with the suffix 2, and so on until the last entry which contains "var" with the number "n" appended to it.

Examples:

```
In> MakeVector(a,3)
Out> {a1,a2,a3};
```

See also: RandomIntegerVector, ZeroVector

Select — select entries satisfying some predicate

(standard library)

Calling format:

```
Select(pred, list)
```

Parameters:

pred – a predicate
list – a list of elements to select from

Description:

Select returns a sublist of "list" which contains all the entries for which the predicate "pred" returns True when applied to this entry.

Examples:

```
In> Select("IsInteger",{a,b,2,c,3,d,4,e,f})
Out> {2,3,4};
```

See also: Length, Find, Count

Nth — return the n-th element of a list

(Yacas internal)

Calling format:

```
Nth(list, n)
```

Parameters:

list – list to choose from
n – index of entry to pick

Description:

The entry with index "n" from "list" is returned. The first entry has index 1. It is possible to pick several entries of the list by taking "n" to be a list of indices.

More generally, Nth returns the n-th operand of the expression passed as first argument.

An alternative but equivalent form of Nth(list, n) is list[n].

Examples:

```
In> lst := {a,b,c,13,19};
Out> {a,b,c,13,19};
In> Nth(lst, 3);
Out> c;
In> lst[3];
Out> c;
In> Nth(lst, {3,4,1});
Out> {c,13,a};
In> Nth(b*(a+c), 2);
Out> a+c;
```

See also: Select, Nth

`DestructiveReverse` — reverse a list destructively

(Yacas internal)

Calling format:

```
DestructiveReverse(list)
```

Parameters:

`list` – list to reverse

Description:

This command reverses "list" in place, so that the original is destroyed. This means that any variable bound to "list" will now have an undefined content, and should not be used any more. The reversed list is returned.

Destructive commands are faster than their nondestructive counterparts. `Reverse` is the non-destructive version of this function.

Examples:

```
In> lst := {a,b,c,13,19};
Out> {a,b,c,13,19};
In> revlst := DestructiveReverse(lst);
Out> {19,13,c,b,a};
In> lst;
Out> {a};
```

See also: `FlatCopy`, `Reverse`

`Reverse` — return the reversed list (without touching the original)

(standard library)

Calling format:

```
Reverse(list)
```

Parameters:

`list` – list to reverse

Description:

This function returns a list reversed, without changing the original list. It is similar to `DestructiveReverse`, but safer and slower.

Example:

```
In> lst:={a,b,c,13,19}
Out> {a,b,c,13,19};
In> revlst:=Reverse(lst)
Out> {19,13,c,b,a};
In> lst
Out> {a,b,c,13,19};
```

See also: `FlatCopy`, `DestructiveReverse`

`List` — construct a list

(Yacas internal)

Calling format:

```
List(expr1, expr2, ...)
```

Parameters:

`expr1`, `expr2` – expressions making up the list

Description:

A list is constructed whose first entry is "expr1", the second entry is "expr2", and so on. This command is equivalent to the expression "`expr1, expr2, ...`".

Examples:

```
In> List();
Out> {};
In> List(a,b);
Out> {a,b};
In> List(a,{1,2},d);
Out> {a,{1,2},d};
```

See also: `UnList`, `Listify`

`UnList` — convert a list to a function application

(Yacas internal)

Calling format:

```
UnList(list)
```

Parameters:

`list` – list to be converted

Description:

This command converts a list to a function application. The first entry of "list" is treated as a function atom, and the following entries are the arguments to this function. So the function referred to in the first element of "list" is applied to the other elements.

Note that "list" is evaluated before the function application is formed, but the resulting expression is left unevaluated. The functions `UnList()` and `Hold()` both stop the process of evaluation.

Examples:

```
In> UnList({Cos, x});
Out> Cos(x);
In> UnList({f});
Out> f();
In> UnList({Taylor,x,0,5,Cos(x)});
Out> Taylor(x,0,5)Cos(x);
In> Eval(%);
Out> 1-x^2/2+x^4/24;
```

See also: `List`, `Listify`, `Hold`

Listify — convert a function application to a list

(Yacas internal)

Calling format:

```
Listify(expr)
```

Parameters:

`expr` – expression to be converted

Description:

The parameter "expr" is expected to be a compound object, i.e. not an atom. It is evaluated and then converted to a list. The first entry in the list is the top-level operator in the evaluated expression and the other entries are the arguments to this operator. Finally, the list is returned.

Examples:

```
In> Listify(Cos(x));
Out> {Cos,x};
In> Listify(3*a);
Out> {*,3,a};
```

See also: `List`, `UnList`, `IsAtom`

Concat — concatenate lists

(Yacas internal)

Calling format:

```
Concat(list1, list2, ...)
```

Parameters:

`list1`, `list2`, ... – lists to concatenate

Description:

The lists "list1", "list2", ... are evaluated and concatenated. The resulting big list is returned.

Examples:

```
In> Concat({a,b}, {c,d});
Out> {a,b,c,d};
In> Concat({5}, {a,b,c}, {{f(x)}});
Out> {5,a,b,c,{f(x)}};
```

See also: `ConcatStrings`, `:`, `Insert`

Delete — delete an element from a list

(Yacas internal)

Calling format:

```
Delete(list, n)
```

Parameters:

`list` – list from which an element should be removed
`n` – index of the element to remove

Description:

This command deletes the n-th element from "list". The first parameter should be a list, while "n" should be a positive integer less than or equal to the length of "list". The entry with index "n" is removed (the first entry has index 1), and the resulting list is returned.

Examples:

```
In> Delete({a,b,c,d,e,f}, 4);
Out> {a,b,c,e,f};
```

See also: `DestructiveDelete`, `Insert`, `Replace`

Insert — insert an element into a list

(Yacas internal)

Calling format:

```
Insert(list, n, expr)
```

Parameters:

`list` – list in which "expr" should be inserted
`n` – index at which to insert
`expr` – expression to insert in "list"

Description:

The expression "expr" is inserted just before the n-th entry in "list". The first parameter "list" should be a list, while "n" should be a positive integer less than or equal to the length of "list" plus one. The expression "expr" is placed between the entries in "list" with entries "n-1" and "n". There are two border line cases: if "n" is 1, the expression "expr" is placed in front of the list (just as by the `:` operator); if "n" equals the length of "list" plus one, the expression "expr" is placed at the end of the list (just as by `Append`). In any case, the resulting list is returned.

Examples:

```
In> Insert({a,b,c,d}, 4, x);
Out> {a,b,c,x,d};
In> Insert({a,b,c,d}, 5, x);
Out> {a,b,c,d,x};
In> Insert({a,b,c,d}, 1, x);
Out> {x,a,b,c,d};
```

See also: `DestructiveInsert`, `:`, `Append`, `Delete`, `Remove`

DestructiveDelete — delete an element destructively from a list

(Yacas internal)

Calling format:

```
DestructiveDelete(list, n)
```

Parameters:

`list` – list from which an element should be removed
`n` – index of the element to remove

Description:

This is the destructive counterpart of `Delete`. This command yields the same result as the corresponding call to `Delete`, but the original list is modified. So if a variable is bound to "list", it will now be bound to the list with the n-th entry removed.

Destructive commands run faster than their nondestructive counterparts because the latter copy the list before they alter it.

Examples:

```
In> lst := {a,b,c,d,e,f};
Out> {a,b,c,d,e,f};
In> Delete(lst, 4);
Out> {a,b,c,e,f};
In> lst;
Out> {a,b,c,d,e,f};
In> DestructiveDelete(lst, 4);
Out> {a,b,c,e,f};
In> lst;
Out> {a,b,c,e,f};
```

See also: `Delete`, `DestructiveInsert`, `DestructiveReplace`

`DestructiveInsert` — insert an element destructively into a list

(Yacas internal)

Calling format:

```
DestructiveInsert(list, n, expr)
```

Parameters:

`list` – list in which "expr" should be inserted
`n` – index at which to insert
`expr` – expression to insert in "list"

Description:

This is the destructive counterpart of `Insert`. This command yields the same result as the corresponding call to `Insert`, but the original list is modified. So if a variable is bound to "list", it will now be bound to the list with the expression "expr" inserted.

Destructive commands run faster than their nondestructive counterparts because the latter copy the list before they alter it.

Examples:

```
In> lst := {a,b,c,d};
Out> {a,b,c,d};
In> Insert(lst, 2, x);
Out> {a,x,b,c,d};
In> lst;
Out> {a,b,c,d};
In> DestructiveInsert(lst, 2, x);
Out> {a,x,b,c,d};
In> lst;
Out> {a,x,b,c,d};
```

See also: `Insert`, `DestructiveDelete`, `DestructiveReplace`

`Replace` — replace an entry in a list

(Yacas internal)

Calling format:

```
Replace(list, n, expr)
```

Parameters:

`list` – list of which an entry should be replaced
`n` – index of entry to replace
`expr` – expression to replace the n-th entry with

Description:

The n-th entry of "list" is replaced by the expression "expr". This is equivalent to calling `Delete` and `Insert` in sequence. To be precise, the expression `Replace(list, n, expr)` has the same result as the expression `Insert(Delete(list, n), n, expr)`.

Examples:

```
In> Replace({a,b,c,d,e,f}, 4, x);
Out> {a,b,c,x,e,f};
```

See also: `Delete`, `Insert`, `DestructiveReplace`

`DestructiveReplace` — replace an entry destructively in a list

(Yacas internal)

Calling format:

```
DestructiveReplace(list, n, expr)
```

Parameters:

`list` – list of which an entry should be replaced
`n` – index of entry to replace
`expr` – expression to replace the n-th entry with

Description:

This is the destructive counterpart of `Replace`. This command yields the same result as the corresponding call to `Replace`, but the original list is modified. So if a variable is bound to "list", it will now be bound to the list with the expression "expr" inserted.

Destructive commands run faster than their nondestructive counterparts because the latter copy the list before they alter it.

Examples:

```
In> lst := {a,b,c,d,e,f};
Out> {a,b,c,d,e,f};
In> Replace(lst, 4, x);
Out> {a,b,c,x,e,f};
In> lst;
Out> {a,b,c,d,e,f};
In> DestructiveReplace(lst, 4, x);
Out> {a,b,c,x,e,f};
In> lst;
Out> {a,b,c,x,e,f};
```

See also: `Replace`, `DestructiveDelete`, `DestructiveInsert`

FlatCopy — copy the top level of a list

(Yacas internal)

Calling format:

```
FlatCopy(list)
```

Parameters:

`list` – list to be copied

Description:

A copy of "list" is made and returned. The list is not recursed into, only the first level is copied. This is useful in combination with the destructive commands that actually modify lists in place (for efficiency).

Examples:

The following shows a possible way to define a command that reverses a list nondestructively.

```
In> reverse(l_IsList) <-- DestructiveReverse \
  (FlatCopy(l));
Out> True;
In> lst := {a,b,c,d,e};
Out> {a,b,c,d,e};
In> reverse(lst);
Out> {e,d,c,b,a};
In> lst;
Out> {a,b,c,d,e};
```

Contains — test whether a list contains a certain element

(standard library)

Calling format:

```
Contains(list, expr)
```

Parameters:

`list` – list to examine
`expr` – expression to look for in "list"

Description:

This command tests whether "list" contains the expression "expr" as an entry. It returns `True` if it does and `False` otherwise. Only the top level of "list" is examined. The parameter "list" may also be a general expression, in that case the top-level operands are tested for the occurrence of "expr".

Examples:

```
In> Contains({a,b,c,d}, b);
Out> True;
In> Contains({a,b,c,d}, x);
Out> False;
In> Contains({a,{1,2,3},z}, 1);
Out> False;
In> Contains(a*b, b);
Out> True;
```

See also: `Find`, `Count`

Find — get the index at which a certain element occurs

(standard library)

Calling format:

```
Find(list, expr)
```

Parameters:

`list` – the list to examine
`expr` – expression to look for in "list"

Description:

This commands returns the index at which the expression "expr" occurs in "list". If "expr" occurs more than once, the lowest index is returned. If "expr" does not occur at all, `-1` is returned.

Examples:

```
In> Find({a,b,c,d,e,f}, d);
Out> 4;
In> Find({1,2,3,2,1}, 2);
Out> 2;
In> Find({1,2,3,2,1}, 4);
Out> -1;
```

See also: `Contains`

Append — append an entry at the end of a list

(standard library)

Calling format:

```
Append(list, expr)
```

Parameters:

`list` – list to append "expr" to
`expr` – expression to append to the list

Description:

The expression "expr" is appended at the end of "list" and the resulting list is returned.

Note that due to the underlying data structure, the time it takes to append an entry at the end of a list grows linearly with the length of the list, while the time for prepending an entry at the beginning is constant.

Examples:

```
In> Append({a,b,c,d}, 1);
Out> {a,b,c,d,1};
```

See also: `Concat`, `:`, `DestructiveAppend`

DestructiveAppend — destructively append an entry to a list

(Yacas internal)

Calling format:

```
DestructiveAppend(list, expr)
```

Parameters:

list – list to append "expr" to
expr – expression to append to the list

Description:

This is the destructive counterpart of Append. This command yields the same result as the corresponding call to Append, but the original list is modified. So if a variable is bound to "list", it will now be bound to the list with the expression "expr" inserted.

Destructive commands run faster than their nondestructive counterparts because the latter copy the list before they alter it.

Examples:

```
In> lst := {a,b,c,d};
Out> {a,b,c,d};
In> Append(lst, 1);
Out> {a,b,c,d,1};
In> lst
Out> {a,b,c,d};
In> DestructiveAppend(lst, 1);
Out> {a,b,c,d,1};
In> lst;
Out> {a,b,c,d,1};
```

See also: Concat, :, Append

RemoveDuplicates — remove any duplicates from a list

(standard library)

Calling format:

```
RemoveDuplicates(list)
```

Parameters:

list – list to act on

Description:

This command removes all duplicate elements from a given list and returns the resulting list. To be precise, the second occurrence of any entry is deleted, as are the third, the fourth, etc.

Examples:

```
In> RemoveDuplicates({1,2,3,2,1});
Out> {1,2,3};
In> RemoveDuplicates({a,1,b,1,c,1});
Out> {a,1,b,c};
```

Push — add an element on top of a stack

(standard library)

Calling format:

```
Push(stack, expr)
```

Parameters:

stack – a list (which serves as the stack container)
expr – expression to push on "stack"

Description:

This is part of a simple implementation of a stack, internally represented as a list. This command pushes the expression "expr" on top of the stack, and returns the stack afterwards.

Examples:

```
In> stack := {};
Out> {};
In> Push(stack, x);
Out> {x};
In> Push(stack, x2);
Out> {x2,x};
In> PopFront(stack);
Out> x2;
```

See also: Pop, PopFront, PopBack

Pop — remove an element from a stack

(standard library)

Calling format:

```
Pop(stack, n)
```

Parameters:

stack – a list (which serves as the stack container)
n – index of the element to remove

Description:

This is part of a simple implementation of a stack, internally represented as a list. This command removes the element with index "n" from the stack and returns this element. The top of the stack is represented by the index 1. Invalid indices, for example indices greater than the number of element on the stack, lead to an error.

Examples:

```
In> stack := {};
Out> {};
In> Push(stack, x);
Out> {x};
In> Push(stack, x2);
Out> {x2,x};
In> Push(stack, x3);
Out> {x3,x2,x};
In> Pop(stack, 2);
Out> x2;
In> stack;
Out> {x3,x};
```

See also: Push, PopFront, PopBack

PopFront — remove an element from the top of a stack

(standard library)

Calling format:

```
PopFront(stack)
```

Parameters:

`stack` – a list (which serves as the stack container)

Description:

This is part of a simple implementation of a stack, internally represented as a list. This command removes the element on the top of the stack and returns it. This is the last element that is pushed onto the stack.

Examples:

```
In> stack := {};
Out> {};
In> Push(stack, x);
Out> {x};
In> Push(stack, x2);
Out> {x2,x};
In> Push(stack, x3);
Out> {x3,x2,x};
In> PopFront(stack);
Out> x3;
In> stack;
Out> {x2,x};
```

See also: `Push`, `Pop`, `PopBack`

PopBack — remove an element from the bottom of a stack

(standard library)

Calling format:

```
PopBack(stack)
```

Parameters:

`stack` – a list (which serves as the stack container)

Description:

This is part of a simple implementation of a stack, internally represented as a list. This command removes the element at the bottom of the stack and returns this element. Of course, the stack should not be empty.

Examples:

```
In> stack := {};
Out> {};
In> Push(stack, x);
Out> {x};
In> Push(stack, x2);
Out> {x2,x};
In> Push(stack, x3);
Out> {x3,x2,x};
In> PopBack(stack);
Out> x;
In> stack;
Out> {x3,x2};
```

See also: `Push`, `Pop`, `PopFront`

Swap — swap two elements in a list

(standard library)

Calling format:

```
Swap(list, i1, i2)
```

Parameters:

`list` – the list in which a pair of entries should be swapped
`i1, i2` – indices of the entries in "list" to swap

Description:

This command swaps the pair of entries with entries "i1" and "i2" in "list". So the element at index "i1" ends up at index "i2" and the entry at "i2" is put at index "i1". Both indices should be valid to address elements in the list. Then the updated list is returned.

`Swap()` works also on generic arrays.

Examples:

```
In> lst := {a,b,c,d,e,f};
Out> {a,b,c,d,e,f};
In> Swap(lst, 2, 4);
Out> {a,d,c,b,e,f};
```

See also: `Replace`, `DestructiveReplace`, `Array'Create`

Count — count the number of occurrences of an expression

(standard library)

Calling format:

```
Count(list, expr)
```

Parameters:

`list` – the list to examine
`expr` – expression to look for in "list"

Description:

This command counts the number of times that the expression "expr" occurs in "list" and returns this number.

Examples:

```
In> lst := {a,b,c,b,a};
Out> {a,b,c,b,a};
In> Count(lst, a);
Out> 2;
In> Count(lst, c);
Out> 1;
In> Count(lst, x);
Out> 0;
```

See also: `Length`, `Select`, `Contains`

`Intersection` — return the intersection of two lists

(standard library)

Calling format:

```
Intersection(l1, l2)
```

Parameters:

`l1`, `l2` – two lists

Description:

The intersection of the lists "l1" and "l2" is determined and returned. The intersection contains all elements that occur in both lists. The entries in the result are listed in the same order as in "l1". If an expression occurs multiple times in both "l1" and "l2", then it will occur the same number of times in the result.

Examples:

```
In> Intersection({a,b,c}, {b,c,d});
Out> {b,c};
In> Intersection({a,e,i,o,u}, {f,o,u,r,t,e,e,n});
Out> {e,o,u};
In> Intersection({1,2,2,3,3,3}, {1,1,2,2,3,3});
Out> {1,2,2,3,3};
```

See also: `Union`, `Difference`

`Union` — return the union of two lists

(standard library)

Calling format:

```
Union(l1, l2)
```

Parameters:

`l1`, `l2` – two lists

Description:

The union of the lists "l1" and "l2" is determined and returned. The union contains all elements that occur in one or both of the lists. In the resulting list, any element will occur only once.

Examples:

```
In> Union({a,b,c}, {b,c,d});
Out> {a,b,c,d};
In> Union({a,e,i,o,u}, {f,o,u,r,t,e,e,n});
Out> {a,e,i,o,u,f,r,t,n};
In> Union({1,2,2,3,3,3}, {2,2,3,3,4,4});
Out> {1,2,3,4};
```

See also: `Intersection`, `Difference`

`Difference` — return the difference of two lists

(standard library)

Calling format:

```
Difference(l1, l2)
```

Parameters:

`l1`, `l2` – two lists

Description:

The difference of the lists "l1" and "l2" is determined and returned. The difference contains all elements that occur in "l1" but not in "l2". The order of elements in "l1" is preserved. If a certain expression occurs "n1" times in the first list and "n2" times in the second list, it will occur "n1-n2" times in the result if "n1" is greater than "n2" and not at all otherwise.

Examples:

```
In> Difference({a,b,c}, {b,c,d});
Out> {a};
In> Difference({a,e,i,o,u}, {f,o,u,r,t,e,e,n});
Out> {a,i};
In> Difference({1,2,2,3,3,3}, {2,2,3,4,4});
Out> {1,3,3};
```

See also: `Intersection`, `Union`

`FillList` — fill a list with a certain expression

(standard library)

Calling format:

```
FillList(expr, n)
```

Parameters:

`expr` – expression to fill the list with
`n` – the length of the list to construct

Description:

This command creates a list of length "n" in which all slots contain the expression "expr" and returns this list.

Examples:

```
In> FillList(x, 5);
Out> {x,x,x,x,x};
```

See also: `MakeVector`, `ZeroVector`, `RandomIntegerVector`

`Drop` — drop a range of elements from a list

(standard library)

Calling format:

```
Drop(list, n)
Drop(list, -n)
Drop(list, {m,n})
```

Parameters:

`list` – list to act on
`n`, `m` – positive integers describing the entries to drop

Description:

This command removes a sublist of “list” and returns a list containing the remaining entries. The first calling sequence drops the first “n” entries in “list”. The second form drops the last “n” entries. The last invocation drops the elements with indices “m” through “n”.

Examples:

```
In> lst := {a,b,c,d,e,f,g};
Out> {a,b,c,d,e,f,g};
In> Drop(lst, 2);
Out> {c,d,e,f,g};
In> Drop(lst, -3);
Out> {a,b,c,d};
In> Drop(lst, {2,4});
Out> {a,e,f,g};
```

See also: `Take`, `Select`, `Remove`

`Take` — take a sublist from a list, dropping the rest

(standard library)

Calling format:

```
Take(list, n)
Take(list, -n)
Take(list, {m,n})
```

Parameters:

`list` – list to act on
`n`, `m` – positive integers describing the entries to take

Description:

This command takes a sublist of “list”, drops the rest, and returns the selected sublist. The first calling sequence selects the first “n” entries in “list”. The second form takes the last “n” entries. The last invocation selects the sublist beginning with entry number “m” and ending with the “n”-th entry.

Examples:

```
In> lst := {a,b,c,d,e,f,g};
Out> {a,b,c,d,e,f,g};
In> Take(lst, 2);
Out> {a,b};
In> Take(lst, -3);
Out> {e,f,g};
In> Take(lst, {2,4});
Out> {b,c,d};
```

See also: `Drop`, `Select`, `Remove`

`Partition` — partition a list in sublists of equal length

(standard library)

Calling format:

```
Partition(list, n)
```

Parameters:

`list` – list to partition
`n` – length of partitions

Description:

This command partitions “list” into non-overlapping sublists of length “n” and returns a list of these sublists. The first “n” entries in “list” form the first partition, the entries from position “n+1” up to “2n” form the second partition, and so on. If “n” does not divide the length of “list”, the remaining entries will be thrown away. If “n” equals zero, an empty list is returned.

Examples:

```
In> Partition({a,b,c,d,e,f,}, 2);
Out> {{a,b},{c,d},{e,f}};
In> Partition(1 .. 11, 3);
Out> {{1,2,3},{4,5,6},{7,8,9}};
```

See also: `Take`, `Permutations`

`Assoc` — return element stored in association list

(standard library)

Calling format:

```
Assoc(key, alist)
```

Parameters:

`key` – string, key under which element is stored
`alist` – association list to examine

Description:

The association list “alist” is searched for an entry stored with index “key”. If such an entry is found, it is returned. Otherwise the atom `Empty` is returned.

Association lists are represented as a list of two-entry lists. The first element in the two-entry list is the key, the second element is the value stored under this key.

The call `Assoc(key, alist)` can (probably more intuitively) be accessed as `alist[key]`.

Examples:

```
In> writer := {};
Out> {};
In> writer["Iliad"] := "Homer";
Out> True;
In> writer["Henry IV"] := "Shakespeare";
Out> True;
In> writer["Ulysses"] := "James Joyce";
Out> True;
In> Assoc("Henry IV", writer);
Out> {"Henry IV","Shakespeare"};
In> Assoc("War and Peace", writer);
Out> Empty;
```

See also: `AssocIndices`, `[]`, `:=`, `AssocDelete`

AssocIndices — return the keys in an association list

(standard library)

Calling format:

```
AssocIndices(alist)
```

Parameters:

alist – association list to examine

Description:

All the keys in the association list "alist" are assembled in a list and this list is returned.

Examples:

```
In> writer := {};
Out> {};
In> writer["Iliad"] := "Homer";
Out> True;
In> writer["Henry IV"] := "Shakespeare";
Out> True;
In> writer["Ulysses"] := "James Joyce";
Out> True;
In> AssocIndices(writer);
Out> {"Iliad","Henry IV","Ulysses"};
```

See also: Assoc, AssocDelete

AssocDelete — delete an entry in an association list

(standard library)

Calling format:

```
AssocDelete(alist, "key")
AssocDelete(alist, {key, value})
```

Parameters:

alist – association list
"key" – string, association key
value – value of the key to be deleted

Description:

The key "key" in the association list alist is deleted. (The list itself is modified.) If the key was found and successfully deleted, returns True, otherwise if the given key was not found, the function returns False.

The second, longer form of the function deletes the entry that has both the specified key and the specified value. It can be used for two purposes:

1. to make sure that we are deleting the right value;
2. if several values are stored on the same key, to delete the specified entry (see the last example).

At most one entry is deleted.

Examples:

```
In> writer := {};
Out> {};
In> writer["Iliad"] := "Homer";
Out> True;
In> writer["Henry IV"] := "Shakespeare";
Out> True;
In> writer["Ulysses"] := "James Joyce";
Out> True;
In> AssocDelete(writer, "Henry IV")
Out> True;
In> AssocDelete(writer, "Henry XII")
Out> False;
In> writer
Out> {{"Ulysses","James Joyce"},
  {"Iliad","Homer"}};
In> DestructiveAppend(writer,
  {"Ulysses", "Dublin"});
Out> {{"Iliad","Homer"},{"Ulysses","James Joyce"},
  {"Ulysses","Dublin"}};
In> writer["Ulysses"];
Out> "James Joyce";
In> AssocDelete(writer,{"Ulysses","James Joyce"});
Out> True;
In> writer
Out> {{"Iliad","Homer"},{"Ulysses","Dublin"}};
```

See also: Assoc, AssocIndices

Flatten — flatten expression w.r.t. some operator

(standard library)

Calling format:

```
Flatten(expression,operator)
```

Parameters:

expression – an expression
operator – string with the contents of an infix operator.

Description:

Flatten flattens an expression with respect to a specific operator, converting the result into a list. This is useful for unnesting an expression. Flatten is typically used in simple simplification schemes.

Examples:

```
In> Flatten(a+b*c+d,"+");
Out> {a,b*c,d};
In> Flatten({a,{b,c},d},"List");
Out> {a,b,c,d};
```

See also: UnFlatten

UnFlatten — inverse operation of Flatten

(standard library)

Calling format:

```
UnFlatten(list,operator,identity)
```

Parameters:

`list` – list of objects the operator is to work on
`operator` – infix operator
`identity` – identity of the operator

Description:

UnFlatten is the inverse operation of Flatten. Given a list, it can be turned into an expression representing for instance the addition of these elements by calling UnFlatten with "+" as argument to operator, and 0 as argument to identity (0 is the identity for addition, since a+0=a). For multiplication the identity element would be 1.

Examples:

```
In> UnFlatten({a,b,c},"+",0)
Out> a+b+c;
In> UnFlatten({a,b,c},"*",1)
Out> a*b*c;
```

See also: `Flatten`

`Type` — return the type of an expression

(Yacas internal)

Calling format:

```
Type(expr)
```

Parameters:

`expr` – expression to examine

Description:

The type of the expression "expr" is represented as a string and returned. So, if "expr" is a list, the string `"List"` is returned. In general, the top-level operator of "expr" is returned. If the argument "expr" is an atom, the result is the empty string `""`.

Examples:

```
In> Type({a,b,c});
Out> "List";
In> Type(a*(b+c));
Out> "*";
In> Type(123);
Out> "";
```

See also: `IsAtom`, `NrArgs`

`NrArgs` — return number of top-level arguments

(standard library)

Calling format:

```
NrArgs(expr)
```

Parameters:

`expr` – expression to examine

Description:

This function evaluates to the number of top-level arguments of the expression "expr". The argument "expr" may not be an atom, since that would lead to an error.

Examples:

```
In> NrArgs(f(a,b,c))
Out> 3;
In> NrArgs(Sin(x));
Out> 1;
In> NrArgs(a*(b+c));
Out> 2;
```

See also: `Type`, `Length`

`VarList` — list of variables appearing in an expression

`VarListArith` — list of variables appearing in an expression

`VarListSome` — list of variables appearing in an expression

(standard library)

Calling format:

```
VarList(expr)
VarListArith(expr)
VarListSome(expr, list)
```

Parameters:

`expr` – an expression
`list` – a list of function atoms

Description:

The command `VarList(expr)` returns a list of all variables that appear in the expression `expr`. The expression is traversed recursively.

The command `VarListSome` looks only at arguments of functions in the `list`. All other functions are considered "opaque" (as if they do not contain any variables) and their arguments are not checked. For example, `VarListSome(a + Sin(b-c))` will return {`a, b, c`}, but `VarListSome(a*Sin(b-c), {*})` will not look at arguments of `Sin()` and will return {`a,Sin(b-c)`}. Here `Sin(b-c)` is considered a "variable" because the function `Sin` does not belong to `list`.

The command `VarListArith` returns a list of all variables that appear arithmetically in the expression `expr`. This is implemented through `VarListSome` by restricting to the arithmetic functions `+`, `-`, `*`, `/`. Arguments of other functions are not checked.

Note that since the operators "`+`" and "`-`" are prefix as well as infix operators, it is currently required to use `Atom("+")` to obtain the unevaluated atom "`+`".

Examples:

```
In> VarList(Sin(x))
Out> {x};
In> VarList(x+a*y)
Out> {x,a,y};
In> VarListSome(x+a*y, {Atom("+")})
Out> {x,a*y};
In> VarListArith(x+y*Cos(Ln(x)/x))
Out> {x,y,Cos(Ln(x)/x)}
In> VarListArith(x+a*y^2-1)
Out> {x,a,y^2};
```

See also: `IsFreeOf`, `IsVariable`, `FuncList`, `HasExpr`, `HasFunc`

`FuncList` — list of functions used in an expression

`FuncListArith` — list of functions used in an expression

`FuncListSome` — list of functions used in an expression

(standard library)

Calling format:

```
FuncList(expr)
FuncListArith(expr)
FuncListSome(expr, list)
```

Parameters:

`expr` – an expression
`list` – list of function atoms to be considered "transparent"

Description:

The command `FuncList(expr)` returns a list of all function atoms that appear in the expression `expr`. The expression is recursively traversed.

The command `FuncListSome(expr, list)` does the same, except it only looks at arguments of a given `list` of functions. All other functions become "opaque" (as if they do not contain any other functions). For example, `FuncListSome(a + Sin(b-c))` will see that the expression has a "`-`" operation and return `{+,Sin,-}`, but `FuncListSome(a + Sin(b-c), {+})` will not look at arguments of `Sin()` and will return `{+,Sin}`.

`FuncListArith` is defined through `FuncListSome` to look only at arithmetic operations `+`, `-`, `*`, `/`.

Note that since the operators "`+`" and "`-`" are prefix as well as infix operators, it is currently required to use `Atom("+")` to obtain the unevaluated atom "`+`".

Examples:

```
In> FuncList(x+y*Cos(Ln(x)/x))
Out> {+,*,Cos,/,Ln};
In> FuncListArith(x+y*Cos(Ln(x)/x))
Out> {+,*,Cos};
In> FuncListSome({a+b*2,c/d},{List})
Out> {List,+,/};
```

See also: `VarList`, `HasExpr`, `HasFunc`

`BubbleSort` — sort a list

`HeapSort` — sort a list

(standard library)

Calling format:

```
BubbleSort(list, compare)
HeapSort(list, compare)
```

Parameters:

`list` – list to sort
`compare` – function used to compare elements of `list`

Description:

This command returns `list` after it is sorted using `compare` to compare elements. The function `compare` should accept two arguments, which will be elements of `list`, and compare them. It should return `True` if in the sorted list the second argument should come after the first one, and `False` otherwise.

The function `BubbleSort` uses the so-called "bubble sort" algorithm to do the sorting by swapping elements that are out of order. This algorithm is easy to implement, though it is not particularly fast. The sorting time is proportional to n^2 where n is the length of the list.

The function `HeapSort` uses a recursive algorithm "heapsort" and is much faster for large lists. The sorting time is proportional to $n \ln n$ where n is the length of the list.

Examples:

```
In> BubbleSort({4,7,23,53,-2,1}, "<");
Out> {-2,1,4,7,23,53};
In> HeapSort({4,7,23,53,-2,1}, ">");
Out> {53,23,7,4,1,-2};
```

`PrintList` — print list with padding

(standard library)

Calling format:

```
PrintList(list)
PrintList(list, padding);
```

Parameters:

`list` – a list to be printed
`padding` – (optional) a string

Description:

Prints `list` and inserts the `padding` string between each pair of items of the list. Items of the list which are strings are printed without quotes, unlike `Write()`. Items of the list which are themselves lists are printed inside braces `{}`. If padding is not specified, a standard one is used (comma, space).

Examples:

```
In> PrintList({a,b,{c, d}}, " .. ")
Out> " a ..  b .. { c ..  d}";
```

See also: `Write`, `WriteString`

Table — evaluate while some variable ranges over interval

(standard library)

Calling format:

```
Table(body, var, from, to, step)
```

Parameters:

`body` – expression to evaluate multiple times
`var` – variable to use as loop variable
`from` – initial value for "var"
`to` – final value for "var"
`step` – step size with which "var" is incremented

Description:

This command generates a list of values from "body", by assigning variable "var" values from "from" up to "to", incrementing "step" each time. So, the variable "var" first gets the value "from", and the expression "body" is evaluated. Then the value "from"+"step" is assigned to "var" and the expression "body" is again evaluated. This continues, incrementing "var" with "step" on every iteration, until "var" exceeds "to". At that moment, all the results are assembled in a list and this list is returned.

Examples:

```
In> Table(i!, i, 1, 9, 1);
Out> {1,2,6,24,120,720,5040,40320,362880};
In> Table(i, i, 3, 16, 4);
Out> {3,7,11,15};
In> Table(i^2, i, 10, 1, -1);
Out> {100,81,64,49,36,25,16,9,4,1};
```

See also: `For`, `MapSingle`, `..`, `TableForm`

TableForm — print each entry in a list on a line

(standard library)

Calling format:

```
TableForm(list)
```

Parameters:

`list` – list to print

Description:

This functions writes out the list `list` in a better readable form, by printing every element in the list on a separate line.

Examples:

```
In> TableForm(Table(i!, i, 1, 10, 1));
1
  2
  6
  24
  120
  720
  5040
  40320
  362880
  3628800
Out> True;
```

See also: `PrettyForm`, `Echo`, `Table`

GlobalPop — restore variables using a global stack

GlobalPush — save variables using a global stack

(standard library)

Calling format:

```
GlobalPop(var)
GlobalPop()
GlobalPush(expr)
```

Parameters:

`var` – atom, name of variable to restore from the stack
`expr` – expression, value to save on the stack

Description:

These functions operate with a global stack, currently implemented as a list that is not accessible externally (it is protected through `LocalSymbols`).

`GlobalPush` stores a value on the stack. `GlobalPop` removes the last pushed value from the stack. If a variable name is given, the variable is assigned, otherwise the popped value is returned.

If the global stack is empty, an error message is printed.

Examples:

```
In> GlobalPush(3)
Out> 3;
In> GlobalPush(Sin(x))
Out> Sin(x);
In> GlobalPop(x)
Out> Sin(x);
In> GlobalPop(x)
Out> 3;
In> x
Out> 3;
```

See also: `Push`, `PopFront`

Chapter 22

Functional operators

These operators can help the user to program in the style of functional programming languages such as Miranda or Haskell.

: — prepend item to list, or concatenate strings

(standard library)

Calling format:

```
item : list
string1 : string2
```

Precedence: 70

Parameters:

`item` – an item to be prepended to a list
`list` – a list
`string1` – a string
`string2` – a string

Description:

The first form prepends "item" as the first entry to the list "list". The second form concatenates the strings "string1" and "string2".

Examples:

```
In> a:b:c:{}
Out> {a,b,c};
In> "This":"Is":"A":"String"
Out> "ThisIsAString";
```

See also: `Concat`, `ConcatStrings`

@ — apply a function

(standard library)

Calling format:

```
fn @ arglist
```

Precedence: 600

Parameters:

`fn` – function to apply
`arglist` – single argument, or a list of arguments

Description:

This function is a shorthand for `Apply`. It applies the function "fn" to the argument(s) in "arglist" and returns the result. The first parameter "fn" can either be a string containing the name of a function or a pure function.

Examples:

```
In> "Sin" @ a
Out> Sin(a);
In> {{a},Sin(a)} @ a
Out> Sin(a);
In> "f" @ {a,b}
Out> f(a,b);
```

See also: `Apply`

/@ — apply a function to all entries in a list

(standard library)

Calling format:

```
fn /@ list
```

Precedence: 600

Parameters:

`fn` – function to apply
`list` – list of arguments

Description:

This function is a shorthand for `MapSingle`. It successively applies the function "fn" to all the entries in "list" and returns a list contains the results. The parameter "fn" can either be a string containing the name of a function or a pure function.

Examples:

```
In> "Sin" /@ {a,b}
Out> {Sin(a),Sin(b)};
In> {{a},Sin(a)*a} /@ {a,b}
Out> {Sin(a)*a,Sin(b)*b};
```

See also: `MapSingle`, `Map`, `MapArgs`

`..` — construct a list of consecutive integers

(standard library)

Calling format:

```
n .. m
```

Precedence: 600

Parameters:

`n` – integer. the first entry in the list
`m` – integer, the last entry in the list

Description:

This command returns the list {`n, n+1, n+2, ..., m`}. If `m` is smaller than `n`, the empty list is returned. Note that the `..` operator should be surrounded by spaces to keep the parser happy, if "n" is a number. So one should write "`1 .. 4`" instead of "`1..4`".

Example:

```
In> 1 .. 4
Out> {1,2,3,4};
```

See also: `Table`

`NFunction` — make wrapper for numeric functions

(standard library)

Calling format:

```
NFunction("newname","funcname", {arglist})
```

Parameters:

`"newname"` – name of new function
`"funcname"` – name of an existing function
`arglist` – symbolic list of arguments

Description:

This function will define a function named "newname" with the same arguments as an existing function named "funcname". The new function will evaluate and return the expression "funcname(arglist)" only when all items in the argument list `arglist` are numbers, and return unevaluated otherwise.

This can be useful when plotting functions defined through other Yacas routines that cannot return unevaluated.

If the numerical calculation does not return a number (for example, it might return the atom `nan`, "not a number", for some arguments), then the new function will return `Undefined`.

Examples:

```
In> f(x) := N(Sin(x));
Out> True;
In> NFunction("f1", "f", {x});
Out> True;
In> f1(a);
Out> f1(a);
In> f1(0);
Out> 0;
```

Suppose we need to define a complicated function `t(x)` which cannot be evaluated unless `x` is a number:

```
In> t(x) := If(x<=0.5, 2*x, 2*(1-x));
Out> True;
In> t(0.2);
Out> 0.4;
In> t(x);
In function "If" :
bad argument number 1 (counting from 1)
CommandLine(1) : Invalid argument
```

Then, we can use `NFunction()` to define a wrapper `t1(x)` around `t(x)` which will not try to evaluate `t(x)` unless `x` is a number.

```
In> NFunction("t1", "t", {x})
Out> True;
In> t1(x);
Out> t1(x);
In> t1(0.2);
Out> 0.4;
```

Now we can plot the function.

```
In> Plot2D(t1(x), -0.1: 1.1)
Out> True;
```

See also: `MacroRule`

`Where` — substitute result into expression

(standard library)

Calling format:

```
expr Where x==v
expr Where x1==v1 And x2==v2 And ...
expr Where {x1==v1 And x2==v2,x1==v3
  And x2==v4,...}
```

Parameters:

`expr` - expression to evaluate
`x` - variable to set
`v` - value to substitute for variable

Description:

The operator `Where` fills in values for variables, in its simplest form. It accepts sets of variable/value pairs defined as

```
var1==val1 And var2==val2 And ...
```

and fills in the corresponding values. Lists of value pairs are also possible, as:

```
{var1==val1 And var2==val2, var1==val3
  And var2==val4}
```

These values might be obtained through `Solve`.

Examples:

```
In> x^2+y^2 Where x==2
Out> y^2+4;
In> x^2+y^2 Where x==2 And y==3
Out> 13;
In> x^2+y^2 Where {x==2 And y==3}
Out> {13};
In> x^2+y^2 Where {x==2 And y==3,x==4 And y==5}
Out> {13,41};
```

See also: `Solve`, `AddTo`

`AddTo` — add an equation to a set of equations or set of set of equations

(standard library)

Calling format:

```
eq1 AddTo eq2
```

Parameters:

`eq` - (set of) set of equations

Description:

Given two (sets of) sets of equations, the command AddTo combines multiple sets of equations into one.

A list `a,b` means that a is a solution, OR b is a solution. AddTo then acts as a AND operation:

```
(a or b) and (c or d) =>
(a or b) Addto (c or d) =>
(a and c) or (a and d) or (b and c)
  or (b and d)
```

This function is useful for adding an identity to an already existing set of equations. Suppose a solve command returned `a>=0 And x==a,a<0 And x== -a` from an expression x==Abs(a), then a new identity a==2 could be added as follows:

```
In> a==2 AddTo {a>=0 And x==a,a<0 And x== -a}
Out> {a==2 And a>=0 And x==a,a==2 And a<0
  And x== -a};
```

Passing this set of set of identities back to solve, solve should recognize that the second one is not a possibility any more, since a==2 And a¡0 can never be true at the same time.

Examples:

```
In> {A==2,c==d} AddTo {b==3 And d==2}
Out> {A==2 And b==3 And d==2,c==d
  And b==3 And d==2};
In> {A==2,c==d} AddTo {b==3, d==2}
Out> {A==2 And b==3,A==2 And d==2,c==d
  And b==3,c==d And d==2};
```

See also: `Where`, `Solve`

Chapter 23

Control flow functions

MaxEvalDepth — set the maximum evaluation depth

(Yacas internal)

Calling format:

```
MaxEvalDepth(n)
```

Parameters:

`n` – new maximum evaluation depth

Description:

Use this command to set the maximum evaluation depth to the integer "n". The default value is 1000. The function `MaxEvalDepth` returns `True`.

The point of having a maximum evaluation depth is to catch any infinite recursion. For example, after the definition `f(x) := f(x)`, evaluating the expression `f(x)` would call `f(x)`, which would call `f(x)`, etc. The interpreter will halt if the maximum evaluation depth is reached. Also indirect recursion, e.g. the pair of definitions `f(x) := g(x)` and `g(x) := f(x)`, will be caught.

Examples:

An example of an infinite recursion, caught because the maximum evaluation depth is reached.

```
In> f(x) := f(x)
Out> True;
In> f(x)
Error on line 1 in file [CommandLine]
Max evaluation stack depth reached.
Please use MaxEvalDepth to increase the stack
size as needed.
```

However, a long calculation may cause the maximum evaluation depth to be reached without the presence of infinite recursion. The function `MaxEvalDepth` is meant for these cases.

```
In> 10 # g(0) <-- 1;
Out> True;
In> 20 # g(n_IsPositiveInteger) <-- \
  2 * g(n-1);
Out> True;
In> g(1001);
Error on line 1 in file [CommandLine]
Max evaluation stack depth reached.
Please use MaxEvalDepth to increase the stack
size as needed.
```

```
In> MaxEvalDepth(10000);
Out> True;
In> g(1001);
Out> 21430172143725346418968500981200036211228096234
1106721488750077674070210224987224498639675763139171
6255189345835106293650374290571384628087196915514939
7149607869135549648461970842149210124742283755908364
3060929499671638825347975351183310878921541258291423
92955373084335320859663305248773674411336138752;
```

Hold — keep expression unevaluated

(Yacas internal)

Calling format:

```
Hold(expr)
```

Parameters:

`expr` – expression to keep unevaluated

Description:

The expression "expr" is returned unevaluated. This is useful to prevent the evaluation of a certain expression in a context in which evaluation normally takes place.

The function `UnList()` also leaves its result unevaluated. Both functions stop the process of evaluation (no more rules will be applied).

Examples:

```
In> Echo({ Hold(1+1), "=", 1+1 });
 1+1 = 2
Out> True;
```

See also: `Eval`, `HoldArg`, `UnList`

Eval — force evaluation of expression

(Yacas internal)

Calling format:

```
Eval(expr)
```

Parameters:

`expr` – expression to evaluate

Description:

This function explicitly requests an evaluation of the expression “expr”, and returns the result of this evaluation.

Examples:

```
In> a := x;
Out> x;
In> x := 5;
Out> 5;
In> a;
Out> x;
In> Eval(a);
Out> 5;
```

The variable `a` is bound to `x`, and `x` is bound to 5. Hence evaluating `a` will give `x`. Only when an extra evaluation of `a` is requested, the value 5 is returned.

Note that the behavior would be different if we had exchanged the assignments. If the assignment `a := x` were given while `x` had the value 5, the variable `a` would also get the value 5 because the assignment operator `:=` evaluates the right-hand side.

See also: `Hold`, `HoldArg`, `:=`

`While` — loop while a condition is met

(Yacas internal)

Calling format:

```
While(pred) body
```

Parameters:

`pred` – predicate deciding whether to keep on looping
`body` – expression to loop over

Description:

Keep on evaluating “body” while “pred” evaluates to `True`. More precisely, `While` evaluates the predicate “pred”, which should evaluate to either `True` or `False`. If the result is `True`, the expression ”body” is evaluated and then the predicate “pred” is again evaluated. If it is still `True`, the expressions ”body” and ”pred” are again evaluated and so on until “pred” evaluates to `False`. At that point, the loop terminates and `While` returns `True`.

In particular, if “pred” immediately evaluates to `False`, the body is never executed. `While` is the fundamental looping construct on which all other loop commands are based. It is equivalent to the `while` command in the programming language C.

Examples:

```
In> x := 0;
Out> 0;
In> While (x! < 10^6) \
  [ Echo({x, x!}); x++; ];
 0  1
 1  1
 2  2
 3  6
 4  24
 5  120
 6  720
 7  5040
 8  40320
 9  362880
Out> True;
```

See also: `Until`, `For`

`Until` — loop until a condition is met

(standard library)

Calling format:

```
Until(pred) body
```

Parameters:

`pred` – predicate deciding whether to stop
`body` – expression to loop over

Description:

Keep on evaluating “body” until “pred” becomes `True`. More precisely, `Until` first evaluates the expression “body”. Then the predicate “pred” is evaluated, which should yield either `True` or `False`. In the latter case, the expressions ”body” and “pred” are again evaluated and this continues as long as “pred” is `False`. As soon as ”pred” yields `True`, the loop terminates and `Until` returns `True`.

The main difference with `While` is that `Until` always evaluates the body at least once, but `While` may not evaluate the body at all. Besides, the meaning of the predicate is reversed: `While` stops if “pred” is `False` while `Until` stops if ”pred” is `True`. The command `Until(pred) body;` is equivalent to `pred; While(Not pred) body;`. In fact, the implementation of `Until` is based on the internal command `While`. The `Until` command can be compared to the `do ... while` construct in the programming language C.

Examples:

```
In> x := 0;
Out> 0;
In> Until (x! > 10^6) \
  [ Echo({x, x!}); x++; ];
 0  1
 1  1
 2  2
 3  6
 4  24
 5  120
 6  720
 7  5040
 8  40320
 9  362880
Out> True;
```

See also: `While`, `For`

`If` — branch point

(Yacas internal)

Calling format:

```
If(pred, then)
If(pred, then, else)
```

Parameters:

`pred` – predicate to test
`then` – expression to evaluate if "pred" is `True`
`else` – expression to evaluate if "pred" is `False`

Description:

This command implements a branch point. The predicate "pred" is evaluated, which should result in either `True` or `False`. In the first case, the expression "then" is evaluated and returned. If the predicate yields `False`, the expression "else" (if present) is evaluated and returned. If there is no "else" branch (i.e. if the first calling sequence is used), the `If` expression returns `False`.

Examples:

The sign function is defined to be 1 if its argument is positive and -1 if its argument is negative. A possible implementation is

```
In> mysign(x) := If (IsPositiveReal(x), 1, -1);
Out> True;
In> mysign(Pi);
Out> 1;
In> mysign(-2.5);
Out> -1;
```

Note that this will give incorrect results, if "x" cannot be numerically approximated.

```
In> mysign(a);
Out> -1;
```

Hence a better implementation would be

```
In> mysign(_x)_IsNumber(N(x)) <-- If \
  (IsPositiveReal(x), 1, -1);
Out> True;
```

SystemCall — pass a command to the shell

(Yacas internal)

Calling format:

```
SystemCall(str)
```

Parameters:

`str` – string containing the command to call

Description:

The command contained in the string "str" is executed by the underlying operating system (OS). The return value of `SystemCall` is `True` or `False` according to the exit code of the command.

The `SystemCall` function is not allowed in the body of the `Secure` command and will lead to an error.

Examples:

In a UNIX environment, the command `SystemCall("ls")` would print the contents of the current directory.

```
In> SystemCall("ls")
AUTHORS
COPYING
ChangeLog
```

... (truncated to save space)

```
Out> True;
```

The standard UNIX command `test` returns success or failure depending on conditions. For example, the following command will check if a directory exists:

```
In> SystemCall("test -d scripts/")
Out> True;
```

Check that a file exists:

```
In> SystemCall("test -f COPYING")
Out> True;
In> SystemCall("test -f nosuchfile.txt")
Out> False;
```

See also: `Secure`

Function — declare or define a function

(standard library)

Calling format:

```
Function() func(arglist)
Function() func(arglist, ...)
Function("op", {arglist}) body
Function("op", {arglist, ...}) body
```

Parameters:

`func(args)` – function declaration, e.g. `f(x,y)`
`"op"` – string, name of the function
`{arglist}` – list of atoms, formal arguments to the function
`...` – literal ellipsis symbol "`...`" used to denote a variable number of arguments
`body` – expression comprising the body of the function

Description:

This command can be used to define a new function with named arguments.

The number of arguments of the new function and their names are determined by the list `arglist`. If the ellipsis "`...`" follows the last atom in `arglist`, a function with a variable number of arguments is declared (using `RuleBaseListed`). Note that the ellipsis cannot be the only element of `arglist` and *must* be preceded by an atom.

A function with variable number of arguments can take more arguments than elements in `arglist`; in this case, it obtains its last argument as a list containing all extra arguments.

The short form of the `Function` call merely declares a `RuleBase` for the new function but does not define any function body. This is a convenient shorthand for `RuleBase` and `RuleBaseListed`, when definitions of the function are to be supplied by rules. If the new function has been already declared with the same number of arguments (with or without variable arguments), `Function` returns false and does nothing.

The second, longer form of the `Function` call declares a function and also defines a function body. It is equivalent to a single rule such as op(_arg1, _arg2) <-- body. The rule will be declared at precedence 1025. Any previous rules associated with "op" (with the same arity) will be discarded. More complicated functions (with more than one body) can be defined by adding more rules.

Examples:

This will declare a new function with two or more arguments, but define no rules for it. This is equivalent to `RuleBase ("f1", {x, y, ...})`.

```
In> Function() f1(x,y,...);
Out> True;
In> Function() f1(x,y);
Out> False;
```

This defines a function `FirstOf` which returns the first element of a list. Equivalent definitions would be `FirstOf(_list) <-- list[1]` or `FirstOf(list) := list[1]`.

```
In> Function("FirstOf", {list})  list[1];
Out> True;
In> FirstOf({a,b,c});
Out> a;
```

The following function will print all arguments to a string:

```
In> Function("PrintAll",{x, ...}) If(IsList(x),
  PrintList(x), ToString()Write(x));
Out> True;
In> PrintAll(1):
Out> " 1";
In> PrintAll(1,2,3);
Out> " 1 2 3";
```

See also: `TemplateFunction`, `Rule`, `RuleBase`, `RuleBaseListed`, `:=`, `Retract`

`Macro` — declare or define a macro

(standard library)

Calling format:

```
Macro() func(arglist)
Macro() func(arglist, ...)
Macro("op", {arglist}) body
Macro("op", {arglist, ...}) body
```

Parameters:

`func(args)` – function declaration, e.g. `f(x,y)`
`"op"` – string, name of the function
`{arglist}` – list of atoms, formal arguments to the function
`...` – literal ellipsis symbol "`...`" used to denote a variable number of arguments
`body` – expression comprising the body of the function

Description:

This does the same as `Function`, but for macros. One can define a macro easily with this function, in stead of having to use `DefMacroRuleBase`.

Examples:

the following example defines a looping function.

```
In> Macro("myfor",{init,pred,inc,body}) [@init;While(@pred)
Out> True;
In> a:=10
Out> 10;
```

Here this new macro `myfor` is used to loop, using a variable `a` from the calling environment.

```
In> myfor(i:=1,i<10,i++,Echo(a*i))
10
20
30
40
50
60
70
80
90
Out> True;
In> i
Out> 10;
```

See also: `Function`, `DefMacroRuleBase`

`Use` — load a file, but not twice

(Yacas internal)

Calling format:

```
Use(name)
```

Parameters:

`name` – name of the file to load

Description:

If the file "name" has been loaded before, either by an earlier call to `Use` or via the `DefLoad` mechanism, nothing happens. Otherwise all expressions in the file are read and evaluated. `Use` always returns `True`.

The purpose of this function is to make sure that the file will at least have been loaded, but is not loaded twice.

See also: `Load`, `DefLoad`, `DefaultDirectory`

`For` — C-style `for` loop

(standard library)

Calling format:

```
For(init, pred, incr) body
```

Parameters:

`init` – expression for performing the initialization
`pred` – predicate deciding whether to continue the loop
`incr` – expression to increment the counter
`body` – expression to loop over

Description:

This commands implements a C style `for` loop. First of all, the expression "init" is evaluated. Then the predicate "pred" is evaluated, which should return `True` or `False`. Next the loop is executed as long as the predicate yields `True`. One traversal of the loop consists of the subsequent evaluations of "body", "incr", and "pred". Finally, the value `True` is returned.

This command is most often used in a form such as `For(i=1, i<=10, i++) body`, which evaluates `body` with `i` subsequently set to 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.

The expression `For(init, pred, incr) body` is equivalent to `init; While(pred) [body; incr;]`.

Examples:

```
In> For (i:=1, i<=10, i++) Echo({i, i!});
 1  1
 2  2
 3  6
 4  24
 5  120
 6  720
 7  5040
 8  40320
 9  362880
 10  3628800
Out> True;
```

See also: `While`, `Until`, `ForEach`

`ForEach` — loop over all entries in list

(standard library)

Calling format:

```
ForEach(var, list) body
```

Parameters:

`var` – looping variable
`list` – list of values to assign to "var"
`body` – expression to evaluate with different values of "var"

Description:

The expression "body" is evaluated multiple times. The first time, "var" has the value of the first element of "list", then it gets the value of the second element and so on. `ForEach` returns `True`.

Examples:

```
 In> ForEach(i,{2,3,5,7,11}) Echo({i, i!});
 2  2
 3  6
 5  120
 7  5040
 11  39916800
Out> True;
```

See also: `For`

`Apply` — apply a function to arguments

(standard library)

Calling format:

```
Apply(fn, arglist)
```

Parameters:

`fn` – function to apply
`arglist` – list of arguments

Description:

This function applies the function "fn" to the arguments in "arglist" and returns the result. The first parameter "fn" can either be a string containing the name of a function or a pure function. Pure functions, modeled after lambda-expressions, have the form "`varlist,body`", where "varlist" is the list of formal parameters. Upon application, the formal parameters are assigned the values in "arglist" (the second parameter of `Apply`) and the "body" is evaluated.

Another way to define a pure function is with the Lambda construct. Here, in stead of passing in "`varlist,body`", one can pass in "Lambda(varlist,body)". Lambda has the advantage that its arguments are not evaluated (using lists can have undesirable effects because lists are evaluated). Lambda can be used everywhere a pure function is expected, in principle, because the function Apply is the only function dealing with pure functions. So all places where a pure function can be passed in will also accept Lambda.

An shorthand for `Apply` is provided by the `@` operator.

Examples:

```
In> Apply("+", {5,9});
Out> 14;
In> Apply({{x,y}, x-y^2}, {Cos(a), Sin(a)});
Out> Cos(a)-Sin(a)^2;
In>  Apply(Lambda({x,y}, x-y^2), {Cos(a), Sin(a)});
Out> Cos(a)-Sin(a)^2
In>  Lambda({x,y}, x-y^2) @ {Cos(a), Sin(a)}
Out> Cos(a)-Sin(a)^2
```

See also: `Map`, `MapSingle`, `@`

`MapArgs` — apply a function to all top-level arguments

(standard library)

Calling format:

```
MapArgs(expr, fn)
```

Parameters:

`expr` – an expression to work on
`fn` – an operation to perform on each argument

Description:

Every top-level argument in "expr" is substituted by the result of applying "fn" to this argument. Here "fn" can be either the name of a function or a pure function (see Apply for more information on pure functions).

Examples:

```
In> MapArgs(f(x,y,z),"Sin");
Out> f(Sin(x),Sin(y),Sin(z));
In> MapArgs({3,4,5,6}, {{x},x^2});
Out> {9,16,25,36};
```

See also: `MapSingle`, `Map`, `Apply`

`Subst` — perform a substitution

(standard library)

Calling format:

```
Subst(from, to) expr
```

Parameters:

`from` – expression to be substituted
`to` – expression to substitute for "from"
`expr` – expression in which the substitution takes place

Description:

This function substitutes every occurrence of "from" in "expr" by "to". This is a syntactical substitution: only places where "from" occurs as a subexpression are affected.

Examples:

```
In> Subst(x, Sin(y)) x^2+x+1;
Out> Sin(y)^2+Sin(y)+1;
In> Subst(a+b, x) a+b+c;
Out> x+c;
In> Subst(b+c, x) a+b+c;
Out> a+b+c;
```

The explanation for the last result is that the expression `a+b+c` is internally stored as `(a+b)+c`. Hence `a+b` is a subexpression, but `b+c` is not.

See also: `WithValue`, `/:`

`WithValue` — temporary assignment during an evaluation

(standard library)

Calling format:

```
WithValue(var, val, expr)
WithValue({var,...}, {val,...}, expr)
```

Parameters:

`var` – variable to assign to
`val` – value to be assigned to "var"
`expr` – expression to evaluate with "var" equal to "val"

Description:

First, the expression "val" is assigned to the variable "var". Then, the expression "expr" is evaluated and returned. Finally, the assignment is reversed so that the variable "var" has the same value as it had before `WithValue` was evaluated.

The second calling sequence assigns the first element in the list of values to the first element in the list of variables, the second value to the second variable, etc.

Examples:

```
In> WithValue(x, 3, x^2+y^2+1);
Out> y^2+10;
In> WithValue({x,y}, {3,2}, x^2+y^2+1);
Out> 14;
```

See also: `Subst`, `/:`

`/:` — local simplification rules

`/::` — local simplification rules

(standard library)

Calling format:

```
expression /: patterns
expressions /:: patterns
```

Precedence: 20000

Parameters:

`expression` – an expression
`patterns` – a list of patterns

Description:

Sometimes you have an expression, and you want to use specific simplification rules on it that are not done by default. This can be done with the `/:` and the `/::` operators. Suppose we have the expression containing things such as `Ln(a*b)`, and we want to change these into `Ln(a)+Ln(b)`, the easiest way to do this is using the `/:` operator, as follows:

```
In> Sin(x)*Ln(a*b)
Out> Sin(x)*Ln(a*b);
In> % /: { Ln(_x*_y) <- Ln(x)+Ln(y) }
Out> Sin(x)*(Ln(a)+Ln(b));
```

A whole list of simplification rules can be built up in the list, and they will be applied to the expression on the left hand side of `/:` .

The forms the patterns can have are one of:

```
pattern <- replacement
{pattern,replacement}
{pattern,postpredicate,replacement}
```

Note that for these local rules, `<-` should be used instead of `<--` which would be used in a global rule.

The `/:` operator traverses an expression much as `Subst` does, that is, top down, trying to apply the rules from the beginning of the list of rules to the end of the list of rules. If the rules cannot be applied to an expression, it will try subexpressions of that expression and so on.

It might be necessary sometimes to use the `/::` operator, which repeatedly applies the `/:` operator until the result doesn't change any more. Caution is required, since rules can contradict each other, which could result in an infinite loop. To detect this situation, just use /: repeatedly on the expression. The repetitive nature should become apparent.

Examples:

```
In> Sin(u)*Ln(a*b) /: {Ln(_x*_y) <- Ln(x)+Ln(y)}
Out> Sin(u)*(Ln(a)+Ln(b));
In> Sin(u)*Ln(a*b) /:: { a <- 2, b <- 3 }
Out> Sin(u)*Ln(6);
```

See also: Subst

TraceStack — show calling stack after an error occurs

(Yacas internal)

Calling format:

```
TraceStack(expression)
```

Parameters:

expression – an expression to evaluate

Description:

TraceStack shows the calling stack after an error occurred. It shows the last few items on the stack, not to flood the screen. These are usually the only items of interest on the stack. This is probably by far the most useful debugging function in Yacas. It shows the last few things it did just after an error was generated somewhere.

For each stack frame, it shows if the function evaluated was a built-in function or a user-defined function, and for the user-defined function, the number of the rule it is trying whether it was evaluating the pattern matcher of the rule, or the body code of the rule.

This functionality is not offered by default because it slows down the evaluation code.

Examples:

Here is an example of a function calling itself recursively, causing Yacas to flood its stack:

```
In> f(x):=f(Sin(x))
Out> True;
In> TraceStack(f(2))
Debug> 982 :  f (Rule # 0 in body)
Debug> 983 :  f (Rule # 0 in body)
Debug> 984 :  f (Rule # 0 in body)
Debug> 985 :  f (Rule # 0 in body)
Debug> 986 :  f (Rule # 0 in body)
Debug> 987 :  f (Rule # 0 in body)
Debug> 988 :  f (Rule # 0 in body)
Debug> 989 :  f (Rule # 0 in body)
Debug> 990 :  f (Rule # 0 in body)
Debug> 991 :  f (Rule # 0 in body)
Debug> 992 :  f (Rule # 0 in body)
Debug> 993 :  f (Rule # 0 in body)
Debug> 994 :  f (Rule # 0 in body)
Debug> 995 :  f (User function)
Debug> 996 :  Sin (Rule # 0 in pattern)
Debug> 997 :  IsList (Internal function)
Error on line 1 in file [CommandLine]
Max evaluation stack depth reached.
Please use MaxEvalDepth to increase the stack
size as needed.
```

See also: TraceExp, TraceRule

TraceExp — evaluate with tracing enabled

(Yacas internal)

Calling format:

```
TraceExp(expr)
```

Parameters:

expr – expression to trace

Description:

The expression "expr" is evaluated with the tracing facility turned on. This means that every subexpression, which is evaluated, is shown before and after evaluation. Before evaluation, it is shown in the form `TrEnter(x)`, where `x` denotes the subexpression being evaluated. After the evaluation the line `TrLeave(x,y)` is printed, where `y` is the result of the evaluation. The indentation shows the nesting level.

Note that this command usually generates huge amounts of output. A more specific form of tracing (eg. `TraceRule`) is probably more useful for all but very simple expressions.

Examples:

```
In> TraceExp(2+3);
  TrEnter(2+3);
    TrEnter(2);
    TrLeave(2, 2);
    TrEnter(3);
    TrLeave(3, 3);
    TrEnter(IsNumber(x));
      TrEnter(x);
      TrLeave(x, 2);
    TrLeave(IsNumber(x),True);
    TrEnter(IsNumber(y));
      TrEnter(y);
      TrLeave(y, 3);
    TrLeave(IsNumber(y),True);
    TrEnter(True);
    TrLeave(True, True);
    TrEnter(MathAdd(x,y));
      TrEnter(x);
      TrLeave(x, 2);
      TrEnter(y);
      TrLeave(y, 3);
    TrLeave(MathAdd(x,y),5);
  TrLeave(2+3, 5);
Out> 5;
```

See also: TraceStack, TraceRule

TraceRule — turn on tracing for a particular function

(Yacas internal)

Calling format:

```
TraceRule(template) expr
```

Parameters:

`template` – template showing the operator to trace
`expr` – expression to evaluate with tracing on

Description:

The tracing facility is turned on for subexpressions of the form "template", and the expression "expr" is evaluated. The template "template" is an example of the function to trace on. Specifically, all subexpressions with the same top-level operator and arity as "template" are shown. The subexpressions are displayed before (indicated with `TrEnter`) and after (`TrLeave`) evaluation. In between, the arguments are shown before and after evaluation (`TrArg`). Only functions defined in scripts can be traced.

This is useful for tracing a function that is called from within another function. This way you can see how your function behaves in the environment it is used in.

Examples:

```
In> TraceRule(x+y) 2+3*5+4;
    TrEnter(2+3*5+4);
      TrEnter(2+3*5);
          TrArg(2, 2);
          TrArg(3*5, 15);
      TrLeave(2+3*5, 17);
        TrArg(2+3*5, 17);
        TrArg(4, 4);
    TrLeave(2+3*5+4, 21);
Out> 21;
```

See also: `TraceStack`, `TraceExp`

`Time` — measure the time taken by a function

(standard library)

Calling format:

```
Time(expr)
```

Parameters:

`expr` – any expression

Description:

The function `Time(expr)` evaluates the expression `expr` and prints the time in seconds needed for the evaluation. The time is printed to the current output stream. The built-in function `GetTime` is used for timing.

The result is the "user time" as reported by the OS, not the real ("wall clock") time. Therefore, any CPU-intensive processes running alongside Yacas will not significantly affect the result of `Time`.

Example:

```
In> Time(N(MathLog(1000),40))
0.34 seconds taken
Out> 6.9077552789821370520539743640530926228033;
```

See also: `GetTime`

Chapter 24

Predicates

A predicate is a function that returns a boolean value, i.e. `True` or `False`. Predicates are often used in patterns, For instance, a rule that only holds for a positive integer would use a pattern such as `n_IsPositiveInteger`.

`!=` — test for "not equal"

(standard library)

Calling format:

```
e1 != e2
```

Precedence: 90

Parameters:

`e1`, `e2` – expressions to be compared

Description:

Both expressions are evaluated and compared. If they turn out to be equal, the result is `False`. Otherwise, the result is `True`.

The expression `e1 != e2` is equivalent to `Not(e1 = e2)`.

Examples:

```
In> 1 != 2;
Out> True;
In> 1 != 1;
Out> False;
```

See also: `=`

`=` — test for equality of expressions

(standard library)

Calling format:

```
e1 = e2
```

Precedence: 90

Parameters:

`e1`, `e2` – expressions to be compared

Description:

Both expressions are evaluated and compared. If they turn out to be equal, the result is `True`. Otherwise, the result is `False`. The function `Equals` does the same.

Note that the test is on syntactic equality, not mathematical equality. Hence even if the result is `False`, the expressions can still be *mathematically* equal; see the examples below. Put otherwise, this function tests whether the two expressions would be displayed in the same way if they were printed.

Examples:

```
In> e1 := (x+1) * (x-1);
Out> (x+1)*(x-1);
In> e2 := x^2 - 1;
Out> x^2-1;

In> e1 = e2;
Out> False;
In> Expand(e1) = e2;
Out> True;
```

See also: `!=`, `Equals`

`Not` — logical negation

(Yacas internal)

Calling format:

```
Not expr
```

Parameters:

`expr` – a boolean expression

Description:

Not returns the logical negation of the argument expr. If `expr` is `False` it returns `True`, and if `expr` is `True`, `Not expr` returns `False`. If the argument is neither `True` nor `False`, it returns the entire expression with evaluated arguments.

Examples:

```
In> Not True
Out> False;
In> Not False
Out> True;
In> Not(a)
Out> Not a;
```

See also: `And`, `Or`

`And` — logical conjunction

(Yacas internal)

Calling format:

```
a1 And a2
```

Precedence: 1000

```
And(a1, a2, a3, ..., aN)
```

Parameters:

`a1`, ..., `aN` – boolean values (may evaluate to `True` or `False`)

Description:

This function returns `True` if all arguments are true. The `And` operation is "lazy", i.e. it returns `False` as soon as a `False` argument is found (from left to right). If an argument other than `True` or `False` is encountered a new `And` expression is returned with all arguments that didn't evaluate to `True` or `False` yet.

Examples:

```
In> True And False
Out> False;
In> And(True,True)
Out> True;
In> False And a
Out> False;
In> True And a
Out> And(a);
In> And(True,a,True,b)
Out> b And a;
```

See also: `Or`, `Not`

`Or` — logical disjunction

(Yacas internal)

Calling format:

```
a1 Or a2
```

Precedence: 1010

```
Or(a1, a2, a3, ..., aN)
```

Parameters:

`a1`, ..., `aN` – boolean expressions (may evaluate to `True` or `False`)

Description:

This function returns `True` if an argument is encountered that is true (scanning from left to right). The `Or` operation is "lazy", i.e. it returns `True` as soon as a `True` argument is found (from left to right). If an argument other than `True` or `False` is encountered, an unevaluated `Or` expression is returned with all arguments that didn't evaluate to `True` or `False` yet.

Examples:

```
In> True Or False
Out> True;
In> False Or a
Out> Or(a);
In> Or(False,a,b,True)
Out> True;
```

See also: `And`, `Not`

`IsFreeOf` — test whether expression depends on variable

(standard library)

Calling format:

```
IsFreeOf(var, expr)
IsFreeOf({var, ...}, expr)
```

Parameters:

`expr` – expression to test
`var` – variable to look for in "expr"

Description:

This function checks whether the expression "expr" (after being evaluated) depends on the variable "var". It returns `False` if this is the case and `True` otherwise.

The second form test whether the expression depends on *any* of the variables named in the list. The result is `True` if none of the variables appear in the expression and `False` otherwise.

Examples:

```
In> IsFreeOf(x, Sin(x));
Out> False;
In> IsFreeOf(y, Sin(x));
Out> True;
In> IsFreeOf(x, D(x) a*x+b);
Out> True;
In> IsFreeOf({x,y}, Sin(x));
Out> False;
```

The third command returns `True` because the expression `D(x) a*x+b` evaluates to `a`, which does not depend on `x`.

See also: `Contains`

`IsZeroVector` — test whether list contains only zeroes

(standard library)

Calling format:

```
IsZeroVector(list)
```

Parameters:

`list` – list to compare against the zero vector

Description:

The only argument given to `IsZeroVector` should be a list. The result is `True` if the list contains only zeroes and `False` otherwise.

Examples:

```
In> IsZeroVector({0, x, 0});
Out> False;
In> IsZeroVector({x-x, 1 - D(x) x});
Out> True;
```

See also: `IsList`, `ZeroVector`

IsNonObject — test whether argument is not an `Object()`

(standard library)

Calling format:

```
IsNonObject(expr)
```

Parameters:

`expr` – the expression to examine

Description:

This function returns `True` if "expr" is not of the form `Object(...)` and `False` otherwise.

Bugs

In fact, the result is always `True`.

See also: `Object`

IsEven — test for an even integer

(standard library)

Calling format:

```
IsEven(n)
```

Parameters:

`n` – integer to test

Description:

This function tests whether the integer "n" is even. An integer is even if it is divisible by two. Hence the even numbers are 0, 2, 4, 6, 8, 10, etc., and -2, -4, -6, -8, -10, etc.

Examples:

```
In> IsEven(4);
Out> True;
In> IsEven(-1);
Out> False;
```

See also: `IsOdd`, `IsInteger`

IsOdd — test for an odd integer

(standard library)

Calling format:

```
IsOdd(n)
```

Parameters:

`n` – integer to test

Description:

This function tests whether the integer "n" is odd. An integer is odd if it is not divisible by two. Hence the odd numbers are 1, 3, 5, 7, 9, etc., and -1, -3, -5, -7, -9, etc.

Examples:

```
In> IsOdd(4);
Out> False;
In> IsOdd(-1);
Out> True;
```

See also: `IsEven`, `IsInteger`

IsEvenFunction — Return true if function is an even function, False otherwise

IsOddFunction — Return true if function is an odd function, False otherwise

(standard library)

Calling format:

```
IsEvenFunction(expression,variable)
IsOddFunction(expression,variable)
```

Parameters:

`expression` – mathematical expression `variable` – variable

Description:

These functions return True if Yacas can determine that the function is even or odd respectively. Even functions are defined to be functions that have the property:

$$f(x) = f(-x)$$

And odd functions have the property:

$$f(x) = -f(-x)$$

`Sin(x)` is an example of an odd function, and `Cos(x)` is an example of an even function.

As a side note, one can decompose a function into an even and an odd part:

$$f(x) = f_{\mathrm{even}}(x) + f_{\mathrm{odd}}(x)$$

Where

$$f_{\mathrm{even}}(x) = \frac{f(x) + f(-x)}{2}$$

and

$$f_{\mathrm{odd}}(x) = \frac{f(x) - f(-x)}{2}$$

Example:

```
In> IsEvenFunction(Cos(b*x),x)
Out> True
In> IsOddFunction(Cos(b*x),x)
Out> False
In> IsOddFunction(Sin(b*x),x)
Out> True
In> IsEvenFunction(Sin(b*x),x)
Out> False
In> IsEvenFunction(1/x^2,x)
Out> True
In> IsEvenFunction(1/x,x)
Out> False
In> IsOddFunction(1/x,x)
Out> True
In> IsOddFunction(1/x^2,x)
Out> False
```

See also: `Sin`, `Cos`

IsFunction — test for a composite object

(Yacas internal)

Calling format:

```
IsFunction(expr)
```

Parameters:

expr – expression to test

Description:

This function tests whether "expr" is a composite object, i.e. not an atom. This includes not only obvious functions such as f(x), but also expressions such as x+5 and lists.

Examples:

```
In> IsFunction(x+5);
Out> True;
In> IsFunction(x);
Out> False;
```

See also: IsAtom, IsList, Type

IsAtom — test for an atom

(Yacas internal)

Calling format:

```
IsAtom(expr)
```

Parameters:

expr – expression to test

Description:

This function tests whether "expr" is an atom. Numbers, strings, and variables are all atoms.

Examples:

```
In> IsAtom(x+5);
Out> False;
In> IsAtom(5);
Out> True;
```

See also: IsFunction, IsNumber, IsString

IsString — test for an string

(Yacas internal)

Calling format:

```
IsString(expr)
```

Parameters:

expr – expression to test

Description:

This function tests whether "expr" is a string. A string is a text within quotes, e.g. "duh".

Examples:

```
In> IsString("duh");
Out> True;
In> IsString(duh);
Out> False;
```

See also: IsAtom, IsNumber

IsNumber — test for a number

(Yacas internal)

Calling format:

```
IsNumber(expr)
```

Parameters:

expr – expression to test

Description:

This function tests whether "expr" is a number. There are two kinds of numbers, integers (e.g. 6) and reals (e.g. -2.75 or 6.0). Note that a complex number is represented by the Complex function, so IsNumber will return False.

Examples:

```
In> IsNumber(6);
Out> True;
In> IsNumber(3.25);
Out> True;
In> IsNumber(I);
Out> False;
In> IsNumber("duh");
Out> False;
```

See also: IsAtom, IsString, IsInteger, IsPositiveNumber, IsNegativeNumber, Complex

IsList — test for a list

(Yacas internal)

Calling format:

```
IsList(expr)
```

Parameters:

expr – expression to test

Description:

This function tests whether "expr" is a list. A list is a sequence between curly braces, e.g. {2, 3, 5}.

Examples:

```
In> IsList({2,3,5});
Out> True;
In> IsList(2+3+5);
Out> False;
```

See also: IsFunction

`IsNumericList` — test for a list of numbers

(standard library)

Calling format:

```
IsNumericList({list})
```

Parameters:

`{list}` – a list

Description:

Returns `True` when called on a list of numbers or expressions that evaluate to numbers using `N()`. Returns `False` otherwise.

See also: `N`, `IsNumber`

`IsBound` — test for a bound variable

(Yacas internal)

Calling format:

```
IsBound(var)
```

Parameters:

`var` – variable to test

Description:

This function tests whether the variable "var" is bound, i.e. whether it has been assigned a value. The argument "var" is not evaluated.

Examples:

```
In> IsBound(x);
Out> False;
In> x := 5;
Out> 5;
In> IsBound(x);
Out> True;
```

See also: `IsAtom`

`IsBoolean` — test for a Boolean value

(standard library)

Calling format:

```
IsBoolean(expression)
```

Parameters:

`expression` – an expression

Description:

IsBoolean returns True if the argument is of a boolean type. This means it has to be either True, False, or an expression involving functions that return a boolean result, e.g. `=`, `>`, `<`, `>=`, `<=`, `!=`, `And`, `Not`, `Or`.

Examples:

```
In> IsBoolean(a)
Out> False;
In> IsBoolean(True)
Out> True;
In> IsBoolean(a And b)
Out> True;
```

See also: `True`, `False`

`IsNegativeNumber` — test for a negative number

(standard library)

Calling format:

```
IsNegativeNumber(n)
```

Parameters:

`n` – number to test

Description:

`IsNegativeNumber(n)` evaluates to `True` if n is (strictly) negative, i.e. if $n < 0$. If `n` is not a number, the functions return `False`.

Examples:

```
In> IsNegativeNumber(6);
Out> False;
In> IsNegativeNumber(-2.5);
Out> True;
```

See also: `IsNumber`, `IsPositiveNumber`, `IsNotZero`, `IsNegativeInteger`, `IsNegativeReal`

`IsNegativeInteger` — test for a negative integer

(standard library)

Calling format:

```
IsNegativeInteger(n)
```

Parameters:

`n` – integer to test

Description:

This function tests whether the integer `n` is (strictly) negative. The negative integers are -1, -2, -3, -4, -5, etc. If `n` is not a integer, the function returns `False`.

Examples:

```
In> IsNegativeInteger(31);
Out> False;
In> IsNegativeInteger(-2);
Out> True;
```

See also: `IsPositiveInteger`, `IsNonZeroInteger`, `IsNegativeNumber`

IsPositiveNumber — test for a positive number

(standard library)

Calling format:

```
IsPositiveNumber(n)
```

Parameters:

n – number to test

Description:

IsPositiveNumber(n) evaluates to True if n is (strictly) positive, i.e. if $n > 0$. If n is not a number the function returns False.

Examples:

```
In> IsPositiveNumber(6);
Out> True;
In> IsPositiveNumber(-2.5);
Out> False;
```

See also: IsNumber, IsNegativeNumber, IsNotZero, IsPositiveInteger, IsPositiveReal

IsPositiveInteger — test for a positive integer

(standard library)

Calling format:

```
IsPositiveInteger(n)
```

Parameters:

n – integer to test

Description:

This function tests whether the integer n is (strictly) positive. The positive integers are 1, 2, 3, 4, 5, etc. If n is not a integer, the function returns False.

Examples:

```
In> IsPositiveInteger(31);
Out> True;
In> IsPositiveInteger(-2);
Out> False;
```

See also: IsNegativeInteger, IsNonZeroInteger, IsPositiveNumber

IsNotZero — test for a nonzero number

(standard library)

Calling format:

```
IsNotZero(n)
```

Parameters:

n – number to test

Description:

IsNotZero(n) evaluates to True if n is not zero. In case n is not a number, the function returns False.

Examples:

```
In> IsNotZero(3.25);
Out> True;
In> IsNotZero(0);
Out> False;
```

See also: IsNumber, IsPositiveNumber, IsNegativeNumber, IsNonZeroInteger

IsNonZeroInteger — test for a nonzero integer

(standard library)

Calling format:

```
IsNonZeroInteger(n)
```

Parameters:

n – integer to test

Description:

This function tests whether the integer n is not zero. If n is not an integer, the result is False.

Examples:

```
In> IsNonZeroInteger(0)
Out> False;
In> IsNonZeroInteger(-2)
Out> True;
```

See also: IsPositiveInteger, IsNegativeInteger, IsNotZero

IsInfinity — test for an infinity

(standard library)

Calling format:

```
IsInfinity(expr)
```

Parameters:

expr – expression to test

Description:

This function tests whether `expr` is an infinity. This is only the case if `expr` is either `Infinity` or `-Infinity`.

Examples:

```
In> IsInfinity(10^1000);
Out> False;
In> IsInfinity(-Infinity);
Out> True;
```

See also: `Integer`

`IsPositiveReal` — test for a numerically positive value

(standard library)

Calling format:

```
IsPositiveReal(expr)
```

Parameters:

`expr` – expression to test

Description:

This function tries to approximate "expr" numerically. It returns `True` if this approximation is positive. In case no approximation can be found, the function returns `False`. Note that round-off errors may cause incorrect results.

Examples:

```
In> IsPositiveReal(Sin(1)-3/4);
Out> True;
In> IsPositiveReal(Sin(1)-6/7);
Out> False;
In> IsPositiveReal(Exp(x));
Out> False;
```

The last result is because `Exp(x)` cannot be numerically approximated if `x` is not known. Hence Yacas can not determine the sign of this expression.

See also: `IsNegativeReal`, `IsPositiveNumber`, `N`

`IsNegativeReal` — test for a numerically negative value

(standard library)

Calling format:

```
IsNegativeReal(expr)
```

Parameters:

`expr` – expression to test

Description:

This function tries to approximate `expr` numerically. It returns `True` if this approximation is negative. In case no approximation can be found, the function returns `False`. Note that round-off errors may cause incorrect results.

Examples:

```
In> IsNegativeReal(Sin(1)-3/4);
Out> False;
In> IsNegativeReal(Sin(1)-6/7);
Out> True;
In> IsNegativeReal(Exp(x));
Out> False;
```

The last result is because `Exp(x)` cannot be numerically approximated if `x` is not known. Hence Yacas can not determine the sign of this expression.

See also: `IsPositiveReal`, `IsNegativeNumber`, `N`

`IsConstant` — test for a constant

(standard library)

Calling format:

```
IsConstant(expr)
```

Parameters:

`expr` – some expression

Description:

`IsConstant` returns `True` if the expression is some constant or a function with constant arguments. It does this by checking that no variables are referenced in the expression. `Pi` is considered a constant.

Examples:

```
In> IsConstant(Cos(x))
Out> False;
In> IsConstant(Cos(2))
Out> True;
In> IsConstant(Cos(2+x))
Out> False;
```

See also: `IsNumber`, `IsInteger`, `VarList`

`IsGaussianInteger` — test for a Gaussian integer

(standard library)

Calling format:

```
IsGaussianInteger(z)
```

Parameters:

`z` – a complex or real number

Description:

This function returns True if the argument is a Gaussian integer and False otherwise. A Gaussian integer is a generalization of integers into the complex plane. A complex number $a + b\imath$ is a Gaussian integer if and only if a and b are integers.

Examples:

```
In> IsGaussianInteger(5)
Out> True;
In> IsGaussianInteger(5+6*I)
Out> True;
In> IsGaussianInteger(1+2.5*I)
Out> False;
```

See also: IsGaussianUnit, IsGaussianPrime

MatchLinear — match an expression to a polynomial of degree one in a variable

(standard library)

Calling format:

```
MatchLinear(x,expr)
```

Parameters:

x – variable to express the univariate polynomial in
expr – expression to match

Description:

MatchLinear tries to match an expression to a linear (degree less than two) polynomial. The function returns True if it could match, and it stores the resulting coefficients in the variables "a" and "b" as a side effect. The function calling this predicate should declare local variables "a" and "b" for this purpose. MatchLinear tries to match to constant coefficients which don't depend on the variable passed in, trying to find a form "a*x+b" with "a" and "b" not depending on x if x is given as the variable.

Examples:

```
In> MatchLinear(x,(R+1)*x+(T-1))
Out> True;
In> {a,b};
Out> {R+1,T-1};
In> MatchLinear(x,Sin(x)*x+(T-1))
Out> False;
```

See also: Integrate

HasExpr — check for expression containing a subexpression

HasExprArith — check for expression containing a subexpression

HasExprSome — check for expression containing a subexpression

(standard library)

Calling format:

```
HasExpr(expr, x)
HasExprArith(expr, x)
HasExprSome(expr, x, list)
```

Parameters:

expr – an expression
x – a subexpression to be found
list – list of function atoms to be considered "transparent"

Description:

The command HasExpr returns True if the expression expr contains a literal subexpression x. The expression is recursively traversed.

The command HasExprSome does the same, except it only looks at arguments of a given list of functions. All other functions become "opaque" (as if they do not contain anything).

HasExprArith is defined through HasExprSome to look only at arithmetic operations +, -, *, /.

Note that since the operators "+" and "-" are prefix as well as infix operators, it is currently required to use Atom("+") to obtain the unevaluated atom "+".

Examples:

```
In> HasExpr(x+y*Cos(Ln(z)/z), z)
Out> True;
In> HasExpr(x+y*Cos(Ln(z)/z), Ln(z))
Out> True;
In> HasExpr(x+y*Cos(Ln(z)/z), z/Ln(z))
Out> False;
In> HasExprArith(x+y*Cos(Ln(x)/x), z)
Out> False;
In> HasExprSome({a+b*2,c/d},c/d,{List})
Out> True;
In> HasExprSome({a+b*2,c/d},c,{List})
Out> False;
```

See also: FuncList, VarList, HasFunc

HasFunc — check for expression containing a function

HasFuncArith — check for expression containing a function

HasFuncSome — check for expression containing a function

(standard library)

Calling format:

```
HasFunc(expr, func)
HasFuncArith(expr, func)
HasFuncSome(expr, func, list)
```

Parameters:

expr – an expression
func – a function atom to be found
list – list of function atoms to be considered "transparent"

Description:

The command HasFunc returns True if the expression expr contains a function func. The expression is recursively traversed.

The command HasFuncSome does the same, except it only looks at arguments of a given list of functions. Arguments of all other functions become "opaque" (as if they do not contain anything).

HasFuncArith is defined through HasFuncSome to look only at arithmetic operations +, -, *, /.

Note that since the operators "+" and "-" are prefix as well as infix operators, it is currently required to use Atom("+") to obtain the unevaluated atom "+".

Examples:

```
In> HasFunc(x+y*Cos(Ln(z)/z), Ln)
Out> True;
In> HasFunc(x+y*Cos(Ln(z)/z), Sin)
Out> False;
In> HasFuncArith(x+y*Cos(Ln(x)/x), Cos)
Out> True;
In> HasFuncArith(x+y*Cos(Ln(x)/x), Ln)
Out> False;
In> HasFuncSome({a+b*2,c/d},/,{List})
Out> True;
In> HasFuncSome({a+b*2,c/d},*,{List})
Out> False;
```

See also: FuncList, VarList, HasExpr

Chapter 25

Yacas-specific constants

`%` — previous result

(Yacas internal)

Calling format:

```
%
```

Description:

`%` evaluates to the previous result on the command line. `%` is a global variable that is bound to the previous result from the command line. Using `%` will evaluate the previous result. (This uses the functionality offered by the `SetGlobalLazyVariable` command).

Typical examples are `Simplify(%)` and `PrettyForm(%)` to simplify and show the result in a nice form respectively.

Examples:

```
In> Taylor(x,0,5)Sin(x)
Out> x-x^3/6+x^5/120;
In> PrettyForm(%)

     3    5
    x    x
x - -- + ---
    6    120
```

See also: `SetGlobalLazyVariable`

`True` — boolean constant representing true

`False` — boolean constant representing false

(Yacas internal)

Calling format:

```
True
False
```

Description:

`True` and `False` are typically a result of boolean expressions such as `2 < 3` or `True And False`.

See also: `And`, `Or`, `Not`

`EndOfFile` — end-of-file marker

(Yacas internal)

Calling format:

```
EndOfFile
```

Description:

End of file marker when reading from file. If a file contains the expression `EndOfFile;` the operation will stop reading the file at that point.

Chapter 26

Mathematical constants

`Infinity` — constant representing mathematical infinity

(standard library)

Calling format:

```
Infinity
```

Description:

Infinity represents infinitely large values. It can be the result of certain calculations.

Note that for most analytic functions Yacas understands `Infinity` as a positive number. Thus `Infinity*2` will return `Infinity`, and `a < Infinity` will evaluate to `True`.

Examples:

```
In> 2*Infinity
Out> Infinity;
In> 2<Infinity
Out> True;
```

`Pi` — mathematical constant, π

(standard library)

Calling format:

```
Pi
```

Description:

Pi symbolically represents the exact value of π. When the `N()` function is used, `Pi` evaluates to a numerical value according to the current precision. It is better to use `Pi` than `N(Pi)` except in numerical calculations, because exact simplification will be possible.

This is a "cached constant" which is recalculated only when precision is increased.

Examples:

```
In> Sin(3*Pi/2)
Out> -1;
In> Pi+1
Out> Pi+1;
In> N(Pi)
Out> 3.14159265358979323846;
```

See also: `Sin`, `Cos`, `N`, `CachedConstant`

`Undefined` — constant signifying an undefined result

(standard library)

Calling format:

```
Undefined
```

Description:

`Undefined` is a token that can be returned by a function when it considers its input to be invalid or when no meaningful answer can be given. The result is then "undefined".

Most functions also return `Undefined` when evaluated on it.

Examples:

```
In> 2*Infinity
Out> Infinity;
In> 0*Infinity
Out> Undefined;
In> Sin(Infinity);
Out> Undefined;
In> Undefined+2*Exp(Undefined);
Out> Undefined;
```

See also: `Infinity`

`GoldenRatio` — the Golden Ratio

(standard library)

Calling format:

```
GoldenRatio
```

Description:

These functions compute the "golden ratio"

$$\phi \approx 1.6180339887 \approx \frac{1+\sqrt{5}}{2}.$$

The ancient Greeks defined the "golden ratio" as follows: If one divides a length 1 into two pieces x and $1-x$, such that the ratio of 1 to x is the same as the ratio of x to $1-x$, then $\frac{1}{x} \approx 1.618...$ is the "golden ratio".

The constant is available symbolically as `GoldenRatio` or numerically through `N(GoldenRatio)`. This is a "cached constant" which is recalculated only when precision is increased. The numerical value of the constant can also be obtained as `N(GoldenRatio)`.

Examples:

```
In> x:=GoldenRatio - 1
Out> GoldenRatio-1;
In> N(x)
Out> 0.6180339887;
In> N(1/GoldenRatio)
Out> 0.6180339887;
In> V(N(GoldenRatio,20));
CachedConstant: Info: constant GoldenRatio is
being recalculated at precision 20
Out> 1.6180339887498948482;
```

See also: N, CachedConstant

Catalan — Catalan's Constant

(standard library)

Calling format:

```
Catalan
```

Description:

These functions compute Catalan's Constant Catalan $\approx$ 0.9159655941.

The constant is available symbolically as Catalan or numerically through N(Catalan) with N(...) the usual operator used to try to coerce an expression in to a numeric approximation of that expression. This is a "cached constant" which is recalculated only when precision is increased. The numerical value of the constant can also be obtained as N(Catalan). The low-level numerical computations are performed by the routine CatalanConstNum.

Examples:

```
In> N(Catalan)
Out> 0.9159655941;
In> DirichletBeta(2)
Out> Catalan;
In> V(N(Catalan,20))
CachedConstant: Info: constant Catalan is
being recalculated at precision 20
Out> 0.91596559417721901505;
```

See also: N, CachedConstant

gamma — Euler's constant γ

(standard library)

Calling format:

```
gamma
```

Description:

These functions compute Euler's constant $\gamma \approx 0.57722...$

The constant is available symbolically as gamma or numerically through using the usual function N(...) to get a numeric result, N(gamma). This is a "cached constant" which is recalculated only when precision is increased. The numerical value of the constant can also be obtained as N(gamma). The low-level numerical computations are performed by the routine GammaConstNum.

Note that Euler's Gamma function $\Gamma(x)$ is the capitalized Gamma in Yacas.

Examples:

```
In> gamma+Pi
Out> gamma+Pi;
In> N(gamma+Pi)
Out> 3.7188083184;
In> V(N(gamma,20))

CachedConstant: Info: constant gamma is being
  recalculated at precision 20
GammaConstNum: Info: used 56 iterations at
  working precision 24
Out> 0.57721566490153286061;
```

See also: Gamma, N, CachedConstant

Chapter 27

Variables

:= — assign a variable or a list; define a function

(standard library)

Calling format:

```
var := expr
{var1, var2, ...} := {expr1, expr2, ...}
var[i] := expr
fn(arg1, arg2, ...) := expr
```

Precedence: 10000

Parameters:

`var` – atom, variable which should be assigned
`expr` – expression to assign to the variable or body of function
`i` – index (can be integer or string)
`fn` – atom, name of a new function to define
`arg1`, `arg2` – atoms, names of arguments of the new function `fn`

Description:

The `:=` operator can be used in a number of ways. In all cases, some sort of assignment or definition takes place.

The first form is the most basic one. It evaluates the expression on the right-hand side and assigns it to the variable named on the left-hand side. The left-hand side is not evaluated. The evaluated expression is also returned.

The second form is a small extension, which allows one to do multiple assignments. The first entry in the list on the right-hand side is assigned to the first variable mentioned in the left-hand side, the second entry on the right-hand side to the second variable on the left-hand side, etc. The list on the right-hand side must have at least as many entries as the list on the left-hand side. Any excess entries are silently ignored. The result of the expression is the list of values that have been assigned.

The third form allows one to change an entry in the list. If the index "i" is an integer, the "i"-th entry in the list is changed to the expression on the right-hand side. It is assumed that the length of the list is at least "i". If the index "i" is a string, then "var" is considered to be an associative list (sometimes called hash table), and the key "i" is paired with the value "exp". In both cases, the right-hand side is evaluated before the assignment and the result of the assignment is `True`.

The last form defines a function. For example, the assignment `fn(x) := x2̂` removes any rules previously associated with `fn(x)` and defines the rule `fn(_x) <-- x2̂`. Note that the left-hand side may take a different form if `fn` is defined to be a prefix, infix or bodied function. This case is special since the right-hand side is not evaluated immediately, but only when the function `fn` is used. If this takes time, it may be better to force an immediate evaluation with `Eval` (see the last example). If the expression on the right hand side begins with `Eval()`, then it *will* be evaluated before defining the new function.

A variant of the function definition can be used to make a function accepting a variable number of arguments. The last argument

Examples:

A simple assignment:

```
In> a := Sin(x) + 3;
Out> Sin(x)+3;
In> a;
Out> Sin(x)+3;
```

Multiple assignments:

```
In> {a,b,c} := {1,2,3};
Out> {1,2,3};
In> a;
Out> 1;
In> b+c;
Out> 5;
```

Assignment to a list:

```
In> xs := { 1,2,3,4,5 };
Out> {1,2,3,4,5};
In> xs[3] := 15;
Out> True;
In> xs;
Out> {1,2,15,4,5};
```

Building an associative list:

```
In> alist := {};
Out> {};
In> alist["cherry"] := "red";
Out> True;
In> alist["banana"] := "yellow";
Out> True;
In> alist["cherry"];
Out> "red";
In> alist;
Out> {{"banana","yellow"},{"cherry","red"}};
```

Defining a function:

```
In> f(x) := x^2;
Out> True;
In> f(3);
Out> 9;
In> f(Sin(a));
Out> Sin(a)^2;
```

Defining a function with variable number of arguments:

```
In> f(x, ...) := If(IsList(x),Sum(x),x);
Out> True;
In> f(2);
Out> 2;
In> f(1,2,3);
Out> 6;
```

Defining a new infix operator:

```
In> Infix("*&*",10);
Out> True;
In> x1 *&* x2 := x1/x2 + x2/x1;
Out> True;
In> Sin(a) *&* Cos(a);
Out> Tan(1)+Cos(1)/Sin(1);
In> Clear(a);
Out> True;
In> Sin(a) *&* Exp(a);
Out> Sin(a)/Exp(a)+Exp(a)/Sin(a);
```

In the following example, it may take some time to compute the Taylor expansion. This has to be done every time the function f is called.

```
In> f(a) := Taylor(x,0,25) Sin(x);
Out> True;
In> f(1);
Out> x-x^3/6+x^5/120-x^7/5040+x^9/362880-
x^11/39916800+x^13/6227020800-x^15/
1307674368000+x^17/355687428096000-x^19/
121645100408832000+x^21/51090942171709440000
-x^23/25852016738884976640000+x^25
/15511210043330985984000000;
In> f(2);
Out> x-x^3/6+x^5/120-x^7/5040+x^9/362880-
x^11/39916800+x^13/6227020800-x^15
/1307674368000+x^17/355687428096000-x^19/
121645100408832000+x^21/51090942171709440000
-x^23/25852016738884976640000+x^25/
15511210043330985984000000;
```

The remedy is to evaluate the Taylor expansion immediately. Now the expansion is computed only once.

```
In> f(a) := Eval(Taylor(x,0,25) Sin(x));
Out> True;
In> f(1);
Out> x-x^3/6+x^5/120-x^7/5040+x^9/362880-
x^11/39916800+x^13/6227020800-x^15/
1307674368000+x^17/355687428096000-x^19/
121645100408832000+x^21/51090942171709440000
-x^23/25852016738884976640000+x^25
/15511210043330985984000000;
In> f(2);
Out> x-x^3/6+x^5/120-x^7/5040+x^9/362880-
x^11/39916800+x^13/6227020800-x^15
/1307674368000+x^17/355687428096000-x^19/
121645100408832000+x^21/51090942171709440000
-x^23/25852016738884976640000+x^25/
15511210043330985984000000;
```

See also: `Set`, `Clear`, `[]`, `Rule`, `Infix`, `Eval`, `Function`

`Set` — assignment

(Yacas internal)

Calling format:

```
Set(var, exp)
```

Parameters:

`var` – variable which should be assigned
`exp` – expression to assign to the variable

Description:

The expression "exp" is evaluated and assigned it to the variable named "var". The first argument is not evaluated. The value True is returned.

The statement `Set(var, exp)` is equivalent to `var := exp`, but the `:=` operator has more uses, e.g. changing individual entries in a list.

Examples:

```
In> Set(a, Sin(x)+3);
Out> True;
In> a;
Out> Sin(x)+3;
```

See also: `Clear`, `:=`

`Clear` — undo an assignment

(Yacas internal)

Calling format:

```
Clear(var, ...)
```

Parameters:

`var` – name of variable to be cleared

Description:

All assignments made to the variables listed as arguments are undone. From now on, all these variables remain unevaluated (until a subsequent assignment is made). The result of the expression is True.

Examples:

```
In> a := 5;
Out> 5;
In> a^2;
Out> 25;

In> Clear(a);
Out> True;
In> a^2;
Out> a^2;
```

See also: `Set`, `:=`

Local — declare new local variables

(Yacas internal)

Calling format:

```
Local(var, ...)
```

Parameters:

`var` – name of variable to be declared as local

Description:

All variables in the argument list are declared as local variables. The arguments are not evaluated. The value True is returned.

By default, all variables in Yacas are global. This means that the variable has the same value everywhere. But sometimes it is useful to have a private copy of some variable, either to prevent the outside world from changing it or to prevent accidental changes to the outside world. This can be achieved by declaring the variable local. Now only expressions within the `Prog` block (or its syntactic equivalent, the `[ ]` block) can access and change it. Functions called within this block cannot access the local copy unless this is specifically allowed with `UnFence`.

Examples:

```
In> a := 3;
Out> 3;

In> [ a := 4; a; ];
Out> 4;
In> a;
Out> 4;

In> [ Local(a); a := 5; a; ];
Out> 5;
In> a;
Out> 4;
```

In the first block, `a` is not declared local and hence defaults to be a global variable. Indeed, changing the variable inside the block also changes the value of `a` outside the block. However, in the second block `a` is defined to be local and now the value outside the block stays the same, even though `a` is assigned the value 5 inside the block.

See also: `LocalSymbols`, `Prog`, `[]`, `UnFence`

++ — increment variable

(standard library)

Calling format:

```
var++
```

Parameters:

`var` – variable to increment

Description:

The variable with name "var" is incremented, i.e. the number 1 is added to it. The expression `x++` is equivalent to the assignment `x := x + 1`, except that the assignment returns the new value of `x` while `x++` always returns true. In this respect, Yacas' `++` differs from the corresponding operator in the programming language C.

Examples:

```
In> x := 5;
Out> 5;
In> x++;
Out> True;
In> x;
Out> 6;
```

See also: `--`, `:=`

-- — decrement variable

(standard library)

Calling format:

```
var--
```

Parameters:

`var` – variable to decrement

Description:

The variable with name "var" is decremented, i.e. the number 1 is subtracted from it. The expression `x--` is equivalent to the assignment `x := x - 1`, except that the assignment returns the new value of `x` while `x--` always returns true. In this respect, Yacas' `--` differs from the corresponding operator in the programming language C.

Examples:

```
In> x := 5;
Out> 5;
In> x--;
Out> True;
In> x;
Out> 4;
```

See also: `++`, `:=`

Object — create an incomplete type

(standard library)

Calling format:

```
Object("pred", exp)
```

Parameters:

`pred` – name of the predicate to apply
`exp` – expression on which "pred" should be applied

Description:

This function returns "obj" as soon as "pred" returns `True` when applied on "obj". This is used to declare so-called incomplete types.

Examples:

```
In> a := Object("IsNumber", x);
Out> Object("IsNumber",x);
In> Eval(a);
Out> Object("IsNumber",x);
In> x := 5;
Out> 5;
In> Eval(a);
Out> 5;
```

See also: `IsNonObject`

SetGlobalLazyVariable — global variable is to be evaluated lazily

(Yacas internal)

Calling format:

```
SetGlobalLazyVariable(var,value)
```

Parameters:

var – variable (held argument)
value – value to be set to (evaluated before it is assigned)

Description:

SetGlobalLazyVariable enforces that a global variable will re-evaluate when used. This functionality doesn't survive if Clear(var) is called afterwards.

Places where this is used include the global variables % and I.

The use of lazy in the name stems from the concept of lazy evaluation. The object the global variable is bound to will only be evaluated when called. The SetGlobalLazyVariable property only holds once: after that, the result of evaluation is stored in the global variable, and it won't be reevaluated again:

```
In> SetGlobalLazyVariable(a,Hold(Taylor(x,0,30)Sin(x)))
Out> True
```

Then the first time you call a it evaluates Taylor(...) and assigns the result to a. The next time you call a it immediately returns the result. SetGlobalLazyVariable is called for % each time % changes.

The following example demonstrates the sequence of execution:

```
In> SetGlobalLazyVariable(test,Hold(Write("hello")))
Out> True
```

The text "hello" is not written out to screen yet. However, evaluating the variable test forces the expression to be evaluated:

```
In> test
"hello"Out> True
```

Examples:

```
In> Set(a,Hold(2+3))
Out> True
In> a
Out> 2+3
In> SetGlobalLazyVariable(a,Hold(2+3))
Out> True
In> a
Out> 5
```

See also: Set, Clear, Local, %, I

UniqueConstant — create a unique identifier

(standard library)

Calling format:

```
UniqueConstant()
```

Description:

This function returns a unique constant atom each time you call it. The atom starts with a C character, and a unique number is appended to it.

Examples:

```
In> UniqueConstant()
Out> C9
In>  UniqueConstant()
Out> C10
```

See also: LocalSymbols

LocalSymbols — create unique local symbols with given prefix

(standard library)

Calling format:

```
LocalSymbols(var1, var2, ...) body
```

Parameters:

var1, var2, ... – atoms, symbols to be made local
body – expression to execute

Description:

Given the symbols passed as the first arguments to LocalSymbols a set of local symbols will be created, and creates unique ones for them, typically of the form $<symbol><number>, where symbol was the symbol entered by the user, and number is a unique number. This scheme was used to ensure that a generated symbol can not accidentally be entered by a user.

This is useful in cases where a guaranteed free variable is needed, for example, in the macro-like functions (For, While, etc.).

Examples:

```
In> LocalSymbols(a,b)a+b
Out> $a6+ $b6;
```

See also: UniqueConstant

Chapter 28

Input/output and plotting

This chapter contains commands to use for input and output and plotting. All output commands write to the same destination stream, called the "current output". This is initially the screen, but may be redirected by some commands. Similarly, most input commands read from the "current input" stream, which can also be redirected. The exception to this rule are the commands for reading script files, which simply read a specified file.

FullForm — print an expression in LISP-format

(Yacas internal)

Calling format:

```
FullForm(expr)
```

Parameters:

`expr` – expression to be printed in LISP-format

Description:

Evaluates "expr", and prints it in LISP-format on the current output. It is followed by a newline. The evaluated expression is also returned.

This can be useful if you want to study the internal representation of a certain expression.

Examples:

```
In> FullForm(a+b+c);
(+ (+ a b )c )
Out> a+b+c;
In> FullForm(2*I*b^2);
(* (Complex 0 2 )(^ b 2 ))
Out> Complex(0,2)*b^2;
```

The first example shows how the expression `a+b+c` is internally represented. In the second example, `2*I` is first evaluated to `Complex(0,2)` before the expression is printed.

See also: `LispRead`, `Listify`, `Unlist`

Echo — high-level printing routine

(standard library)

Calling format:

```
Echo(item)
Echo(list)
Echo(item,item,item,...)
```

Parameters:

`item` – the item to be printed
`list` – a list of items to be printed

Description:

If passed a single item, `Echo` will evaluate it and print it to the current output, followed by a newline. If `item` is a string, it is printed without quotation marks.

If there is one argument, and it is a list, `Echo` will print all the entries in the list subsequently to the current output, followed by a newline. Any strings in the list are printed without quotation marks. All other entries are followed by a space.

`Echo` can be called with a variable number of arguments, they will all be printed, followed by a newline.

`Echo` always returns `True`.

Examples:

```
In> Echo(5+3);
 8
Out> True;
In> Echo({"The square of two is ", 2*2});
The square of two is 4
Out> True;
In> Echo("The square of two is ", 2*2);
The square of two is 4
Out> True;
```

Note that one must use the second calling format if one wishes to print a list:

```
In> Echo({a,b,c});
a b c
Out> True;
In> Echo({{a,b,c}});
{a,b,c}
Out> True;
```

See also: `PrettyForm`, `Write`, `WriteString`, `RuleBaseListed`

PrettyForm — print an expression nicely with ASCII art

(standard library)

Calling format:

```
PrettyForm(expr)
```

Parameters:

`expr` – an expression

Description:

`PrettyForm` renders an expression in a nicer way, using ascii art. This is generally useful when the result of a calculation is more complex than a simple number.

Examples:

```
In> Taylor(x,0,9)Sin(x)
Out> x-x^3/6+x^5/120-x^7/5040+x^9/362880;
In> PrettyForm(%)

     3     5       7        9
    x     x       x        x
x - -- + --- - ---- + ------
    6     120   5040   362880

Out> True;
```

See also: `EvalFormula`, `PrettyPrinter'Set`

`EvalFormula` — print an evaluation nicely with ASCII art

(standard library)

Calling format:

```
EvalFormula(expr)
```

Parameters:

`expr` – an expression

Description:

Show an evaluation in a nice way, using `PrettyPrinter'Set` to show 'input = output'.

Examples:

```
In> EvalFormula(Taylor(x,0,7)Sin(x))

                                            3     5
                                           x     x
Taylor( x , 0 , 5 , Sin( x ) ) = x - -- + ---
                                           6     120
```

See also: `PrettyForm`

`TeXForm` — export expressions to LaTeX

(standard library)

Calling format:

```
TeXForm(expr)
```

Parameters:

`expr` – an expression to be exported

Description:

`TeXForm` returns a string containing a LaTeX representation of the Yacas expression `expr`. Currently the exporter handles most expression types but not all.

Example:

```
In> TeXForm(Sin(a1)+2*Cos(b1))
Out> "$\sin a_{1} + 2 \cos b_{1}$";
```

See also: `PrettyForm`, `CForm`

`CForm` — export expression to C++ code

(standard library)

Calling format:

```
CForm(expr)
```

Parameters:

`expr` – expression to be exported

Description:

`CForm` returns a string containing C++ code that attempts to implement the Yacas expression `expr`. Currently the exporter handles most expression types but not all.

Example:

```
In> CForm(Sin(a1)+2*Cos(b1));
Out> "sin(a1) + 2 * cos(b1)";
```

See also: `PrettyForm`, `TeXForm`, `IsCFormable`

`IsCFormable` — check possibility to export expression to C++ code

(standard library)

Calling format:

```
IsCFormable(expr)
IsCFormable(expr, funclist)
```

Parameters:

`expr` – expression to be exported (this argument is not evaluated)

`funclist` – list of "allowed" function atoms

Description:

IsCFormable returns True if the Yacas expression expr can be exported into C++ code. This is a check whether the C++ exporter CForm can be safely used on the expression.

A Yacas expression is considered exportable if it contains only functions that can be translated into C++ (e.g. UnList cannot be exported). All variables and constants are considered exportable.

The verbose option prints names of functions that are not exportable.

The second calling format of IsCFormable can be used to "allow" certain function names that will be available in the C++ code.

Examples:

```
In> IsCFormable(Sin(a1)+2*Cos(b1))
Out> True;
In> V(IsCFormable(1+func123(b1)))
IsCFormable: Info: unexportable function(s):
  func123
Out> False;
```

This returned False because the function func123 is not available in C++. We can explicitly allow this function and then the expression will be considered exportable:

```
In> IsCFormable(1+func123(b1), {func123})
Out> True;
```

See also: CForm, V

Write — low-level printing routine

(Yacas internal)

Calling format:

```
Write(expr, ...)
```

Parameters:

expr – expression to be printed

Description:

The expression "expr" is evaluated and written to the current output. Note that Write accept an arbitrary number of arguments, all of which are written to the current output (see second example). Write always returns True.

Examples:

```
In> Write(1);
1Out> True;
In> Write(1,2);
 1 2Out> True;
```

Write does not write a newline, so the Out> prompt immediately follows the output of Write.

See also: Echo, WriteString

WriteString — low-level printing routine for strings

(Yacas internal)

Calling format:

```
WriteString(string)
```

Parameters:

string – the string to be printed

Description:

The expression "string" is evaluated and written to the current output without quotation marks. The argument should be a string. WriteString always returns True.

Examples:

```
In> Write("Hello, world!");
"Hello, world!"Out> True;
In> WriteString("Hello, world!");
Hello, world!Out> True;
```

This example clearly shows the difference between Write and WriteString. Note that Write and WriteString do not write a newline, so the Out> prompt immediately follows the output.

See also: Echo, Write

Space — print one or more spaces

(standard library)

Calling format:

```
Space()
Space(nr)
```

Parameters:

nr – the number of spaces to print

Description:

The command Space() prints one space on the current output. The second form prints nr spaces on the current output. The result is always True.

Examples:

```
In> Space(5);
     Out> True;
```

See also: Echo, Write, NewLine

NewLine — print one or more newline characters

(standard library)

Calling format:

```
NewLine()
NewLine(nr)
```

Parameters:

nr – the number of newline characters to print

Description:

The command NewLine() prints one newline character on the current output. The second form prints "nr" newlines on the current output. The result is always True.

Examples:

```
In> NewLine();

Out> True;
```

See also: Echo, Write, Space

FromFile — connect current input to a file

(Yacas internal)

Calling format:

```
FromFile(name) body
```

Parameters:

name - string, the name of the file to read
body - expression to be evaluated

Description:

The current input is connected to the file "name". Then the expression "body" is evaluated. If some functions in "body" try to read from current input, they will now read from the file "name". Finally, the file is closed and the result of evaluating "body" is returned.

Examples:

Suppose that the file foo contains

```
2 + 5;
```

Then we can have the following dialogue:

```
In> FromFile("foo") res := Read();
Out> 2+5;
In> FromFile("foo") res := ReadToken();
Out> 2;
```

See also: ToFile, FromString, Read, ReadToken

FromString — connect current input to a string

(Yacas internal)

Calling format:

```
FromString(str) body;
```

Parameters:

str – a string containing the text to parse
body – expression to be evaluated

Description:

The commands in "body" are executed, but everything that is read from the current input is now read from the string "str". The result of "body" is returned.

Examples:

```
In> FromString("2+5; this is never read") \
  res := Read();
Out> 2+5;
In> FromString("2+5; this is never read") \
  res := Eval(Read());
Out> 7;
```

See also: ToString, FromFile, Read, ReadToken

ToFile — connect current output to a file

(Yacas internal)

Calling format:

```
ToFile(name) body
```

Parameters:

name – string, the name of the file to write the result to
body – expression to be evaluated

Description:

The current output is connected to the file "name". Then the expression "body" is evaluated. Everything that the commands in "body" print to the current output, ends up in the file "name". Finally, the file is closed and the result of evaluating "body" is returned.

If the file is opened again, the old contents will be overwritten. This is a limitation of ToFile: one cannot append to a file that has already been created.

Examples:

Here is how one can create a file with C code to evaluate an expression:

```
In> ToFile("expr1.c") WriteString(
  CForm(Sqrt(x-y)*Sin(x)) );
Out> True;
```

The file expr1.c was created in the current working directory and it contains the line

```
sqrt(x-y)*sin(x)
```

As another example, take a look at the following command:

```
In> [ Echo("Result:");  \
  PrettyForm(Taylor(x,0,9) Sin(x)); ];
Result:

     3    5      7       9
    x    x      x       x
x - -- + --- - ---- + ------
    6    120   5040   362880

Out> True;
```

Now suppose one wants to send the output of this command to a file. This can be achieved as follows:

```
In> ToFile("out") [ Echo("Result:");  \
  PrettyForm(Taylor(x,0,9) Sin(x)); ];
Out> True;
```

After this command the file out contains:

```
Result:

     3    5      7       9
    x    x      x       x
x - -- + --- - ---- + ------
    6    120   5040   362880
```

See also: FromFile, ToString, Echo, Write, WriteString, PrettyForm, Taylor

ToString — connect current output to a string

(Yacas internal)

Calling format:

```
ToString() body
```

Parameters:

body – expression to be evaluated

Description:

The commands in "body" are executed. Everything that is printed on the current output, by Echo for instance, is collected in a string and this string is returned.

Examples:

```
In> str := ToString() [ WriteString(  \
  "The square of 8 is "); Write(8^2); ];
Out> "The square of 8 is  64";
```

See also: FromFile, ToString, Echo, Write, WriteString

Read — read an expression from current input

(Yacas internal)

Calling format:

```
Read()
```

Description:

Read an expression from the current input, and return it unevaluated. When the end of an input file is encountered, the token atom EndOfFile is returned.

Examples:

```
In> FromString("2+5;") Read();
Out> 2+5;
In> FromString("") Read();
Out> EndOfFile;
```

See also: FromFile, FromString, LispRead, ReadToken, Write

ToStdout — select initial output stream for output

(Yacas internal)

Calling format:

```
ToStdout() body
```

Parameters:

body – expression to be evaluated

Description:

When using ToString or ToFile, it might happen that something needs to be written to the standard default initial output (typically the screen). ToStdout can be used to select this stream.

Example:

```
In> ToString()[Echo("aaaa");ToStdout()Echo("bbbb");];
bbbb
Out> "aaaa
"
```

See also: ToString, ToFile

ReadCmdLineString — read an expression from command line and return in string

(Yacas internal)

Calling format:

```
ReadCmdLineString(prompt)
```

Parameters:

`prompt` – string representing the prompt shown on screen

Description:

This function allows for interactive input similar to the command line. When using this function, the history from the command line is also available.

The result is returned in a string, so it still needs to be parsed.

This function will typically be used in situations where one wants a custom read-eval-print loop.

Examples:

The following defines a function that when invoked keeps asking for an expression (the *read* step), and then takes the derivative of it (the *eval* step) and then uses PrettyForm to display the result (the *print* step).

```
In> ReEvPr() := \
In>   While(True) [ \
In>     PrettyForm(Deriv(x) \
In>      FromString(ReadCmdLineString("Deriv> "):";")Read()); \
In> ];
Out> True;
```

Then one can invoke the command, from which the following interaction might follow:

```
In> ReEvPr()
Deriv> Sin(a^2*x/b)

   /  2     \
   | a  * x |    2
Cos| ------ | * a  * b
   \   b    /
----------------------
           2
          b

Deriv> Sin(x)

Cos( x )

Deriv>
```

See also: `Read`, `LispRead`, `LispReadListed`

`LispRead` — read expressions in LISP syntax

`LispReadListed` — read expressions in LISP syntax

(Yacas internal)

Calling format:

```
LispRead()
LispReadListed()
```

Description:

The function `LispRead` reads an expression in the LISP syntax from the current input, and returns it unevaluated. When the end of an input file is encountered, the special token atom `EndOfFile` is returned.

The Yacas expression `a+b` is written in the LISP syntax as `(+ a b)`. The advantage of this syntax is that it is less ambiguous than the infix operator grammar that Yacas uses by default.

The function `LispReadListed` reads a LISP expression and returns it in a list, instead of the form usual to Yacas (expressions). The result can be thought of as applying `Listify` to `LispRead`. The function `LispReadListed` is more useful for reading arbitrary LISP expressions, because the first object in a list can be itself a list (this is never the case for Yacas expressions where the first object in a list is always a function atom).

Examples:

```
In> FromString("(+ a b)") LispRead();
Out> a+b;
In> FromString("(List (Sin x) (- (Cos x)))") \
  LispRead();
Out> {Sin(x),-Cos(x)};
In> FromString("(+ a b)")LispRead()
Out> a+b;
In> FromString("(+ a b)")LispReadListed()
Out> {+,a,b};
```

See also: `FromFile`, `FromString`, `Read`, `ReadToken`, `FullForm`

`ReadToken` — read a token from current input

(Yacas internal)

Calling format:

```
ReadToken()
```

Description:

Read a token from the current input, and return it unevaluated. The returned object is a Yacas atom (not a string). When the end of an input file is encountered, the token atom `EndOfFile` is returned.

A token is for computer languages what a word is for human languages: it is the smallest unit in which a command can be divided, so that the semantics (that is the meaning) of the command is in some sense a combination of the semantics of the tokens. Hence `a := foo` consists of three tokens, namely `a`, `:=`, and `foo`.

The parsing of the string depends on the syntax of the language. The part of the kernel that does the parsing is the "tokenizer". Yacas can parse its own syntax (the default tokenizer) or it can be instructed to parse XML or C++ syntax using the directives `DefaultTokenizer` or `XmlTokenizer`. Setting a tokenizer is a global action that affects all `ReadToken` calls.

Examples:

```
In> FromString("a := Sin(x)") While \
  ((tok := ReadToken()) != EndOfFile) \
  Echo(tok);
a
:=
Sin
(
x
)
Out> True;
```

We can read some junk too:

```
In> FromString("-$3")ReadToken();
Out> -$;
```

The result is an atom with the string representation `-$`. Yacas assumes that `-$` is an operator symbol yet to be defined. The "3" will be in the next token. (The results will be different if a non-default tokenizer is selected.)

See also: `FromFile`, `FromString`, `Read`, `LispRead`, `DefaultTokenizer`

`Load` — evaluate all expressions in a file

(Yacas internal)

Calling format:

```
Load(name)
```

Parameters:

`name` – string, name of the file to load

Description:

The file "name" is opened. All expressions in the file are read and evaluated. `Load` always returns `true`.

See also: `Use`, `DefLoad`, `DefaultDirectory`, `FindFile`

`Use` — load a file, but not twice

(Yacas internal)

Calling format:

```
Use(name)
```

Parameters:

`name` – string, name of the file to load

Description:

If the file "name" has been loaded before, either by an earlier call to `Use` or via the `DefLoad` mechanism, nothing happens. Otherwise all expressions in the file are read and evaluated. `Use` always returns `true`.

The purpose of this function is to make sure that the file will at least have been loaded, but is not loaded twice.

See also: `Load`, `DefLoad`, `DefaultDirectory`

`DefLoad` — load a `.def` file

(Yacas internal)

Calling format:

```
DefLoad(name)
```

Parameters:

`name` – string, name of the file (without `.def` suffix)

Description:

The suffix `.def` is appended to "name" and the file with this name is loaded. It should contain a list of functions, terminated by a closing brace } (the end-of-list delimiter). This tells the system to load the file "name" as soon as the user calls one of the functions named in the file (if not done so already). This allows for faster startup times, since not all of the rules databases need to be loaded, just the descriptions on which files to load for which functions.

See also: `Load`, `Use`, `DefaultDirectory`

`FindFile` — find a file in the current path

(Yacas internal)

Calling format:

```
FindFile(name)
```

Parameters:

`name` – string, name of the file or directory to find

Description:

The result of this command is the full path to the file that would be opened when the command `Load(name)` would be invoked. This means that the input directories are subsequently searched for a file called "name". If such a file is not found, `FindFile` returns an empty string.

`FindFile("")` returns the name of the default directory (the first one on the search path).

See also: `Load`, `DefaultDirectory`

`PatchLoad` — execute commands between `<?` and `?>` in file

(Yacas internal)

Calling format:

```
PatchLoad(name)
```

Parameters:

`name` – string, name of the file to "patch"

Description:

`PatchLoad` loads in a file and outputs the contents to the current output. The file can contain blocks delimited by `<?` and `?>` (meaning "Yacas Begin" and "Yacas End"). The piece of text between such delimiters is treated as a separate file with Yacas instructions, which is then loaded and executed. All output of write statements in that block will be written to the same current output.

This is similar to the way PHP works. You can have a static text file with dynamic content generated by Yacas.

See also: `PatchString`, `Load`

Nl — the newline character

(standard library)

Calling format:

```
Nl()
```

Description:

This function returns a string with one element in it, namely a newline character. This may be useful for building strings to send to some output in the end.

Note that the second letter in the name of this command is a lower case L (from "line").

Examples:

```
In> WriteString("First line" : Nl() : "Second line" : Nl());
First line
Second line
Out> True;
```

See also: NewLine

V, InVerboseMode — set verbose output mode

(standard library)

Calling format:

```
V(expression)
InVerboseMode()
```

Parameters:

`expression` – expression to be evaluated in verbose mode

Description:

The function `V(expression)` will evaluate the expression in verbose mode. Various parts of Yacas can show extra information about the work done while doing a calculation when using `V`.

In verbose mode, `InVerboseMode()` will return `True`, otherwise it will return `False`.

Examples:

```
In> OldSolve({x+2==0},{x})
Out> {{-2}};
In> V(OldSolve({x+2==0},{x}))
Entering OldSolve
From  x+2==0  it follows that  x  = -2
   x+2==0  simplifies to  True
Leaving OldSolve
Out> {{-2}};
In> InVerboseMode()
Out> False
In> V(InVerboseMode())
Out> True
```

See also: Echo, N, OldSolve

Plot2D — adaptive two-dimensional plotting

(standard library)

Calling format:

```
Plot2D(f(x))
Plot2D(f(x), a:b)
Plot2D(f(x), a:b, option=value)
Plot2D(f(x), a:b, option=value, ...)
Plot2D(list, ...)
```

Parameters:

`f(x)` – unevaluated expression containing one variables (function to be plotted)
`list` – list of functions to plot
`a, b` – numbers, plotting range in the x coordinate
`option` – atom, option name
`value` – atom, number or string (value of option)

Description:

The routine `Plot2D` performs adaptive plotting of one or several functions of one variable in the specified range. The result is presented as a line given by the equation $y = f(x)$. Several functions can be plotted at once. Various plotting options can be specified. Output can be directed to a plotting program (the default is to use `data`) to a list of values.

The function parameter `f(x)` must evaluate to a Yacas expression containing at most one variable. (The variable does not have to be called `x`.) Also, `N(f(x))` must evaluate to a real (not complex) numerical value when given a numerical value of the argument `x`. If the function `f(x)` does not satisfy these requirements, an error is raised.

Several functions may be specified as a list and they do not have to depend on the same variable, for example, `{f(x), g(y)}`. The functions will be plotted on the same graph using the same coordinate ranges.

If you have defined a function which accepts a number but does not accept an undefined variable, `Plot2D` will fail to plot it. Use `NFunction` to overcome this difficulty.

Data files are created in a temporary directory `/tmp/plot.tmp/` unless otherwise requested. File names and other information is printed if `InVerboseMode()` returns `True` on using `V()`.

The current algorithm uses Newton-Cotes quadratures and some heuristics for error estimation (see *The Yacas book of algorithms, Chapter 3, Section 1*). The initial grid of `points+1` points is refined between any grid points a, b if the integral $\int_a^b f(x)\,dx$ is not approximated to the given precision by the existing grid.

Default plotting range is `-5:5`. Range can also be specified as `x= -5:5` (note the mandatory space separating "`=`" and "`-`"); currently the variable name `x` is ignored in this case.

Options are of the form `option=value`. Currently supported option names are: "points", "precision", "depth", "output", "filename", "yrange". Option values are either numbers or special unevaluated atoms such as `data`. If you need to use the names of these atoms in your script, strings can be used. Several option/value pairs may be specified (the function `Plot2D` has a variable number of arguments).

- `yrange`: the range of ordinates to use for plotting, e.g. `yrange=0:20`. If no range is specified, the default is usually to leave the choice to the plotting backend.

- `points`: initial number of points (default 23) – at least that many points will be plotted. The initial grid of this many points will be adaptively refined.
- `precision`: graphing precision (default 10^{-6}). This is interpreted as the relative precision of computing the integral of $f(x) - \min(f(x))$ using the grid points. For a smooth, non-oscillating function this value should be roughly 1/(number of screen pixels in the plot).
- `depth`: max. refinement depth, logarithmic (default 5) – means there will be at most 2^{depth} extra points per initial grid point.
- `output`: name of the plotting backend. Supported names: `data` (default). The `data` backend will return the data as a list of pairs such as `{{x1,y1}, {x2,y2}, ...}`.
- `filename`: specify name of the created data file. For example: `filename="data1.txt"`. The default is the name `"output.data"`. Note that if several functions are plotted, the data files will have a number appended to the given name, for example `data.txt1`, `data.txt2`.

Other options may be supported in the future.

The current implementation can deal with a singularity within the plotting range only if the function `f(x)` returns `Infinity`, `-Infinity` or `Undefined` at the singularity. If the function `f(x)` generates a numerical error and fails at a singularity, `Plot2D` will fail if one of the grid points falls on the singularity. (All grid points are generated by bisection so in principle the endpoints and the `points` parameter could be chosen to avoid numerical singularities.)

*WIN32

See also: `V`, `NFunction`, `Plot3DS`

`Plot3DS` — three-dimensional (surface) plotting

(standard library)

Calling format:

```
Plot3DS(f(x,y))
Plot3DS(f(x,y), a:b, c:d)
Plot3DS(f(x,y), a:b, c:d, option=value)
Plot3DS(f(x,y), a:b, c:d, option=value, ...)
Plot3DS(list, ...)
```

Parameters:

`f(x,y)` – unevaluated expression containing two variables (function to be plotted)

`list` – list of functions to plot

`a`, `b`, `c`, `d` – numbers, plotting ranges in the x and y coordinates

`option` – atom, option name

`value` – atom, number or string (value of option)

Description:

The routine `Plot3DS` performs adaptive plotting of a function of two variables in the specified ranges. The result is presented as a surface given by the equation $z = f(x,y)$. Several functions can be plotted at once, by giving a list of functions. Various plotting options can be specified. Output can be directed to a plotting program (the default is to use `data`), to a list of values.

The function parameter `f(x,y)` must evaluate to a Yacas expression containing at most two variables. (The variables do not have to be called `x` and `y`.) Also, `N(f(x,y))` must evaluate to a real (not complex) numerical value when given numerical values of the arguments `x`, `y`. If the function `f(x,y)` does not satisfy these requirements, an error is raised.

Several functions may be specified as a list but they have to depend on the same symbolic variables, for example, `{f(x,y), g(y,x)}`, but not `{f(x,y), g(a,b)}`. The functions will be plotted on the same graph using the same coordinate ranges.

If you have defined a function which accepts a number but does not accept an undefined variable, `Plot3DS` will fail to plot it. Use `NFunction` to overcome this difficulty.

Data files are created in a temporary directory `/tmp/plot.tmp/` unless otherwise requested. File names and other information is printed if `InVerboseMode()` returns `True` on using `V()`.

The current algorithm uses Newton-Cotes cubatures and some heuristics for error estimation (see *The Yacas book of algorithms, Chapter 3, Section 1*). The initial rectangular grid of `xpoints+1*ypoints+1` points is refined within any rectangle where the integral of $f(x,y)$ is not approximated to the given precision by the existing grid.

Default plotting range is `-5:5` in both coordinates. A range can also be specified with a variable name, e.g. `x= -5:5` (note the mandatory space separating "`=`" and "`-`"). The variable name `x` should be the same as that used in the function `f(x,y)`. If ranges are not given with variable names, the first variable encountered in the function `f(x,y)` is associated with the first of the two ranges.

Options are of the form `option=value`. Currently supported option names are "points", "xpoints", "ypoints", "precision", "depth", "output", "filename", "xrange", "yrange", "zrange". Option values are either numbers or special unevaluated atoms such as `data`. If you need to use the names of these atoms in your script, strings can be used (e.g. `output="data"`). Several option/value pairs may be specified (the function `Plot3DS` has a variable number of arguments).

- `xrange`, `yrange`: optionally override coordinate ranges. Note that `xrange` is always the first variable and `yrange` the second variable, regardless of the actual variable names.
- `zrange`: the range of the z axis to use for plotting, e.g. `zrange=0:20`. If no range is specified, the default is usually to leave the choice to the plotting backend. Automatic choice based on actual values may give visually inadequate plots if the function has a singularity.
- `points`, `xpoints`, `ypoints`: initial number of points (default 10 each) – at least that many points will be plotted in each coordinate. The initial grid of this many points will be adaptively refined. If `points` is specified, it serves as a default for both `xpoints` and `ypoints`; this value may be overridden by `xpoints` and `ypoints` values.
- `precision`: graphing precision (default 0.01). This is interpreted as the relative precision of computing the integral of $f(x,y) - \min(f(x,y))$ using the grid points. For a smooth, non-oscillating function this value should be roughly 1/(number of screen pixels in the plot).
- `depth`: max. refinement depth, logarithmic (default 3) – means there will be at most 2^{depth} extra points per initial grid point (in each coordinate).
- `output`: name of the plotting backend. Supported names: `data` (default). The `data` backend will return the data as a list of triples such as `{{x1, y1, z1}, {x2, y2, z2}, ...}`.

Other options may be supported in the future.

The current implementation can deal with a singularity within the plotting range only if the function `f(x,y)` returns `Infinity`, `-Infinity` or `Undefined` at the singularity. If the function `f(x,y)` generates a numerical error and fails at a singularity, `Plot3DS` will fail only if one of the grid points falls on the singularity. (All grid points are generated by bisection so in principle the endpoints and the `xpoints`, `ypoints` parameters could be chosen to avoid numerical singularities.)

The `filename` option is optional if using graphical backends, but can be used to specify the location of the created data file.

*WIN32

Same limitations as `Plot2D`.

Examples:

```
In> Plot3DS(a*b^2)
Out> True;
In> V(Plot3DS(Sin(x)*Cos(y),x=0:20, y=0:20,depth=3))
CachedConstant: Info: constant Pi is being
  recalculated at precision 10
CachedConstant: Info: constant Pi is being
  recalculated at precision 11
Plot3DS: using 1699  points for function Sin(x)*Cos(y)
Plot3DS: max. used 8 subdivisions for Sin(x)*Cos(y)
Plot3DS'datafile: created file '/tmp/plot.tmp/data1'
Out> True;
```

See also: `V`, `NFunction`, `Plot2D`

`XmlExplodeTag` — convert XML strings to tag objects

(Yacas internal)

Calling format:

```
XmlExplodeTag(xmltext)
```

Parameters:

`xmltext` – string containing some XML tokens

Description:

`XmlExplodeTag` parses the first XML token in `xmltext` and returns a Yacas expression.

The following subset of XML syntax is supported currently:

- `<TAG [options]>` – an opening tag
- `</TAG [options]>` – a closing tag
- `<TAG [options] />` – an open/close tag
- plain (non-tag) text

The tag options take the form `paramname="value"`.

If given an XML tag, `XmlExplodeTag` returns a structure of the form `XmlTag(name,params,type)`. In the returned object, `name` is the (capitalized) tag name, `params` is an assoc list with the options (key fields capitalized), and type can be either "Open", "Close" or "OpenClose".

If given a plain text string, the same string is returned.

Examples:

```
In> XmlExplodeTag("some plain text")
Out> "some plain text";
In> XmlExplodeTag("<a name=\"blah blah\"
  align=\"left\">")
Out> XmlTag("A",{{"ALIGN","left"},
  {"NAME","blah blah"}},"Open");
In> XmlExplodeTag("</p>")
Out> XmlTag("P",{},"Close");
In> XmlExplodeTag("<br/>")
Out> XmlTag("BR",{},"OpenClose");
```

See also: `XmlTokenizer`

`DefaultTokenizer` — select the default syntax tokenizer for parsing the input

`XmlTokenizer` — select an XML syntax tokenizer for parsing the input

(Yacas internal)

Calling format:

```
DefaultTokenizer()
XmlTokenizer()
```

Description:

A "tokenizer" is an internal routine in the kernel that parses the input into Yacas expressions. This affects all input typed in by a user at the prompt and also the input redirected from files or strings using `FromFile` and `FromString` and read using `Read` or `ReadToken`.

The Yacas environment currently supports some experimental tokenizers for various syntaxes. `DefaultTokenizer` switches to the tokenizer used for default Yacas syntax. `XmlTokenizer` switches to an XML syntax. Note that setting the tokenizer is a global side effect. One typically needs to switch back to the default tokenizer when finished reading the special syntax.

Care needs to be taken when kernel errors are raised during a non-default tokenizer operation (as with any global change in the environment). Errors need to be caught with the `TrapError` function. The error handler code should re-instate the default tokenizer, or else the user will be unable to continue the session (everything a user types will be parsed using a non-default tokenizer).

When reading XML syntax, the supported formats are the same as those of `XmlExplodeTag`. The parser does not validate anything in the XML input. After an XML token has been read in, it can be converted into an Yacas expression with `XmlExplodeTag`. Note that when reading XML, any plain text between tags is returned as one token. Any malformed XML will be treated as plain text.

Example:

```
In> [XmlTokenizer(); q:=ReadToken(); \
  DefaultTokenizer();q;]
<a>Out> <a>;
```

Note that:

1. after switching to `XmlTokenizer` the `In>` prompt disappeared; the user typed `<a>` and the `Out>` prompt with the resulting expression appeared.

2. The resulting expression is an atom with the string representation <a>; it is *not* a string.

See also: `OMRead`, `TrapError`, `XmlExplodeTag`, `ReadToken`, `FromFile`, `FromString`

`OMForm` — convert Yacas expression to OpenMath

`OMRead` — convert expression from OpenMath to Yacas expression

(standard library)

Calling format:

```
OMForm(expression)
OMRead()
```

Parameters:

`expression` – expression to convert

Description:

`OMForm` prints an OpenMath representation of the input parameter `expression` to standard output. `OMRead` reads an OpenMath expression from standard input and returns a normal Yacas expression that matches the input OpenMath expression.

If a Yacas symbol does not have a mapping defined by `OMDef`, it is translated to and from OpenMath as the OpenMath symbol in the CD "yacas" with the same name as it has in Yacas.

Example:

```
In> str:=ToString()OMForm(2+Sin(a*3))
Out> "<OMOBJ>
  <OMA>
    <OMS cd="arith1" name="plus"/>
    <OMI>2</OMI>
    <OMA>
      <OMS cd="transc1" name="sin"/>
      <OMA>
        <OMS cd="arith1" name="times"/>
        <OMV name="a"/>
        <OMI>3</OMI>
      </OMA>
    </OMA>
  </OMA>
</OMOBJ>
";
In> FromString(str)OMRead()
Out> 2+Sin(a*3);
```

```
In> OMForm(NotDefinedInOpenMath(2+3))
<OMOBJ>
  <OMA>
    <OMS cd="yacas" name="NotDefinedInOpenMath"/>
    <OMA>
      <OMS cd="arith1" name="plus"/>
      <OMI>2</OMI>
      <OMI>3</OMI>
    </OMA>
  </OMA>
</OMOBJ>
Out> True
```

See also: `XmlTokenizer`, `XmlExplodeTag`, `OMDef`

`OMDef` — define translations from Yacas to OpenMath and vice-versa.

(standard library)

Calling format:

```
OMDef(yacasForm, cd, name)
OMDef(yacasForm, cd, name, yacasToOM)
OMDef(yacasForm, cd, name, yacasToOM, omToYacas)
```

Parameters:

`yacasForm` – string with the name of a Yacas symbol, or a Yacas expression

`cd` – OpenMath Content Dictionary for the symbol

`name` – OpenMath name for the symbol

`yacasToOM` – rule for translating an application of that symbol in Yacas into an OpenMath expression

`omToYacas` – rule for translating an OpenMath expression into an application of this symbol in Yacas

Description:

`OMDef` defines the translation rules for symbols between the Yacas representation and `OpenMath`. The first parameter, `yacasForm`, can be a string or an expression. The difference is that when giving an expression only the `omToYacas` translation is defined, and it uses the exact expression given. This is used for `OpenMath` symbols that must be translated into a whole subexpression in Yacas, such as `set1:emptyset` which gets translated to an empty list as follows:

```
In> OMDef( {}, "set1","emptyset" )
Out> True
In> FromString("<OMOBJ><OMS cd=\"set1\" name=\"emptyset\"/
Out> {}
In> IsList(%)
Out> True
```

Otherwise, a symbol that is not inside an application (OMA) gets translated to the Yacas atom with the given name:

```
In> OMDef( "EmptySet", "set1","emptyset" )
Warning: the mapping for set1:emptyset was already defined
Out> True
In> FromString("<OMOBJ><OMS cd=\"set1\" name=\"emptyset\"/
Out> EmptySet
```

The definitions for the symbols in the Yacas library are in the `*.rep` script subdirectories. In those modules for which the mappings are defined, there is a file called `om.ys` that contains the `OMDef` calls. Those files are loaded in `openmath.rep/om.ys`, so any new file must be added to the list there, at the end of the file.

A rule is represented as a list of expressions. Since both OM and Yacas expressions are actually lists, the syntax is the same in both directions. There are two template forms that are expanded before the translation:

- `$`: this symbol stands for the translation of the symbol applied in the original expression.
- `_path`: a path into the original expression (list) to extract an element, written as an underscore applied to an integer or a list of integers. Those integers are indexes into expressions, and integers in a list are applied recursively starting at the original expression. For example, `_2` means the second parameter of the expression, while `_{3,2,1}` means the first parameter of the second parameter of the third parameter of the original expression.

They can appear anywhere in the rule as expressions or subexpressions.

Finally, several alternative rules can be specified by joining them with the | symbol, and each of them can be annotated with a post-predicate applied with the underscore _ symbol, in the style of Yacas' simplification rules. Only the first alternative rule that matches is applied, so the more specific rules must be written first.

There are special symbols recognized by `OMForm` to output `OpenMath` constructs that have no specific parallel in Yacas, such as an OpenMath symbol having a `CD` and `name`: Yacas symbols have only a name. Those special symbols are:

- `OMS(cd, name)`: <OMS cd="cd" name="name">
- `OMA(f x y ...)`: <OMA>f x y ...</OMA>
- `OMBIND(binderSymbol, bvars, expression)`: <OMBIND>binderSymbol bvars expression</OMBIND>, where `bvars` must be produced by using `OMBVAR(...)`.
- `OMBVAR(x y ...)`: <OMBVAR>x y ...</OMBVAR>
- `OME(...)`: <OME>...</OME>

When translating from OpenMath to Yacas, we just store unknown symbols as `OMS("cd", "name")`. This way we don't have to bother defining bogus symbols for concepts that Yacas does not handle, and we can evaluate expressions that contain them.

Examples:

```
In> OMDef( "Sqrt" ,  "arith1", "root", { $, _1, 2 }, $(_1)^(_2=2) | (_1^(1/_2)) );
Out> True
In> OMForm(Sqrt(3))
<OMOBJ>
  <OMA>
    <OMS cd="arith1" name="root"/>
    <OMI>3</OMI>
    <OMI>2</OMI>
  </OMA>
</OMOBJ>
Out> True
In> FromString("<OMOBJ><OMA><OMS cd=\"arith1\" name=\"root\"/><OMI>16</OMI><OMI>2</OMI></OMA></OMOBJ> ")OMRead()
Out> Sqrt(16)
In> FromString("<OMOBJ><OMA><OMS cd=\"arith1\" name=\"root\"/><OMI>16</OMI><OMI>3</OMI></OMA></OMOBJ> ")OMRead()
Out> 16^(1/3)

In> OMDef("Limit", "limit1", "limit", \
      {  $, _2, OMS("limit1", "under"), OMBIND(OMS("fns1", "lambda"), OMBVAR(_1), _4) }_(_3=Left)  \
      |{ $, _2, OMS("limit1", "above"), OMBIND(OMS("fns1", "lambda"), OMBVAR(_1), _4) }_(_3=Right) \
      |{ $, _2, OMS("limit1", "both_sides"), OMBIND(OMS("fns1", "lambda"), OMBVAR(_1), _3) },      \
      { $, _{3,2,1}, _1, Left,  _{3,3}}_(_2=OMS("limit1", "below")) \
      |{$, _{3,2,1}, _1, Right, _{3,3}}_(_2=OMS("limit1", "above")) \
      |{$, _{3,2,1}, _1, _{3,3}}                                    \
     );
In> OMForm(Limit(x,0) Sin(x)/x)
<OMOBJ>
  <OMA>
    <OMS cd="limit1" name="limit"/>
    <OMI>0</OMI>
    <OMS cd="limit1" name="both_sides"/>
    <OMBIND>
      <OMS cd="fns1" name="lambda"/>
      <OMBVAR>
        <OMV name="x"/>
      </OMBVAR>
      <OMA>
        <OMS cd="arith1" name="divide"/>
        <OMA>
          <OMS cd="transc1" name="sin"/>
          <OMV name="x"/>
        </OMA>
        <OMV name="x"/>
      </OMA>
    </OMBIND>
  </OMA>
</OMOBJ>
Out> True
In> OMForm(Limit(x,0,Right) 1/x)
<OMOBJ>
  <OMA>
    <OMS cd="limit1" name="limit"/>
    <OMI>0</OMI>
    <OMS cd="limit1" name="above"/>
    <OMBIND>
      <OMS cd="fns1" name="lambda"/>
      <OMBVAR>
        <OMV name="x"/>
      </OMBVAR>
      <OMA>
        <OMS cd="arith1" name="divide"/>
        <OMI>1</OMI>
        <OMV name="x"/>
      </OMA>
    </OMBIND>
  </OMA>
</OMOBJ>
Out> True
In> FromString(ToString()OMForm(Limit(x,0,Right) 1/x))OMRea
Out> Limit(x,0,Right)1/x
In> %
Out> Infinity
```

See also: `OMForm`, `OMRead`

Chapter 29

String manipulation

`StringMid'Set` — change a substring

(Yacas internal)

Calling format:

```
StringMid'Set(index,substring,string)
```

Parameters:

`index` – index of substring to get
`substring` – substring to store
`string` – string to store substring in.

Description:

Set (change) a part of a string. It leaves the original alone, returning a new changed copy.

Examples:

```
In> StringMid'Set(3,"XY","abcdef")
Out> "abXYef";
```

See also: `StringMid'Get`, `Length`

`StringMid'Get` — retrieve a substring

(Yacas internal)

Calling format:

```
StringMid'Get(index,length,string)
```

Parameters:

`index` – index of substring to get
`length` – length of substring to get
`string` – string to get substring from

Description:

`StringMid'Get` returns a part of a string. Substrings can also be accessed using the `[]` operator.

Examples:

```
In> StringMid'Get(3,2,"abcdef")
Out> "cd";
In> "abcdefg"[2 .. 4]
Out> "bcd";
```

See also: `StringMid'Set`, `Length`

`String` — convert atom to string

`Atom` — convert string to atom

(Yacas internal)

Calling format:

```
Atom("string")
String(atom)
```

Parameters:

`atom` – an atom
`"string"` – a string

Description:

Returns an atom with the string representation given as the evaluated argument. Example: `Atom("foo");` returns `foo`.
`String` is the inverse of `Atom`: turns `atom` into `"atom"`.

Examples:

```
In> String(a)
Out> "a";
In> Atom("a")
Out> a;
```

`ConcatStrings` — concatenate strings

(Yacas internal)

Calling format:

```
ConcatStrings(strings)
```

Parameters:

`strings` – one or more strings

Description:

Concatenates strings.

Examples:

```
In> ConcatStrings("a","b","c")
Out> "abc";
```

See also: `:`

`PatchString` — execute commands between `<?` and `?>` in strings

(Yacas internal)

Calling format:

```
PatchString(string)
```

Parameters:

`string` – a string to patch

Description:

This function does the same as PatchLoad, but it works on a string in stead of on the contents of a text file. See PatchLoad for more details.

Examples:

```
In> PatchString("Two plus three \
  is <? Write(2+3); ?> ");
Out> "Two plus three is 5 ";
```

See also: `PatchLoad`

Chapter 30

Probability and Statistics

30.1 Probability

Each distribution is represented as an entity. For each distribution known to the system the consistency of parameters is checked. If the parameters for a distribution are invalid, the functions return `Undefined`. For example, `NormalDistribution(a,-1)` evaluates to `Undefined`, because of negative variance.

`BernoulliDistribution` — Bernoulli distribution

(standard library)

Calling format:

```
BernoulliDistribution(p)
```

Parameters:

`p` – number, probability of an event in a single trial

Description:

A random variable has a Bernoulli distribution with probability `p` if it can be interpreted as an indicator of an event, where `p` is the probability to observe the event in a single trial.

Numerical value of `p` must satisfy $0 < p < 1$.

See also: `BinomialDistribution`

`BinomialDistribution` — binomial distribution

(standard library)

Calling format:

```
BinomialDistribution(p,n)
```

Parameters:

`p` – number, probability to observe an event in single trial

`n` – number of trials

Description:

Suppose we repeat a trial `n` times, the probability to observe an event in a single trial is `p` and outcomes in all trials are mutually independent. Then the number of trials when the event occurred is distributed according to the binomial distribution. The probability of that is `BinomialDistribution(p,n)`.

Numerical value of `p` must satisfy $0 < p < 1$. Numerical value of `n` must be a positive integer.

See also: `BernoulliDistribution`

`tDistribution` — Student's t distribution

(standard library)

Calling format:

```
{tDistribution}(m)
```

Parameters:

`m` – integer, number of degrees of freedom

Description:

Let Y and Z be independent random variables, Y have the NormalDistribution(0,1), `Z` have ChiSquareDistribution(m). Then $\frac{Y}{\sqrt{\frac{Z}{m}}}$ has tDistribution(m).

Numerical value of `m` must be positive integer.

`PDF` — probability density function

(standard library)

Calling format:

```
PDF(dist,x)
```

Parameters:

`dist` – a distribution type

`x` – a value of random variable

Description:

If `dist` is a discrete distribution, then `PDF` returns the probability for a random variable with distribution `dist` to take a value of `x`. If `dist` is a continuous distribution, then `PDF` returns the density function at point x.

See also: `CDF`

30.2 Statistics

`ChiSquareTest` — Pearson's ChiSquare test

(standard library)

Calling format:

```
ChiSquareTest(observed,expected)
ChiSquareTest(observed,expected,params)
```

Parameters:

`observed` – list of observed frequencies
`expected` – list of expected frequencies
`params` – number of estimated parameters

Description:

`ChiSquareTest` is intended to find out if our sample was drawn from a given distribution or not. To find this out, one has to calculate observed frequencies into certain intervals and expected ones. To calculate expected frequency the formula $n_i \equiv np_i$ must be used, where p_i is the probability measure of i-th interval, and n is the total number of observations. If any of the parameters of the distribution were estimated, this number is given as `params`.

The function returns a list of three local substitution rules. First of them contains the test statistic, the second contains the value of the parameters, and the last one contains the degrees of freedom.

The test statistic is distributed as ChiSquareDistribution.

Chapter 31

Number theory

This chapter describes functions that are of interest in number theory. These functions typically operate on integers. Some of these functions work quite slowly.

IsPrime — test for a prime number

IsSmallPrime — test for a (small) prime number

(standard library)

Calling format:

```
IsPrime(n)
IsSmallPrime(n)
```

Parameters:

`n` – integer to test

Description:

The commands checks whether n, which should be a positive integer, is a prime number. A number n is a prime number if it is only divisible by 1 and itself. As a special case, 1 is not considered a prime number. The first prime numbers are 2, 3, 5, ...

The function `IsShortPrime` only works for numbers $n \leq 65537$ but it is very fast.

The function `IsPrime` operates on all numbers and uses different algorithms depending on the magnitude of the number n. For small numbers $n \leq 65537$, a constant-time table lookup is performed. (The function `IsShortPrime` is used for that.) For numbers n between 65537 and 34155071728321, the function uses the Rabin-Miller test together with table lookups to guarantee correct results.

For even larger numbers a version of the probabilistic Rabin-Miller test is executed. The test can sometimes mistakenly mark a number as prime while it is in fact composite, but a prime number will never be mistakenly declared composite. The parameters of the test are such that the probability for a false result is less than 10^{-24}.

Examples:

```
In> IsPrime(1)
Out> False;
In> IsPrime(2)
Out> True;
In> IsPrime(10)
Out> False;
In> IsPrime(23)
Out> True;
In> Select("IsPrime", 1 .. 100)
Out> {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
  53,59,61,67,71,73,79,83,89,97};
```

See also: `IsPrimePower`, `Factors`

IsComposite — test for a composite number

(standard library)

Calling format:

```
IsComposite(n)
```

Parameters:

`n` – positive integer

Description:

This function is the logical negation of `IsPrime`, except for the number 1, which is neither prime nor composite.

Examples:

```
In> IsComposite(1)
Out> False;
In> IsComposite(7)
Out> False;
In> IsComposite(8)
Out> True;
In> Select(IsComposite,1 .. 20)
Out> {4,6,8,9,10,12,14,15,16,18,20};
```

See also: `IsPrime`

IsCoprime — test if integers are coprime

(standard library)

Calling format:

```
IsCoprime(m,n)
IsCoprime(list)
```

Parameters:

m,n – positive integers
list – list of positive integers

Description:

This function returns True if the given pair or list of integers are coprime, also called relatively prime. A pair or list of numbers are coprime if they share no common factors.

Examples:

```
In> IsCoprime({3,4,5,8})
Out> False;
In> IsCoprime(15,17)
Out> True;
```

See also: Prime

IsSquareFree — test for a square-free number

(standard library)

Calling format:

```
IsSquareFree(n)
```

Parameters:

n – positive integer

Description:

This function uses the Moebius function to tell if the given number is square-free, which means it has distinct prime factors. If $\mathrm{Moebius}(n) \neq 0$, then n is square free. All prime numbers are trivially square-free.

Examples:

```
In> IsSquareFree(37)
Out> True;
In> IsSquareFree(4)
Out> False;
In> IsSquareFree(16)
Out> False;
In> IsSquareFree(18)
Out> False;
```

See also: Moebius, SquareFreeDivisorsList

IsPrimePower — test for a power of a prime number

(standard library)

Calling format:

```
IsPrimePower(n)
```

Parameters:

n – integer to test

Description:

This command tests whether "n", which should be a positive integer, is a prime power, that is whether it is of the form p^m, with "p" prime and "m" an integer.

This function does not try to decompose the number n into factors. Instead we check for all prime numbers $r = 2, 3, \ldots$ that the r-th root of n is an integer, and we find such r and m that $n = m^r$, we check that m is a prime. If it is not a prime, we execute the same function call on m.

Examples:

```
In> IsPrimePower(9)
Out> True;
In> IsPrimePower(10)
Out> False;
In> Select("IsPrimePower", 1 .. 50)
Out> {2,3,4,5,7,8,9,11,13,16,17,19,23,25,27,
  29,31,32,37,41,43,47,49};
```

See also: IsPrime, Factors

NextPrime — generate a prime following a number

(standard library)

Calling format:

```
NextPrime(i)
```

Parameters:

i – integer value

Description:

The function finds the smallest prime number that is greater than the given integer value.

The routine generates "candidate numbers" using the formula $n + 2(-n) \bmod 3$ where n is an odd number (this generates the sequence 5, 7, 11, 13, 17, 19, ...) and IsPrime() to test whether the next candidate number is in fact prime.

Example:

```
In> NextPrime(5)
Out> 7;
```

See also: IsPrime

IsTwinPrime — test for a twin prime

(standard library)

Calling format:

```
IsTwinPrime(n)
```

Parameters:

n – positive integer

Description:

This function returns True if n is a twin prime. By definition, a twin prime is a prime number n such that $n+2$ is also a prime number.

Examples:

```
In> IsTwinPrime(101)
Out> True;
In> IsTwinPrime(7)
Out> False;
In> Select(IsTwinPrime, 1 .. 100)
Out> {3,5,11,17,29,41,59,71};
```

See also: IsPrime

IsIrregularPrime — test for an irregular prime

(standard library)

Calling format:

```
IsIrregularPrime(n)
```

Parameters:

n – positive integer

Description:

This function returns True if n is an irregular prime. A prime number n is irregular if and only if n divides the numerator of a Bernoulli number B_{2i}, where $2i+1 < n$. Small irregular primes are quite rare; the only irregular primes under 100 are 37, 59 and 67. Asymptotically, roughly 40% of primes are irregular.

Examples:

```
In> IsIrregularPrime(5)
Out> False;
In> Select(IsIrregularPrime,1 .. 100)
Out> {37,59,67};
```

See also: IsPrime

IsCarmichaelNumber — test for a Carmichael number

(standard library)

Calling format:

```
IsCarmichaelNumber(n)
```

Parameters:

n – positive integer

Description:

This function returns True if n is a Carmichael number, also called an absolute pseudoprime. They have the property that $b^{n-1} \bmod n = 1$ for all b satisfying $\mathrm{Gcd}(b, n) = 1$. These numbers cannot be proved composite by Fermat's little theorem. Because the previous property is extremely slow to test, the following equivalent property is tested by Yacas: for all prime factors p_i of n, $(n-1) \bmod (p_i - 1) = 0$ and n must be square free. Also, Carmichael numbers must be odd and have at least three prime factors. Although these numbers are rare (there are only 43 such numbers between 1 and 10^6), it has recently been proven that there are infinitely many of them.

Examples:

```
In> IsCarmichaelNumber(561)
Out> True;
In> Time(Select(IsCarmichaelNumber,1 .. 10000))
504.19 seconds taken
Out> {561,1105,1729,2465,2821,6601,8911};
```

See also: IsSquareFree, IsComposite

Factors — factorization

(standard library)

Calling format:

```
Factors(x)
```

Parameters:

x – integer or univariate polynomial

Description:

This function decomposes the integer number x into a product of numbers. Alternatively, if x is a univariate polynomial, it is decomposed in irreducible polynomials.

The factorization is returned as a list of pairs. The first member of each pair is the factor, while the second member denotes the power to which this factor should be raised. So the factorization $x = p_1^{n_1} ... p_9^{n_9}$ is returned as {{p1,n1}, ..., {p9,n9}}.

Examples:

```
In> Factors(24);
Out> {{2,3},{3,1}};
In> Factors(2*x^3 + 3*x^2 - 1);
Out> {{2,1},{x+1,2},{x-1/2,1}};
```

See also: Factor, IsPrime, GaussianFactors

IsAmicablePair — test for a pair of amicable numbers

(standard library)

Calling format:

```
IsAmicablePair(m,n)
```

Parameters:

m, n – positive integers

Description:

This function tests if a pair of numbers are amicable. A pair of numbers m, n has this property if the sum of the proper divisors of m is n and the sum of the proper divisors of n is m.

Examples:

```
In> IsAmicablePair(200958394875, 209194708485 )
Out> True;
In> IsAmicablePair(220, 284)
Out> True;
```

See also: `ProperDivisorsSum`

`Factor` — factorization, in pretty form

(standard library)

Calling format:

```
Factor(x)
```

Parameters:

`x` – integer or univariate polynomial

Description:

This function factorizes "x", similarly to `Factors`, but it shows the result in a nicer human readable format.

Examples:

```
In> PrettyForm(Factor(24));

 3
2  * 3

Out> True;
In> PrettyForm(Factor(2*x^3 + 3*x^2 - 1));

             2   /     1 \
2 * ( x + 1 )  * | x - - |
                 \     2 /

Out> True;
```

See also: `Factors`, `IsPrime`, `PrettyForm`

`Divisors` — number of divisors

(standard library)

Calling format:

```
Divisors(n)
```

Parameters:

`n` – positive integer

Description:

`Divisors` returns the number of positive divisors of a number. A number is prime if and only if it has two divisors, 1 and itself.

Examples:

```
In> Divisors(180)
Out> 18;
In> Divisors(37)
Out> 2;
```

See also: `DivisorsSum`, `ProperDivisors`, `ProperDivisorsSum`, `Moebius`

`DivisorsSum` — the sum of divisors

(standard library)

Calling format:

```
DivisorsSum(n)
```

Parameters:

`n` – positive integer

Description:

`DivisorsSum` returns the sum all numbers that divide it. A number `n` is prime if and only if the sum of its divisors are `n+1`.

Examples:

```
In> DivisorsSum(180)
Out> 546;
In> DivisorsSum(37)
Out> 38;
```

See also: `DivisorsSum`, `ProperDivisors`, `ProperDivisorsSum`, `Moebius`

`ProperDivisors` — the number of proper divisors

(standard library)

Calling format:

```
ProperDivisors(n)
```

Parameters:

`n` – positive integer

Description:

`ProperDivisors` returns the number of proper divisors, i.e `Divisors(n)-1`, since `n` is not counted. An integer n is prime if and only if it has 1 proper divisor.

Examples:

```
In> ProperDivisors(180)
Out> 17;
In> ProperDivisors(37)
Out> 1;
```

See also: `DivisorsSum`, `ProperDivisors`, `ProperDivisorsSum`, `Moebius`

ProperDivisorsSum — the sum of proper divisors

(standard library)

Calling format:

```
ProperDivisorsSum(n)
```

Parameters:

`n` – positive integer

Description:

`ProperDivisorsSum` returns the sum of proper divisors, i.e. `ProperDivisors(n)-n`, since `n` is not counted. `n` is prime if and only if `ProperDivisorsSum(n)==1`.

Examples:

```
In> ProperDivisorsSum(180)
Out> 366;
In> ProperDivisorsSum(37)
Out> 1;
```

See also: `DivisorsSum`, `ProperDivisors`, `ProperDivisorsSum`, `Moebius`

Moebius — the Moebius function

(standard library)

Calling format:

```
Moebius(n)
```

Parameters:

`n` – positive integer

Description:

The Moebius function is 0 when a prime factor is repeated (which means it is not square-free) and is $(-1)^r$ if n has r distinct factors. Also, $\mathrm{Moebius}(1) = 1$.

Examples:

```
In> Moebius(10)
Out> 1;
In> Moebius(11)
Out> -1;
In> Moebius(12)
Out> 0;
In> Moebius(13)
Out> -1;
```

See also: `DivisorsSum`, `ProperDivisors`, `ProperDivisorsSum`, `MoebiusDivisorsList`

CatalanNumber — return the nth Catalan Number

(standard library)

Calling format:

```
CatalanNumber(n)
```

Parameters:

`n` – positive integer

Description:

This function returns the `n`-th Catalan number, defined as $\frac{\binom{2n}{n}}{n+1}$.

Examples:

```
In> CatalanNumber(10)
Out> 16796;
In> CatalanNumber(5)
Out> 42;
```

See also: `Bin`

FermatNumber — return the nth Fermat Number

(standard library)

Calling format:

```
FermatNumber(n)
```

Parameters:

`n` – positive integer

Description:

This function returns the `n`-th Fermat number, which is defined as $2^{2^n} + 1$.

Examples:

```
In> FermatNumber(7)
Out> 340282366920938463463374607431768211457;
```

See also: `Factor`

HarmonicNumber — return the nth Harmonic Number

(standard library)

Calling format:

```
HarmonicNumber(n)
HarmonicNumber(n,r)
```

Parameters:

`n`, `r` – positive integers

Description:

This function returns the `n`-th Harmonic number, which is defined as $\sum_{k=1}^{n} \frac{1}{k}$. If given a second argument, the Harmonic number of order r is returned, which is defined as $\sum_{k=1}^{n} k^{-r}$.

Examples:

```
In> HarmonicNumber(10)
Out> 7381/2520;
In> HarmonicNumber(15)
Out> 1195757/360360;
In> HarmonicNumber(1)
Out> 1;
In> HarmonicNumber(4,3)
Out> 2035/1728;
```

See also: `Sum`

`StirlingNumber1` — return the `n,mth` Stirling Number of the first kind

(standard library)

Calling format:

```
StirlingNumber1(n,m)
```

Parameters:

`n`, `m` – positive integers

Description:

This function returns the signed Stirling Number of the first kind. All Stirling Numbers are integers. If $m > n$, then `StirlingNumber1` returns 0.

Examples:

```
In> StirlingNumber1(10,5)
Out> -269325;
In> StirlingNumber1(3,6)
Out> 0;
```

See also: `StirlingNumber2`

`StirlingNumber2` — return the `n,mth` Stirling Number of the second kind

(standard library)

Calling format:

StirlingNumber1(n,m)

Parameters:

`n`, `m` – positive integers

Description:

This function returns the Stirling Number of the second kind. All Stirling Numbers are positive integers. If $m > n$, then `StirlingNumber2` returns 0.

Examples:

```
In> StirlingNumber2(3,6)
Out> 0;
In> StirlingNumber2(10,4)
Out> 34105;
```

See also: `StirlingNumber1`

`DivisorsList` — the list of divisors

(standard library)

Calling format:

```
DivisorsList(n)
```

Parameters:

`n` – positive integer

Description:

`DivisorsList` creates a list of the divisors of n. This is useful for loops like

```
ForEach(d,DivisorsList(n))
```

Examples:

```
In> DivisorsList(18)
Out> {1,2,3,6,9,18};
```

See also: `DivisorsSum`

`SquareFreeDivisorsList` — the list of square-free divisors

(standard library)

Calling format:

```
SquareFreeDivisorsList(n)
```

Parameters:

`n` – positive integer

Description:

`SquareFreeDivisorsList` creates a list of the square-free divisors of n. Square-free numbers are numbers that have only simple prime factors (no prime powers). For example, $18 = 2 \cdot 3 \cdot 3$ is not square-free because it contains a square of 3 as a factor.

Examples:

```
In> SquareFreeDivisorsList(18)
Out> {1,2,3,6};
```

See also: `DivisorsList`

`MoebiusDivisorsList` — the list of divisors and Moebius values

(standard library)

Calling format:

```
MoebiusDivisorsList(n)
```

Parameters:

`n` – positive integer

Description:

Returns a list of pairs of the form {d,m}, where d runs through the squarefree divisors of n and $m = \mathrm{Moebius}(d)$. This is more efficient than making a list of all square-free divisors of n and then computing Moebius on each of them. It is useful for computing the cyclotomic polynomials. It can be useful in other computations based on the Moebius inversion formula.

Examples:

```
In> MoebiusDivisorsList(18)
Out> {{1,1},{2,-1},{3,-1},{6,1}};
```

See also: DivisorsList, Moebius

SumForDivisors — loop over divisors

(standard library)

Calling format:

```
SumForDivisors(var,n,expr)
```

Parameters:

var – atom, variable name
n – positive integer
expr – expression depending on var

Description:

This function performs the sum of the values of the expression expr while the variable var runs through the divisors of n. For example, SumForDivisors(d, 10, d2̂) sums d^2 where d runs through the divisors of 10. This kind of computation is frequently used in number theory.

See also: DivisorsList

RamanujanSum — compute the "Ramanujan sum"

(standard library)

Calling format:

```
RamanujanSum(k,n)
```

Parameters:

k, n – positive integers

Description:

This function computes the Ramanujan sum, i.e. the sum of the n-th powers of the k-th primitive roots of the unit:

$$\sum_{l=1}^{k} \exp\left(2\pi\imath\frac{ln}{k}\right)$$

where l runs thought the integers between 1 and $k-1$ that are coprime to l.

The computation is done by using the formula in T. M. Apostol, *Introduction to Analytic Theory* (Springer-Verlag), Theorem 8.6.

Cyclotomic — construct the cyclotomic polynomial

(standard library)

Calling format:

```
Cyclotomic(n,x)
```

Parameters:

n – positive integer
x – variable

Description:

Returns the cyclotomic polynomial in the variable x (which is the minimal polynomial of the n-th primitive roots of the unit, over the field of rational numbers).

For n even, we write $n = mk$, where k is a power of 2 and m is odd, and reduce it to the case of even m since

$$\mathrm{Cyclotomic}(n,x) = \mathrm{Cyclotomic}\left(m, -x^{\frac{k}{2}}\right).$$

If $m = 1$, n is a power of 2, and $\mathrm{Cyclotomic}(n,x) = x^k + 1$.

For n odd, the algorithm is based on the formula

$$\mathrm{Cyclotomic}(n,x) \equiv \mathrm{Prod}\left(\left(x^{\frac{n}{d}} - 1\right)^{\mu(d)}\right),$$

where d runs through the divisors of n. In order to compute this in a efficient way, we use the function MoebiusDivisorsList. Then we compute in poly1 the product of $x^{\frac{n}{d}} - 1$ with $\mu(d) = 1$, and in poly2 the product of these polynomials with $\mu(d) = -1$. Finally we compute the quotient poly1/poly2.

See also: RamanujanSum

PAdicExpand — p-adic expansion

(standard library)

Calling format:

```
PAdicExpand(n, p)
```

Parameters:

n – number or polynomial to expand
p – base to expand in

Description:

This command computes the p-adic expansion of n. In other words, n is expanded in powers of p. The argument n can be either an integer or a univariate polynomial. The base p should be of the same type.

Examples:

```
In> PrettyForm(PAdicExpand(1234, 10));

               2     3
3 * 10 + 2 * 10  + 10  + 4

Out> True;
In> PrettyForm(PAdicExpand(x^3, x-1));

                             2                3
3 * ( x - 1 ) + 3 * ( x - 1 )  + ( x - 1 )  + 1

Out> True;
```

See also: Mod, ContFrac, FromBase, ToBase

`IsQuadraticResidue` — functions related to finite groups

`LegendreSymbol` — functions related to finite groups

`JacobiSymbol` — functions related to finite groups

(standard library)

Calling format:

```
IsQuadraticResidue(m,n)
LegendreSymbol(m,n)
JacobiSymbol(m,n)
```

Parameters:

`m`, `n` – integers, n must be odd and positive

Description:

A number m is a "quadratic residue modulo n" if there exists a number k such that $k^2 \equiv m \bmod n$.

The Legendre symbol (m/n) is defined as $+1$ if m is a quadratic residue modulo n and -1 if it is a non-residue. The Legendre symbol is equal to 0 if $\frac{m}{n}$ is an integer.

The Jacobi symbol $\left(\frac{m}{n}\right)$ is defined as the product of the Legendre symbols of the prime factors f_i of $n = f_1^{p_1} ... f_s^{p_s}$,

$$\left(\frac{m}{n}\right) \equiv \left(\frac{m}{f_1}\right)^{p_1} ... \left(\frac{m}{f_s}\right)^{p_s}.$$

(Here we used the same notation $\left(\frac{a}{b}\right)$ for the Legendre and the Jacobi symbols; this is confusing but seems to be the current practice.) The Jacobi symbol is equal to 0 if m, n are not mutually prime (have a common factor). The Jacobi symbol and the Legendre symbol have values $+1$, -1 or 0. If n is prime, then the Jacobi symbol is the same as the Legendre symbol.

The Jacobi symbol can be efficiently computed without knowing the full factorization of the number n.

Examples:

```
In> IsQuadraticResidue(9,13)
Out> True;
In> LegendreSymbol(15,23)
Out> -1;
In> JacobiSymbol(7,15)
Out> -1;
```

See also: `Gcd`

`GaussianFactors` — factorization in Gaussian integers

(standard library)

Calling format:

```
GaussianFactors(z)
```

Parameters:

`z` – Gaussian integer

Description:

This function decomposes a Gaussian integer number `z` into a product of Gaussian prime factors. A Gaussian integer is a complex number with integer real and imaginary parts. A Gaussian integer z can be decomposed into Gaussian primes essentially in a unique way (up to Gaussian units and associated prime factors), i.e. one can write z as

$$z = u p_1^{n_1} ... p_s^{n_s},$$

where u is a Gaussian unit and p_1, p_2, ..., p_s are Gaussian primes.

The factorization is returned as a list of pairs. The first member of each pair is the factor (a Gaussian integer) and the second member denotes the power to which this factor should be raised. So the factorization is returned as a list, e.g. `{{p1,n1}, {p2,n2}, ...}`.

Examples:

```
In> GaussianFactors(5)
Out> {{Complex(2,1),1},{Complex(2,-1),1}};
In> GaussianFactors(3+I)
Out> {{Complex(1,1),1},{Complex(2,-1),1}};
```

See also: `Factors`, `IsGaussianPrime`, `IsGaussianUnit`

`GaussianNorm` — norm of a Gaussian integer

(standard library)

Calling format:

```
GaussianNorm(z)
```

Parameters:

`z` – Gaussian integer

Description:

This function returns the norm of a Gaussian integer $z = a + b\imath$, defined as $a^2 + b^2$.

Examples:

```
In> GaussianNorm(2+I)
Out> 5;
```

See also: `IsGaussianInteger`

`IsGaussianUnit` — test for a Gaussian unit

(standard library)

Calling format:

```
IsGaussianUnit(z)
```

Parameters:

z – a Gaussian integer

Description:

This function returns True if the argument is a unit in the Gaussian integers and False otherwise. A unit in a ring is an element that divides any other element.

There are four "units" in the ring of Gaussian integers, which are 1, -1, $\imath$, and $-\imath$.

Examples:

```
In> IsGaussianInteger(I)
Out> True;
In> IsGaussianUnit(5+6*I)
Out> False;
```

See also: IsGaussianInteger, IsGaussianPrime, GaussianNorm

IsGaussianPrime — test for a Gaussian prime

(standard library)

Calling format:

```
IsGaussianPrime(z)
```

Parameters:

z – a complex or real number

Description:

This function returns True if the argument is a Gaussian prime and False otherwise.

A prime element x of a ring is divisible only by the units of the ring and by associates of x. ("Associates" of x are elements of the form xu where u is a unit of the ring).

Gaussian primes are Gaussian integers $z = a + b\imath$ that satisfy one of the following properties:

- If $Re(z)$ and $Im(z)$ are nonzero then z is a Gaussian prime if and only if $Re(z)^2 + Im(z)^2$ is an ordinary prime.
- If $Re(z) = 0$ then z is a Gaussian prime if and only if $Im(z)$ is an ordinary prime and $Im(z) \equiv 3 \bmod 4$.
- If $Im(z) = 0$ then z is a Gaussian prime if and only if $Re(z)$ is an ordinary prime and $Re(z) \equiv 3 \bmod 4$.

Examples:

```
In> IsGaussianPrime(13)
Out> False;
In> IsGaussianPrime(2+2*I)
Out> False;
In> IsGaussianPrime(2+3*I)
Out> True;
In> IsGaussianPrime(3)
Out> True;
```

See also: IsGaussianInteger, GaussianFactors

GaussianGcd — greatest common divisor in Gaussian integers

(standard library)

Calling format:

```
GaussianGcd(z,w)
```

Parameters:

z, w – Gaussian integers

Description:

This function returns the greatest common divisor, in the ring of Gaussian integers, computed using Euclid's algorithm. Note that in the Gaussian integers, the greatest common divisor is only defined up to a Gaussian unit factor.

Examples:

```
In> GaussianGcd(2+I,5)
Out> Complex(2,1);
```

The GCD of two mutually prime Gaussian integers might come out to be equal to some Gaussian unit instead of 1:

```
In> GaussianGcd(2+I,3+I)
Out> -1;
```

See also: Gcd, Lcm, IsGaussianUnit

Chapter 32

GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, `LaTeX` input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.
9. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
11. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
13. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
14. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C) YEAR   YOUR NAME. Permission is
granted to copy, distribute and/or modify this
document under the terms of the GNU Free
Documentation License, Version 1.1 or any later
version published by the Free Software Foundation;
with the Invariant Sections being LIST THEIR
TITLES, with the Front-Cover Texts being LIST, and
with the Back-Cover Texts being LIST. A copy of
the license is included in the section entitled
``GNU Free Documentation License''.
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index