Yacas

Release 1.4.0

Ayal Pinkus, Serge Winitzki and Grzegorz Mazur

February 04, 2016

Contents

CHAPTER 1

Getting started

1.1 Installation

1.1.1 MacOS X

- Download yacas-1.4.0-Darwin.dmg from https://github.com/grzegorzmazur/yacas/releases/
- Double-click the file to open it

1.1.2 Microsoft Windows

- Depending on the Microsoft Windows version you use, download
  - yacas-1.4.0-win64.exe (Windows 7 or newer, 64 bit)
  - yacas-1.4.0-win32.exe (Windows 7 or newer, 32 bit)
  - yacas-1.4.0-winxp.exe (Windows XP)

  from https://github.com/grzegorzmazur/yacas/releases/
- Run the downloaded installator program, which will guide you through the installation process

Alternatively, download either yacas-1.4.0-win64.zip or yacas-1.4.0-win32.zip and unpack it. Then, you can run yacas by executing `yacas.exe` from the `bin` subdirectory. This method, while not recommended, allows one to use yacas without having to install it system-wide.

1.1.3 Ubuntu

- Download yacas-1.4.0-1_amd64.deb from https://github.com/grzegorzmazur/yacas/releases/
- Double-click the file to open it in Ubuntu Software Center and click the *Install* button

1.2 Installation from sources

1.2.1 Getting sources

Version 1.4.0 can be downloaded from https://github.com/grzegorzmazur/yacas/archive/v1.4.0.zip or https://github.com/grzegorzmazur/yacas/archive/v1.4.0.tar.gz, while the current development version is accessible from https://github.com/grzegorzmazur/yacas/archive/master.zip.

1.2.2 Compilation

MacOS X

- open `Terminal` window
- change directory to the yacas source directory
- execute

```
mkdir build
cd build
cmake -G Xcode ..
```

- open generated project in `Xcode`

Microsoft Windows

- open `Command Prompt` window
- change directory to the yacas source directory
- execute

```
mkdir build
cd build
cmake -G "Visual Studio 14 2015 Win64" ..
```

- open generated project in `Visual Studio`

Linux

- open `Terminal` window
- change directory to the yacas source directory
- execute

```
mkdir build
cd build
cmake ..
make
```

- to install newly built binaries either `make install` or build the binary package using `make package` and install it

Java

- open `Terminal` or `Command Prompt` window
- change directory to the yacas source directory
- execute `ant jar`

CHAPTER 2

Tutorial

2.1 Yacas syntax

Expressions in Yacas are generally built up of words. We will not bore you with the exact definitions of such words, but roughly speaking they are either sequences of alphabetic letters, or a number, or a bracket, or space to separate words, or a word built up from symbols like +, −, ⋆, <, *etc.*. If you want, you can mix these different types of characters, by surrounding them with quotes. Thus, `"This text"` is what is called one token, surrounded by quotes.

The usual notation people use when writing down a calculation is called the infix notation, and you can readily recognize it, as for example `2+3` and `3*4`. Prefix operators also exist. These operators come before an expression, like for example the unary minus sign (called unary because it accepts one argument), `-(3*4)`. In addition to prefix operators there are also postfix operators, like the exclamation mark to calculate the factorial of a number, `10!`.

Yacas understands standard simple arithmetic expressions. Some examples:

- `2+3` (addition)
- `2*3` (multiplication)
- `2-3` (subtraction)
- `2^3` (raising powers)
- `2+3*4`
- `(2+3)*4`
- `6/3` (division)
- `1/3`

Divisions are not reduced to real numbers, but kept as a rational for as long as possible, since the rational is an exact correct expression (and any real number would just be an approximation). Yacas is able to change a rational in to a number with the function `N`, for example `N(1/3)`.

Operators have *precedence*, meaning that certain operations are done first before others are done. For example, in `2+3*4` the multiplication is performed before the addition. The usual way to change the order of a calculation is with round brackets. The round brackets in the expression `(2+3)*4` will force Yacas to first add 2 and 3, and then multiply the result.

Simple function calls have their arguments between round brackets, separated by commas. Examples are `Sin(Pi)` (which indicates that you are interested in the value of the trigonometric function sin applied to the constant π), and `Min(5,1,3,-5,10)` (which should return the lowest of its arguments, `-5` in this case). Functions usually have the form `f()`, `f(x)` or `f(x,y,z,...)` depending on how many arguments the function accepts. Functions always return a result. For example, `Cos(0)` should return `1`. Evaluating functions can be thought of as simplifying an expression as much as possible. Sometimes further simplification is not possible and a function returns itself

unsimplified, like taking the square root of an integer `Sqrt(2)`. A reduction to a number would be an approximation. We explain elsewhere how to get Yacas to simplify an expression to a number.

Yacas allows for use of the infix notation, but with some additions. Functions can be *bodied*, meaning that the last argument is written past the close bracket. An example is `ForEach`, where we write `ForEach(item, 1 .. 10) Echo(item);`. `Echo(item)` is the last argument to the function `ForEach`.
 A list is enclosed with curly braces, and is written out with commas between the elements, like for example `{1,2,3}`. items in lists (and things that can be made to look like lists, like arrays and strings), can then be accessed by indicating the index between square brackets after the object. `{a,b,c}[2]` should return `b`, as `b` is the second element in the list (Yacas starts counting from 1 when accessing elements). The same can be done with strings: `"abc"[2]`.
 And finally, function calls can be grouped together, where they get executed one at a time, and the result of executing the last expression is returned. This is done through square brackets, as `[ Echo("Hello"); Echo("World"); True; ];`, which first writes `Hello` to screen, then `World` on the next line, and then returns `True`.

When you type in an expression, you have to take in to account the fact that Yacas is case-sensitive. This means that a function `sin` (with all lowercase) is a different function from `Sin` (which starts with a capital S), and the variable `v` is a different one from `V`.

2.2 Using Yacas from the calculation center

As mentioned earlier, you can type in commands on the command line in the calculation center. Typically, you would enter one statement per line, for example, click on `Sin(Pi/2);`. The has a memory, and remembers results from calculations performed before. For example, if you define a function on a line (or set a variable to a value), the defined function (or variable) are available to be used in following lines. A session can be restarted (forgetting all previous definitions and results) by typing `restart`. All memory is erased in that case.

Statements should end with a semicolon `;` although this is not required in interactive sessions (Yacas will append a semicolon at end of line to finish the statement).

The command line has a history list, so it should be easy to browse through the expressions you entered previously using the up and down arrow keys.

When a few characters have been typed, the command line will use the characters before the cursor as a filter into the history, and allow you to browse through all the commands in the history that start with these characters quickly, instead of browsing through the entire history. If the system recognized the first few characters, it will also show the commands that start with the sequence entered. You can use the arrow keys to browse through this list, and then select the intended function to be inserted by pressing enter.

Commands spanning multiple lines can (and actually have to) be entered by using a trailing backslash at end of each continued line. For example, clicking on `2+3+` will result in an error, but entering the same with a backslash at the end and then entering another expression will concatenate the two lines and evaluate the concatenated input.

Incidentally, any text Yacas prints without a prompt is either a message printed by a function as a side-effect, or an error message. Resulting values of expressions are always printed after an `Out>` prompt.

2.3 Yacas as a symbolic calculator

We are ready to try some calculations. Yacas uses a C-like infix syntax and is case-sensitive. Here are some exact manipulations with fractions for a start: `1/14+5/21*(30-(1+1/2)*5^2);`

The standard scripts already contain a simple math library for symbolic simplification of basic algebraic functions. Any names such as `x` are treated as independent, symbolic variables and are not evaluated by default. Some examples to try:

- `0+x`

- `x+1*y`
- `Sin(ArcSin(alpha))+Tan(ArcTan(beta))`

Note that the answers are not just simple numbers here, but actual expressions. This is where Yacas shines. It was built specifically to do calculations that have expressions as answers.

In Yacas after a calculation is done, you can refer to the previous result with `%`. For example, we could first type `(x+1)*(x-1)`, and then decide we would like to see a simpler version of that expression, and thus type `Simplify(%)`, which should result in `x^2-1`.

The special operator `%` automatically recalls the result from the previous line. The function `Simplify` attempts to reduce an expression to a simpler form. Note that standard function names in Yacas are typically capitalized. Multiple capitalization such as `ArcSin` is sometimes used. The underscore character _ is a reserved operator symbol and cannot be part of variable or function names.

Yacas offers some more powerful symbolic manipulation operations. A few will be shown here to wetten the appetite.

Some simple equation solving algorithms are in place:

- `Solve(x/(1+x) == a, x);`
- `Solve(x^2+x == 0, x);`
- `Solve(a+x*y==z,x);`

(Note the use of the == operator, which does not evaluate to anything, to denote an "equation" object.)

Taylor series are supported, for example: `Taylor(x,0,3) Exp(x)` is a bodied operator that expands `Exp(x)` for `x` around `x=0`, up to order 3.

Symbolic manipulation is the main application of Yacas. This is a small tour of the capabilities Yacas currently offers. Note that this list of examples is far from complete. Yacas contains a few hundred commands, of which only a few are shown here.

- `Expand((1+x)^5);` (expand the expression into a polynomial)
- `Limit(x,0) Sin(x)/x;` (calculate the limit of `Sin(x)/x` as `x` approaches zero)
- `Newton(Sin(x),x,3,0.0001);` (use Newton's method to find the value of `x` near `3` where `Sin(x)` equals zero numerically and stop if the result is closer than `0.0001` to the real result)
- `DiagonalMatrix({a,b,c});` (create a matrix with the elements specified in the vector on the diagonal)
- `Integrate(x,a,b) x*Sin(x);` (integrate a function over variable `x`, from `a` to `b`)
- `Factor(x^2-1);` (factorize a polynomial)
- `Apart(1/(x^2-1),x);` (create a partial fraction expansion of a polynomial)
- `Simplify((x^2-1)/(x-1));` (simplification of expressions)
- `CanProve( (a And b) Or (a And Not b) );` (special-purpose simplifier that tries to simplify boolean expressions as much as possible)
- `TrigSimpCombine(Cos(a)*Sin(b));` (special-purpose simplifier that tries to transform trigonometric expressions into a form where there are only additions of trigonometric functions involved and no multiplications)

2.4 Arbitrary precision numbers

Yacas can deal with arbitrary precision numbers. It can work with large integers, like `20!` (The `!` means factorial, thus `1*2*3*...*20`).

As we saw before, rational numbers will stay rational as long as the numerator and denominator are integers, so `55/10` will evaluate to `11/2`. You can override this behavior by using the numerical evaluation function `N()`. For example, `N(55/10)` will evaluate to `5.5` . This behavior holds for most math functions. Yacas will try to maintain an exact answer (in terms of integers or fractions) instead of using floating point numbers, unless `N()` is used. Where the value for the constant pi is needed, use the built-in variable `Pi`. It will be replaced by the (approximate) numerical value when `N(Pi)` is called. Yacas knows some simplification rules using `Pi` (especially with trigonometric functions).

The function `N` takes either one or two arguments. It evaluates its first argument and tries to reduce it as much as possible to a real-valued approximation of the expression. If the second argument is present, it states the number of digits precision required. Thus `N(1/234)` returns a number with the current default precision (which starts at 20 digits), but you can request as many digits as you like by passing a second argument, as in `N(1/234, 10)`, `N(1/234, 20)`,`N(1/234, 30)`, etcetera.

Note that we need to enter `N()` to force the approximate calculation, otherwise the fraction would have been left unevaluated.

Revisiting `Pi`, we can get as many digits of `Pi` as we like, by providing the precision required as argument to `N`. So to get 50 digits precision, we can evaluate `N(Pi,50)`.

Taking a derivative of a function was amongst the very first of symbolic calculations to be performed by a computer, as the operation lends itself surprisingly well to being performed automatically. Naturally, it is also implemented in Yacas, through the function `D`. `D` is a bodied function, meaning that its last argument is past the closing brackets. Where normal functions are called with syntax similar to `f(x,y,z)`, a bodied function would be called with a syntax `f(x,y)z`. Here are two examples of taking a derivative: <ul> `D(x) Sin(x);` (taking a derivative) ‘‘D(x) D(x) Sin(x);‘‘ (taking a derivative twice) The {D} function also accepts an argument specifying how often the derivative has to be taken. In that case, the above expressions can also be written as: <ul> `D(x,1) Sin(x);` (taking a derivative) ‘‘D(x,2) Sin(x);‘‘ (taking a derivative twice)

2.5 Analytic functions

Many of the usual analytic functions have been defined in the Yacas library. Examples are `Exp(1)`, `Sin(2)`, `ArcSin(1/2)`, `Sqrt(2)`. These will not evaluate to a numeric result in general, unless the result is an integer, like `Sqrt(4)`. If asked to reduce the result to a numeric approximation with the function `N`, then Yacas will do so, as for example in `N(Sqrt(2),50)`.

2.6 Variables

Yacas supports variables. You can set the value of a variable with the `:=` infix operator, as in `a:=1;`. The variable can then be used in expressions, and everywhere where it is referred to, it will be replaced by its value.

To clear a variable binding, execute `Clear(a);`. A variable will evaluate to itself after a call to clear it (so after the call to clear `a` above, calling a‘‘ should now return `a`). This is one of the properties of the evaluation scheme of Yacas; when some object can not be evaluated or transformed any further, it is returned as the final result.

2.7 Functions

The `:=` operator can also be used to define simple functions: `f(x):=2*x*x`. will define a new function, `f`, that accepts one argument and returns twice the square of that argument. This function can now be called, `f(a)`. You can change the definition of a function by defining it again.

One and the same function name such as `f` may define different functions if they take different numbers of arguments. One can define a function `f` which takes one argument, as for example `f(x):=x^2;`, or two arguments, `f(x,y):=x*y;`. If you clicked on both links, both functions should now be defined, and `f(a)` calls the one function whereas `f(a,b)` calls the other.

Yacas is very flexible when it comes to types of mathematical objects. Functions can in general accept or return any type of argument.

2.8 Boolean expressions and predicates

Yacas predefines `True` and `False` as boolean values. Functions returning boolean values are called predicates. For example, `IsNumber()` and‘‘ IsInteger()‘‘ are predicates defined in the Yacas environment. For example, try `IsNumber(2+x);`, or `IsInteger(15/5);`.

There are also comparison operators. Typing `2 > 1` would return `True`. You can also use the infix operators `And` and `Or`, and the prefix operator `Not`, to make more complex boolean expressions. For example, try `True And False`, `True Or False`, `True And Not(False)`.

2.9 Strings and lists

In addition to numbers and variables, Yacas supports strings and lists. Strings are simply sequences of characters enclosed by double quotes, for example: `"this is a string with \"quotes\" in it"`.

Lists are ordered groups of items, as usual. Yacas represents lists by putting the objects between braces and separating them with commas. The list consisting of objects a, b, and c could be entered by typing `{a,b,c}`. In Yacas, vectors are represented as lists and matrices as lists of lists.

Items in a list can be accessed through the `[ ]` operator. The first element has index one. Examples: when you enter `uu:={a,b,c,d,e,f};` then `uu[2];` evaluates to `b`, and `uu[2 .. 4];` evaluates to `{b,c,d}`. The “range” expression `2 .. 4` evaluates to `{2,3,4}`. Note that spaces around the `..` operator are necessary, or else the parser will not be able to distinguish it from a part of a number.

Lists evaluate their arguments, and return a list with results of evaluating each element. So, typing `{1+2,3};` would evaluate to `{3,3}`.

The idea of using lists to represent expressions dates back to the language LISP developed in the 1970’s. From a small set of operations on lists, very powerful symbolic manipulation algorithms can be built. Lists can also be used as function arguments when a variable number of arguments are necessary.

Let’s try some list operations now. First click on `m:={a,b,c};` to set up an initial list to work on. Then click on links below: <ul> ‘‘Length(m);‘‘ (return the length of a list) ‘‘Reverse(m);‘‘ (return the string reversed) ‘‘Concat(m,m);‘‘ (concatenate two strings) ‘‘m[1]:=d;‘‘ (setting the first element of the list to a new value, d, as can be verified by evaluating `m`) Many more list operations are described in the reference manual.

2.10 Writing simplification rules

Mathematical calculations require versatile transformations on symbolic quantities. Instead of trying to define all possible transformations, Yacas provides a simple and easy to use pattern matching scheme for manipulating expressions according to user-defined rules. Yacas itself is designed as a small core engine executing a large library of rules to match and replace patterns.

One simple application of pattern-matching rules is to define new functions. (This is actually the only way Yacas can learn about new functions.) As an example, let's define a function `f` that will evaluate factorials of non-negative integers. We will define a predicate to check whether our argument is indeed a non-negative integer, and we will use this predicate and the obvious recursion `f(n)=n*f(n-1) if n>0 and 1 if n=0` to evaluate the factorial.

We start with the simple termination condition, which is that `f(n)` should return one if `n` is zero: <ul> ‘‘10 # f(0) <– 1;’’ You can verify that this already works for input value zero, with `f(0)`.

Now we come to the more complex line, <ul> ‘‘20 # f(n_IsIntegerGreaterThanZero) <– n*f(n-1);’’ Now we realize we need a function `IsGreaterThanZero`, so we define this function, with <ul> ‘‘IsIntegerGreaterThanZero(_n) <– (IsInteger(n) And n>0);’’ You can verify that it works by trying `f(5)`, which should return the same value as `5!`.

In the above example we have first defined two "simplification rules" for a new function `f()`. Then we realized that we need to define a predicate `IsIntegerGreaterThanZero()`. A predicate equivalent to `IsIntegerGreaterThanZero()` is actually already defined in the standard library and it's called `IsPositiveInteger`, so it was not necessary, strictly speaking, to define our own predicate to do the same thing. We did it here just for illustration purposes.

The first two lines recursively define a factorial function `f(n)=n*(n-1)*...*1`. The rules are given precedence values 10 and 20, so the first rule will be applied first. Incidentally, the factorial is also defined in the standard library as a postfix operator ! and it is bound to an internal routine much faster than the recursion in our example. The example does show how to create your own routine with a few lines of code. One of the design goals of Yacas was to allow precisely that, definition of a new function with very little effort.

The operator `<--` defines a rule to be applied to a specific function. (The `<--` operation cannot be applied to an atom.) The `_n` in the rule for `IsIntegerGreaterThanZero()` specifies that any object which happens to be the argument of that predicate is matched and assigned to the local variable `n`. The expression to the right of `<--` can use n (without the underscore) as a variable.

Now we consider the rules for the function `f`. The first rule just specifies that `f(0)` should be replaced by 1 in any expression. The second rule is a little more involved. `n_IsIntegerGreaterThanZero` is a match for the argument of `f`, with the proviso that the predicate `IsIntegerGreaterThanZero(n)` should return `True`, otherwise the pattern is not matched. The underscore operator is to be used only on the left hand side of the rule definition operator `<--`.

There is another, slightly longer but equivalent way of writing the second rule: <ul> ‘‘20 # f(_n)_(IsIntegerGreaterThanZero(n)) <– n*f(n-1);’’ The underscore after the function object denotes a "postpredicate" that should return `True` or else there is no match. This predicate may be a complicated expression involving several logical operations, unlike the simple checking of just one predicate in the `n_IsIntegerGreaterThanZero` construct. The postpredicate can also use the variable `n` (without the underscore).

Precedence values for rules are given by a number followed by the # infix operator (and the transformation rule after it). This number determines the ordering of precedence for the pattern matching rules, with 0 the lowest allowed precedence value, i.e. rules with precedence 0 will be tried first. Multiple rules can have the same number: this just means that it doesn't matter what order these patterns are tried in. If no number is supplied, 0 is assumed. In our example, the rule `f(0) <-- 1` must be applied earlier than the recursive rule, or else the recursion will never terminate. But as long as there are no other rules concerning the function `f`, the assignment of numbers 10 and 20 is arbitrary, and they could have been 500 and 501 just as well. It is usually a good idea however to keep some space between these numbers, so you have room to insert new transformation rules later on.

Predicates can be combined: for example, {IsIntegerGreaterThanZero()} could also have been defined as: <ul> ‘‘10 # IsIntegerGreaterThanZero(n_IsInteger)_(n>0) <– True;’’ ‘‘20 # IsIntegerGreaterThanZero(_n) <– False;’’ The first rule specifies that if n is an integer, and is greater than zero, the result is `True`, and the second rule states that otherwise (when the rule with precedence 10 did not apply) the predicate returns `False`.

In the above example, the expression `n > 0` is added after the pattern and allows the pattern to match only if

this predicate return `True`. This is a useful syntax for defining rules with complicated predicates. There is no difference between the rules‘‘ F(n_IsPositiveInteger) <–...‘‘ and `F(_n)_(IsPositiveInteger(n)) <-- ...` except that the first syntax is a little more concise.

The left hand side of a rule expression has the following form:

```
<i>precedence</i> # <i>pattern</i> _ <i>postpredicate</i> <-- <i>replacement</i> ;
```

The optional *precedence* must be a positive integer.

Some more examples of rules (not made clickable because their equivalents are already in the basic Yacas library): <ul> ‘‘10 # _x + 0 <– x;‘‘ ‘‘20 # _x - _x <– 0;‘‘ ‘‘ArcSin(Sin(_x)) <– x;‘‘ The last rule has no explicit precedence specified in it (the precedence zero will be assigned automatically by the system).

Yacas will first try to match the pattern as a template. Names preceded or followed by an underscore can match any one object: a number, a function, a list, etc. Yacas will assign the relevant variables as local variables within the rule, and try the predicates as stated in the pattern. The post-predicate (defined after the pattern) is tried after all these matched. As an example, the simplification rule `_x - _x <--0` specifies that the two objects at left and at right of the minus sign should be the same for this transformation rule to apply.

2.11 Local simplification rules

Sometimes you have an expression, and you want to use specific simplification rules on it that should not be universally applied. 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:

- `Sin(x)*Ln(a*b)` (example expression without simplification)
- `Sin(x)*Ln(a*b) /: {Ln(_x*_y) <- Ln(x)+Ln(y) }` (with instruction to simplify the expression)

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 `/:`.

Note that for these local rules, `<-` should be used instead of `<--`. Using latter would result in a global definition of a new transformation rule on evaluation, which is not the intention.

The `/:` operator traverses an expression from the top down, trying to apply the rules from the beginning of the list of rules to the end of the list of rules. If no rules can be applied to the whole expression, it will try the sub-expressions of the expression being analyzed.

It might be sometimes necessary to use the `/::` operator, which repeatedly applies the `/:` operator until the result does not change any more. Caution is required, since rules can contradict each other, and that could result in an infinite loop. To detect this situation, just use `/:` repeatedly on the expression. The repetitive nature should become apparent.

2.12 Programming essentials

An important feature of Yacas is its programming language which allows you to create your own programs for doing calculations. This section describes some constructs and functions for control flow.

Looping can be done with the function `ForEach`. There are more options, but ForEach is the simplest to use for now and will suffice for this turorial. The statement form `ForEach(x, list) body` executes its body for each element of the list and assigns the variable x to that element each time. The statement form `While(predicate)`

`body` repeats execution of the expression represented by `body` until evaluation of the expression represented by `predicate` returns `False`.

This example loops over the integers from one to three, and writes out a line for each, multiplying the integer by 3 and displaying the result with the function `Echo`: `ForEach(x,1 .. 5) Echo(x," times 3 equals ",3*x);`

2.12.1 Compound statements

Multiple statements can be grouped together using the `[` and `]` brackets. The compound `[a; Echo("In the middle"); 1+2;];` evaluates `a`, then the `Echo` command, and finally evaluates `1+2`, and returns the result of evaluating the last statement `1+2`.

A variable can be declared local to a compound statement block by the function `Local(var1, var2, ...)`. For example, if you execute `[Local(v);v:=1+2;v;];` the result will be `3`. The program body created a variable called `v`, assigned the value of evaluating `1+2` to it, and made sure the contents of the variable `v` were returned. If you now evaluate `v` afterwards you will notice that the variable `v` is not bound to a value any more. The variable `v` was defined locally in the program body between the two square brackets `[` and `]`.

Conditional execution is implemented by the `If(predicate, body1, body2)` function call. If the expression `predicate` evaluates to `True`, the expression represented by `body1` is evaluated, otherwise `body2` is evaluated, and the corresponding value is returned. For example, the absolute value of a number can be computed with: `f(x) := If(x < 0,-x,x);` (note that there already is a standard library function that calculates the absolute value of a number).

Variables can also be made to be local to a small set of functions, with `LocalSymbols(variables) body`. For example, the following code snippet: `LocalSymbols(a,b) [a:=0;b:=0; inc():=[a:=a+1;b:=b-1;show();]; show():=Echo("a = ",a," b = ",b); ];` defines two functions, `inc` and `show`. Calling `inc()` repeatedly increments `a` and decrements `b`, and calling `show()` then shows the result (the function "inc" also calls the function "show", but the purpose of this example is to show how two functions can share the same variable while the outside world cannot get at that variable). The variables are local to these two functions, as you can see by evaluating `a` and `b` outside the scope of these two functions. This feature is very important when writing a larger body of code, where you want to be able to guarantee that there are no unintended side-effects due to two bits of code defined in different files accidentally using the same global variable.

To illustrate these features, let us create a list of all even integers from 2 to 20 and compute the product of all those integers except those divisible by 3

```
[
    Local(L,i,answer);
    L:={}; i:=2;
    /*Make a list of all even integers from 2 to 20 */
    While (i <= 20) [ L := Append(L, i); i := i + 2; ];
    /* Now calculate the product of all of these numbers that are not divisible by 3 */
    answer := 1;
    ForEach(i,L) If (Mod(i, 3) != 0, answer := answer * i);
    /* And return the answer */
    answer;
];
```

(Note that it is not necessarily the most economical way to do it in Yacas.)

We used a shorter form of `If(predicate, body)` with only one body which is executed when the condition holds. If the condition does not hold, this function call returns `False`. We also introduced comments, which can be placed between `/*` and `*/`. Yacas will ignore anything between those two. When putting a program in a file you can also use `//`. Everything after `//` up until the end of the line will be a comment. Also shown is the use of the `While` function. Its form is `While (predicate) body`. While the expression represented by `predicate` evaluates to `True`, the expression represented by `body` will keep on being evaluated.

The above example is not the shortest possible way to write out the algorithm. It is written out in a procedural way, where the program explains step by step what the computer should do. There is nothing fundamentally wrong with the approach of writing down a program in a procedural way, but the symbolic nature of Yacas also allows you to write it in a more concise, elegant, compact way, by combining function calls.

There is nothing wrong with procedural style, but there is amore 'functional' approach to the same problem would go as follows below. The advantage of the functional approach is that it is shorter and more concise (the difference is cosmetic mostly).

Before we show how to do the same calculation in a functional style, we need to explain what a *pure function* is, as you will need it a lot when programming in a functional style. We will jump in with an example that should be self-explanatory. Consider the expression `Lambda({x,y},x+y)`. This has two arguments, the first listing x and y, and the second an expression. We can use this construct with the function Apply as follows: `Apply(Lambda({x,y},x+y),{2,3})`. The result should be `5`, the result of adding `2` and `3`. The expression starting with `Lambda` is essentially a prescription for a specific operation, where it is stated that it accepts 2 arguments, and returns the arguments added together. In this case, since the operation was so simple, we could also have used the name of a function to apply the arguments to, the addition operator in this case `Apply("+",{2,3})`. When the operations become more complex however, the Lambda construct becomes more useful.

Now we are ready to do the same example using a functional approach. First, let us construct a list with all even numbers from 2 to 20. For this we use the `..` operator to set up all numbers from one to ten, and then multiply that with two: `2 * (1 .. 10)`.

Now we want an expression that returns all the even numbers up to 20 which are not divisible by 3. For this we can use `Select`, which takes as first argument a predicate that should return `True` if the list item is to be accepted, and `False` otherwise, and as second argument the list in question: `Select(Lambda({n},Mod(n,3)!=0),2*(1 .. 10))`. The numbers 6, 12 and 18 have been correctly filtered out. Here you see one example of a pure function where the operation is a little bit more complex.

All that remains is to factor the items in this list. For this we can use `UnFlatten`. Two examples of the use of `UnFlatten` are

- `UnFlatten({a,b,c},"*",1)`
- `UnFlatten({a,b,c},"+",0)`

The 0 and 1 are a base element to start with when grouping the arguments in to an expression (they should be the respective identity elements, hence it is zero for addition and 1 for multiplication).

Now we have all the ingredients to finally do the same calculation we did above in a procedural way, but this time we can do it in a functional style, and thus captured in one concise single line:

```
UnFlatten(Select(Lambda({n},Mod(n,3)!=0),2*(1 .. 10)),"*",1)
```

As was mentioned before, the choice between the two is mostly a matter of style.

2.13 Macros

One of the powerful constructs in Yacas is the construct of a macro. In its essence, a macro is a prescription to create another program before executing the program. An example perhaps explains it best. Evaluate the following expression `Macro(for,{st,pr,in,bd}) [(@st);While(@pr)[(@bd);(@in);];];`. This expression defines a macro that allows for looping. Yacas has a `For` function already, but this is how it could be defined in one line (In Yacas the `For` function is bodied, we left that out here for clarity, as the example is about macros).

To see it work just type `for(i:=0,i&lt;3,i:=i+1,Echo(i))`. You will see the count from one to three.

The construct works as follows; The expression defining the macro sets up a macro named `for` with four arguments. On the right is the body of the macro. This body contains expressions of the form `@var`. These are replaced by the

values passed in on calling the macro. After all the variables have been replaced, the resulting expression is evaluated. In effect a new program has been created. Such macro constructs come from LISP, and are famous for allowing you to almost design your own programming language constructs just for your own problem at hand. When used right, macros can greatly simplify the task of writing a program.

You can also use the back-quote ` to expand a macro in-place. It takes on the form `(expression), where the expression can again contain sub-expressions of the form @variable. These instances will be replaced with the values of these variables.

2.14 The practice of programming in Yacas

When you become more proficient in working with Yacas you will be doing more and more sophisticated calculations. For such calculations it is generally necessary to write little programs. In real life you will usually write these programs in a text editor, and then start Yacas, load the text file you just wrote, and try out the calculation. Generally this is an iterative process, where you go back to the text editor to modify something, and then go back to Yacas, type restart and then reload the file.

On this site you can run Yacas in a little window called a Yacas calculation center (the same as the one below this tutorial). On page there is tab that contains a Yacas calculation center. If you click on that tab you will be directed to a larger calculation center than the one below this tutorial. In this page you can easily switch between doing a calculation and editing a program to load at startup. We tried to make the experience match the general use of Yacas on a desktop as much as possible. The Yacas journal (which you see when you go to the Yacas web site) contains examples of calculations done before by others.

2.15 Defining your own operators

Large part of the Yacas system is defined in the scripting language itself. This includes the definitions of the operators it accepts, and their precedences. This means that you too can define your own operators. This section shows you how to do that.

Suppose we wanted to define a function F(x,y)=x/y+y/x. We could use the standard syntax F(a,b) := a/b + b/a;. F(1,2);. For the purpose of this demonstration, lets assume that we want to define an infix operator xx for this operation. We can teach Yacas about this infix operator with Infix("xx", OpPrecedence("/"));. Here we told Yacas that the operator xx is to have the same precedence as the division operator. We can now proceed to tell Yacas how to evaluate expressions involving the operator xx by defining it as we would with a function, a xx b := a/b + b/a;.

You can verify for yourself 3 xx 2 + 1; and 1 + 3 xx 2; return the same value, and that they follow the precedence rules (eg. xx binds stronger than +).

We have chosen the name xx just to show that we don't need to use the special characters in the infix operator's name. However we must define this operator as infix before using it in expressions, otherwise Yacas will raise a syntax error.

Finally, we might decide to be completely flexible with this important function and also define it as a mathematical operator ## . First we define ## as a bodied function and then proceed as before. First we can tell Yacas that ## is a bodied operator with Bodied("##", OpPrecedence("/"));. Then we define the function itself: ##(a) b := a xx b;. And now we can use the function, ##(1) 3 + 2;.

We have used the name ## but we could have used any other name such as xx or F or even _-+@+-_. Apart from possibly confusing yourself, it doesn't matter what you call the functions you define.

There is currently one limitation in Yacas: once a function name is declared as infix (prefix, postfix) or bodied, it will always be interpreted that way. If we declare a function f to be bodied, we may later define different functions named f with different numbers of arguments, however all of these functions must be bodied.

When you use infix operators and either a prefix of postfix operator next to it you can run in to a situation where Yacas can not quite figure out what you typed. This happens when the operators are right next to each other and all consist of symbols (and could thus in principle form a single operator). Yacas will raise an error in that case. This can be avoided by inserting spaces.

2.16 Some assorted programming topics

One use of lists is the associative list, sometimes called a dictionary in other programming languages, which is implemented in Yacas simply as a list of key-value pairs. Keys must be strings and values may be any objects. Associative lists can also work as mini-databases, where a name is associated to an object. As an example, first enter `record:={};` to set up an empty record. After that, we can fill arbitrary fields in this record:

```
record["name"]:="Isaia";
record["occupation"]:="prophet";
record["is alive"]:=False;
```

Now, evaluating `record["name"]` should result in the answer `"Isaia"`. The record is now a list that contains three sublists, as you can see by evaluating `record`.

Assignment of multiple variables is also possible using lists. For instance, evaluating `{x,y}:={2!,3!}` will result in 2 being assigned to `x` and 6 to `y`.

When assigning variables, the right hand side is evaluated before it is assigned. Thus `a:=2*2` will set a to 4. This is however not the case for functions. When entering `f(x):=x+x` the right hand side, `x+x`, is not evaluated before being assigned. This can be forced by using `Eval()`. Defining `f(x)` with `f(x):=Eval(x+x)` will tell the system to first evaluate `x+x` (which results in `2*x`) before assigning it to the user function `f`. This specific example is not a very useful one but it will come in handy when the operation being performed on the right hand side is expensive. For example, if we evaluate a Taylor series expansion before assigning it to the user-defined function, the engine doesn't need to create the Taylor series expansion each time that user-defined function is called.

The imaginary unit i is denoted `I` and complex numbers can be entered as either expressions involving `I`, as for example `1+I*2`, or explicitly as `Complex(a,b)` for a+ib. The form `Complex(re,im)` is the way Yacas deals with complex numbers internally.

2.17 Linear Algebra

Vectors of fixed dimension are represented as lists of their components. The list `{1, 2+x, 3*Sin(p)}` would be a three-dimensional vector with components `1`, `2+x` and `3*Sin(p)`. Matrices are represented as a lists of lists.

Vector components can be assigned values just like list items, since they are in fact list items. If we first set up a variable called "vector" to contain a three-dimensional vector with the command `vector:=ZeroVector(3);` (you can verify that it is indeed a vector with all components set to zero by evaluating `vector`), you can change elements of the vector just like you would the elements of a list (seeing as it is represented as a list). For example, to set the second element to two, just evaluate `vector[2] := 2;`. This results in a new value for `vector`.

Yacas can perform multiplication of matrices, vectors and numbers as usual in linear algebra. The standard Yacas script library also includes taking the determinant and inverse of a matrix, finding eigenvectors and eigenvalues (in simple cases) and solving linear sets of equations, such as A * x = b where A is a matrix, and x and b are vectors. As a little example to wetten your appetite, we define a Hilbert matrix: `hilbert:=HilbertMatrix(3)`. We can then calculate the determinant with `Determinant(hilbert)`, or the inverse with `Inverse(hilbert)`. There are several more matrix operations supported. See the reference manual for more details.

2.17.1 "Threading" of functions

Some functions in Yacas can be "threaded". This means that calling the function with a list as argument will result in a list with that function being called on each item in the list. E.g. `Sin({a,b,c});` will result in `{Sin(a),Sin(b),Sin(c)}`. This functionality is implemented for most normal analytic functions and arithmetic operators.

2.17.2 Functions as lists

For some work it pays to understand how things work under the hood. Internally, Yacas represents all atomic expressions (numbers and variables) as strings and all compound expressions as lists, like Lisp. Try `FullForm(a+b*c);` and you will see the text `(+ a (* b c ))` appear on the screen. This function is occasionally useful, for example when trying to figure out why a specific transformation rule does not work on a specific expression.

If you try `FullForm(1+2)` you will see that the result is not quite what we intended. The system first adds up one and two, and then shows the tree structure of the end result, which is a simple number `3`. To stop Yacas from evaluating something, you can use the function `Hold`, as `FullForm(Hold(1+2))`. The function `Eval` is the opposite, it instructs Yacas to re-evaluate its argument (effectively evaluating it twice). This undoes the effect of `Hold`, as for example `Eval(Hold(1+2))`.

Also, any expression can be converted to a list by the function `Listify` or back to an expression by the function `UnList`:

- `Listify(a+b*(c+d));`
- `UnList({Atom("+"),x,1});`

Note that the first element of the list is the name of the function `+` which is equivalently represented as `Atom("+")` and that the subexpression `b*(c+d)` was not converted to list form. Listify just took the top node of the expression.

CHAPTER 3

Reference Manual

{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.

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.

3.1 Arithmetic and other operations on numbers

x **+** y
: addition

  Addition can work on integers, rational numbers, complex numbers, vectors, matrices and lists.

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

  Example

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

−x
: negation

  Negation can work on integers, rational numbers, complex numbers, vectors, matrices and lists.

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

  Example

```
In> - 3
Out> -3
```

x **-** y

subtraction

Subtraction can work on integers, rational numbers, complex numbers, vectors, matrices and lists.

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

Example

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

x ***** y

multiplication

Multiplication can work on integers, rational numbers, complex numbers, vectors, matrices and lists.

Note: In the case of matrices, multiplication is defined in terms of standard matrix product.

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

Example

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

x **/** y

division

Division can work on integers, rational numbers, complex numbers, vectors, matrices and lists.

Note: For matrices division is element-wise.

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

Example

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

x **^** y

exponentiation

Exponentiation can work on integers, rational numbers, complex numbers, vectors, matrices and lists.

Note: In the case of matrices, exponentiation is defined in terms of standard matrix product.

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

Example

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

Div (*x*, *y*)

determine divisor

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

Example

```
In> Div(5,3)
Out> 1
```

See also:

Mod(), *Gcd()*, *Lcm()*

Mod (*x*, *y*)

determine remainder

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

Example

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

See also:

Div(), *Gcd()*, *Lcm()*

Gcd (*n*, *m*)
Gcd (*list*)

greatest common divisor

This function returns the greatest common divisor of `n` and `m` or of all elements of `list`.

See also:

Lcm()

Lcm (*n*, *m*)
Lcm (*list*)

least common multiple

This command returns the least common multiple of `n` and `m` or of all elements of `list`.

Example

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

See also:

Gcd()

n **<<** m
n **>>** m
binary shift operators

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.

Example

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

FromBase(*base*, *"string"*)
conversion of a number from non-decimal base to decimal base

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

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}.

N(*expression*)
try determine numerical approximation of expression

Parameters

- **expression** – expression to evaluate
- **precision** – integer, precision to use

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.

Note: *N()* is a macro. Its argument `expr` will only be evaluated after switching to numeric mode.

Example

```
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(*expr*)
convert floating point numbers to fractions

Parameters **expr** – an expression containing real numbers

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.

Example

```
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(*x*[, *depth=6*])
continued fraction expansion

Parameters

- **x** – number or polynomial to expand in continued fractions
- **depth** – positive integer, maximum required depth

This command returns the continued fraction expansion of `x`, which should be either a floating point number or a polynomial. 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.

Example

```
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 (*frac*)
decimal representation of a rational

Parameters **frac** – a rational number

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.

Example

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

See also:

N()

Floor (x)
round a number downwards

Parameters **x** – a number

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

Example

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

See also:

Ceil(), *Round()*

Ceil (x)
round a number upwards

Parameters **x** – a number

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

Example

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

See also:

Floor(), *Round()*

Round (*x*)
round a number to the nearest integer

Parameters **x** – a number

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.

Example

```
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 (*x*, *y*)
minimum of a number of values

Parameters

- **{y** (*x}*,) – pair of values to determine the minimum of
- **list** – list of values from which the minimum is sought

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.

Example

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

See also:

Max(), *Sum()*

Max (*x*, *y*)
maximum of a number of values

Parameters

- **{y** (*x},*) – pair of values to determine the maximum of
- **list** – list of values from which the maximum is sought

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.

Example

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

See also:

Min(), *Sum()*

Numer(*expr*)

numerator of an expression

Parameters **expr** – expression to determine numerator of

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.

Example

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

See also:

Denom(), *IsRational()*, *IsNumber()*

Denom(*expr*)

denominator of an expression

Parameters **expr** – expression to determine denominator of

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.

Example

```
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(*xlist*, *precision*)
search for integer relations between reals

Parameters

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

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 $a_1 * x_1 + \ldots + a_n * x_n = 0$. The list of integer coefficients is returned. The numbers in "xlist" must evaluate to floating point numbers when the *N()* operator is applied to them.

e1 **<** e2
test for "less than"

Parameters

- **e1** – expression to be compared
- **e2** – expression to be compared

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.

Example

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

See also:

IsNumber(), *IsInfinity()*, *N()*

e1 **>** e2
test for "greater than"

Parameters

- **e1** – expression to be compared
- **e2** – expression to be compared

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.

Example

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

See also:

IsNumber(), *IsInfinity()*, *N()*

e1 **<=** e2
test for "less or equal"

Parameters

- **e1** – expression to be compared
- **e2** – expression to be compared

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.

Example

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

See also:

IsNumber(), *IsInfinity()*, *N()*

e1 **>=** e2
test for "greater or equal"

Parameters

- **e1** – expression to be compared
- **e2** – expression to be compared

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.

Example

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

See also:

IsNumber(), *IsInfinity()*, *N()*

IsZero(*n*)
test whether argument is zero

Parameters **n** – number to test

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

Example

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

See also:

`IsNumber()`, *`IsNotZero()`*

`IsRational`(*expr*)
test whether argument is a rational

Parameters **`expr`** – expression to test

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

Example

```
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()`*

3.2 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`(*x*)
trigonometric sine function

Parameters **`x`** – argument to the function, in radians

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.

`Sin()` is *threaded*.

Example

```
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 (*x*)
trigonometric cosine function

Parameters **x** – argument to the function, in radians

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.

Cos() is *threaded*.

Example

```
In> Cos(1)
Out> Cos(1);
In> N(Cos(1),20)
Out> 0.54030230586813971740;
In> Cos(Pi/4)
Out> Sqrt(1/2);
```

See also:

Sin(), *Tan()*, *ArcSin()*, *ArcCos()*, *ArcTan()*, *N()*, *Pi()*

Tan (*x*)
trigonometric tangent function

Parameters **x** – argument to the function, in radians

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.

Example

```
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 (*x*)
inverse trigonometric function arc-sine

Parameters **x** – argument to the function

This function represents the inverse trigonometric function arcsine. 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 [-Pi/2,Pi/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.

Example

```
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 (*x*)

inverse trigonometric function arc-cosine

Parameters **x** – argument to the function

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, $Cos(Pi/2)$ and $Cos(3*Pi/2)$ 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,Pi] . 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.

Example

```
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 (*x*)

inverse trigonometric function arc-tangent

Parameters **x** – argument to the function

This function represents the inverse trigonometric function arctangent. 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 [-Pi/2,Pi/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.

Example

```
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 (*x*)
exponential function

Parameters **x** – argument to the function

This function calculates e^x where e is the mathematic constant 2.71828... One can use `Exp(1)` to represent e. This function is *threaded function*, meaning that if the argument `x` is a list, the function is applied to all entries in the list.

Example

```
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 (*x*)
natural logarithm

Parameters **x** – argument to the function

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 (-Pi,Pi]. 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.

Example

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

See also:

Exp(), Arg()

Sqrt (*x*)
square root

Parameters **x** – argument to the function

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.

Example

```
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 (*x*)
absolute value or modulus of complex number

Parameters **x** – argument to the function

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 $x = re^{i\phi}$. 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.

Example

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

See also:

Sign(), Arg()

Sign (*x*)
sign of a number

Parameters **x** – argument to the function

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 $Abs(x) * Sign(x) = x$ 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.

Example

```
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(*variable*[, *n=1*]) expression
derivative

Parameters

- **variable** – variable
- **expression** – expression to take derivatives of
- **n** – order

Returns `n`-th derivative of `expression` with respect to `variable`

D(*variable*) expression
derivative

Parameters

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

Returns derivative of `expression` with respect to `variable`

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 to 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.

Example

```
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 (*vector*, *basis*)
curl of a vector field

Parameters

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

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.

Diverge (*vector*, *basis*)
divergence of a vector field

Parameters

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

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.

Integrate (*var*) expr
Integrate (*var*, *x1*, *x2*) expr
integral

Parameters

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

This function integrates the expression {expr} with respect to the variable {var}. In the case of definite integral, the integration is carried out from $var=x1$ to $var=x2$". 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.

Example

```
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(*var*, *val*) expr
limit of an expression

Parameters

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

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.

Example

```
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;
Random numbers
```

Random()
(pseudo-) random number generator

Parameters

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

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:

RandomIntegerMatrix(*rows*, *cols*, *from*, *to*)
generate a matrix of random integers

Parameters

- **rows** – number of rows in matrix
- **cols** – number of cols in matrix
- **from** – lower bound

- **to** – upper bound

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.

Example

```
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(*nr*, *from*, *to*)
generate a vector of random integers

Parameters

- **nr** – number of integers to generate
- **from** – lower bound
- **to** – upper bound

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.

Example

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

See also:

Random(), *RandomPoly()*

RandomPoly(*var*, *deg*, *coefmin*, *coefmax*)
construct a random polynomial

Parameters

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

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.

Example

```
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()*

Add(*val1*, *val2*, ...)
find sum of a list of values

Parameters

- **{val2** (*val1},*) – expressions
- **{list}** – list of expressions to add

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.

Example

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

Sum(*var*, *from*, *to*, *body*)
find sum of a sequence

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

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.

Example

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

See also:

Factorize()

Factorize(*list*)
product of a list of values

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

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".

Example

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

See also:

Sum(), *Apply()*

Taylor(var, at, order) expr
univariate Taylor series expansion

Parameters

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

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".

Example

```
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(var, at, order) expr
Taylor expansion of inverse

Parameters

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

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.

Example

```
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 (*f*, *g*, *var*, *newvar*, *degree*)
solve $h(f(x)) = g(x) + O(x^n)$ for h

Parameters

- **f** – function of var
- **g** – function of var
- **var** – a variable
- **newvar** – a new variable to express the result in
- **degree** – the degree of the required solution

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".

Example

```
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)
```

See also:

InverseTaylor(), Taylor(), *BigOh()*

BigOh (*poly*, *var*, *degree*)
drop all terms of a certain order in a polynomial

Parameters

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

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

Example

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

See also:

`Taylor()`, `InverseTaylor()`

LagrangeInterpolant (*xlist*, *ylist*, *var*)
polynomial interpolation

Parameters

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

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.

Example

```
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()

n!
factorial

Parameters

- **m** – integer
- **n** – integer, half-integer, or list
- **{b** (*a}*,) – numbers

The factorial function {n!} calculates the factorial of integer or half-integer numbers. For nonnegative integers, $n! := n*(n-1)*(n-2)*...*1$. The factorial of half-integers is defined via Euler's Gamma function, $z! := Gamma(z+1)$. 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} = $b! / (a-1)!$. 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.

Example

```
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()

n**!!**
: double factorial

x ******* y
: whatever

Bin (*n*, *m*)
: binomial coefficients

Parameters **{m** (*n}*,) – integers

This function calculates the binomial coefficient "n" above "m", which equals $$n! / (m! * (n-m)!)$$ 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.

Example

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

See also:

(), *Eulerian()*

Eulerian(*n*, *m*)
Eulerian numbers

Parameters **{m**(*n},*) – integers

The Eulerian numbers can be viewed as a generalization of the binomial coefficients, and are given explicitly by $$ Sum(j,0,k+1,(-1)^j*Bin(n+1,j)*(k-j+1)^n) $$.

Example

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

See also:

Bin()

LeviCivita(*list*)
totally anti-symmetric Levi-Civita symbol

Parameters **list** – a list of integers 1 .. n in some order

{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.

Example

```
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(*list*)
get all permutations of a list

Parameters **list** – a list of elements

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

Example

```
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()

Gamma (*x*)
Euler's Gamma function

Parameters

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

{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 $gamma<=>0.57722$ is the lowercase {gamma} in Yacas.

Example

```
In> Gamma(1.3)
Out> Gamma(1.3);
In> N(Gamma(1.3),30)
Out> 0.89747069630627718849375495477 1;
In> Gamma(1.5)
Out> Sqrt(Pi)/2;
In> N(Gamma(1.5),30);
Out> 0.88622692545275801364908374167;
```

See also:

(), *N()*, *gamma()*

Zeta (*x*)
Riemann's Zeta function

Parameters

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

{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.

Example

```
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> 1.64493406684822643647241516664602;
```

See also:

(), *N()*

Bernoulli (*index*)
Bernoulli numbers and polynomials

Parameters

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

{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.

Euler (*index*)
Euler numbers and polynomials

Parameters

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

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

Example

```
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 (*x*)
Lambert's W function

Parameters **x** – expression, argument of the function

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

$$W(z)e^{W(z)} = z$$

The W function is sometimes useful to represent solutions of transcendental equations. For example, the equation $Ln(x)=3*x$ can be "solved" by writing $x= -3*W(-1/3)$. It is also possible to take a derivative or integrate this function "explicitly". For real arguments x, $W(x)$ is real if $x>= -Exp(-1)$. To compute the numeric value of the principal branch of Lambert's W function for real arguments $x>= -Exp(-1)$ to current precision, one can call {N(LambertW(x))} (where the function {N} tries to approximate its argument with a real value).

Example

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

See also:

Exp()

3.3 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(*expr*)
try to simplify an expression

Parameters **expr** – expression to simplify

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.

Example

```
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(*expr*)
simplify expression with nested radicals

Parameters **expr** – an expression containing nested radicals

This function tries to write the expression "expr" as a sum of roots of integers: $Sqrt(e1) + Sqrt(e2) + ...$, where $e1$, $e2$ and so on are natural numbers. The expression "expr" may not contain free variables. It does this by trying all possible combinations for $e1$, $e2$, ... 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.

Example

```
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(*expression*)
Simplify hypergeometric expressions containing factorials

Parameters **expression** – expression to simplify

{FactorialSimplify} takes an expression that may contain factorials, and tries to simplify it. An expression like $ (n+1)! / n! $ would simplify to $(n+1)$. The following steps are taken to simplify:

LnExpand(*expr*)
expand a logarithmic expression using standard logarithm rules

Parameters **expr** – the logarithm of an expression

{LnExpand} takes an expression of the form $Ln(expr)$, and applies logarithm rules to expand this into multiple {Ln} expressions where possible. An expression like $Ln(a*b^n)$ 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 $Ln(x)+2*Ln(3)$.

LnCombine(*expr*)
combine logarithmic expressions using standard logarithm rules

Parameters **expr** – an expression possibly containing multiple {Ln} terms to be combined

{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}.

TrigSimpCombine(*expr*)
combine products of trigonometric functions

Parameters **expr** – expression to simplify

This function applies the product rules of trigonometry, e.g. $Cos(u)*Sin(v) = (1/2)*(Sin(v-u) + Sin(v+u))$. 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.

Example

```
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()*

3.4 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.

3.4.1 Symbolic Solvers

Solve (*eq*, *var*)
solve an equation

Parameters

- **eq** – equation to solve
- **var** – variable to solve for

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:

OldSolve (*eq*, *var*)
old version of {Solve}

Parameters

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

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.

Example

```
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 (*expr*, *var*)
special purpose solver

Parameters

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

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.

Example

```
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 (*var*, *value*, *expr*)
substitute and simplify

Parameters

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

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.

Example

```
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 (*poly*, *var*)
solve a polynomial equation

Parameters

- **poly** – a polynomial in "var"
- **var** – a variable

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.

Example

```
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 (*A*, *b*)
solve a system of equations

Parameters

- **A** – coefficient matrix
- **b** – row vector

{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.

Example

```
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};
Numeric solvers
```

3.4.2 Numeric Solvers

Newton (*expr*, *var*, *initial*, *accuracy*)
solve an equation numerically with Newton's method

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

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.

Example

```
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(*p*)

find the real roots of a polynomial

Parameters **p** – a polynomial in {x}

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.

Example

```
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(*p*)

return the number of real roots of a polynomial

Parameters **p** – a polynomial in {x}

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

Example

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

See also:

FindRealRoots(), *SquareFree()*, *MinimumBound()*, MaximumBound(), *Factor()*

MinimumBound(*p*)

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

Parameters **p** – a polynomial in x

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}.

Example

```
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()*

3.5 Differential Equations

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

OdeSolve(*expr1==expr2*)
general ODE solver

Parameters **expr1,expr2** – expressions containing a function to solve for

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.

Example

```
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(*eqn*, *testsol*)
test the solution of an ODE

Parameters

- **eqn** – equation to test
- **testsol** – test solution

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.

Example

```
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 (*eqn*)
return order of an ODE

Parameters **eqn** – equation

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

Example

```
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()

3.6 Propositional logic theorem prover

CanProve (*proposition*)
try to prove statement

Parameters **proposition** – an expression with logical operations

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 $Not p$ is false. It continues to simplify $Not p$ 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.

Example

```
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()

3.7 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 (*t1*, *t2*)
get dot product of tensors

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

{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.

Example

```
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 (*a*, *b*)
inner product of vectors (deprecated)

Parameters **{b** (*a},*) – vectors of equal length

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.

Example

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

See also:

Dot(), *CrossProduct()*

CrossProduct(*a*, *b*)
outer product of vectors

Parameters **{b** (*a},*) – three-dimensional vectors

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.

Example

```
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(*t1*, *t2*)
get outer tensor product

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

{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.

Example

```
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(*n*)
create a vector with all zeroes

Parameters **n** – length of the vector to return

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

Example

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

See also:

BaseVector(), *ZeroMatrix()*, *IsZeroVector()*

BaseVector(*k*, *n*)
base vector

Parameters

- **k** – index of the base vector to construct
- **n** – dimension of the vector

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.

Example

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

See also:

ZeroVector(), *Identity()*

Identity(*n*)
make identity matrix

Parameters **n** – size of the matrix

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

Example

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

See also:

BaseVector(), *ZeroMatrix()*, *DiagonalMatrix()*

ZeroMatrix(*n*)
make a zero matrix

Parameters

- **n** – number of rows
- **m** – number of columns

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.

Example

```
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(*A*)
extract the diagonal from a matrix

Parameters **A** – matrix

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

Example

```
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(*d*)
construct a diagonal matrix

Parameters **d** – list of values to put on the diagonal

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.

Example

```
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(*W*)
create an orthogonal basis

Parameters **W** – A linearly independent set of row vectors (aka a matrix)

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.

Example

```
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(*W*)
create an orthonormal basis

Parameters **W** – A linearly independent set of row vectors (aka a matrix)

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.

Example

```
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(1/3)},
{-Sqrt(1/6),Sqrt(1/6),Sqrt(2/3)}};
```

See also:

OrthogonalBasis(), *InProduct()*, *Normalize()*

Normalize(*v*)
normalize a vector

Parameters **v** – a vector

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

Example

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

See also:

InProduct(), *CrossProduct()*

Transpose(*M*)
get transpose of a matrix

Parameters **M** – a matrix

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

Example

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

Determinant(*M*)
determinant of a matrix

Parameters **M** – a matrix

Returns the determinant of a matrix M.

Example

```
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(*M*)
trace of a matrix

Parameters **M** – a matrix

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

Example

```
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(*M*)
get inverse of a matrix

Parameters **M** – a matrix

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.

Example

```
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 (*M*, *i*, *j*)
get principal minor of a matrix

Parameters

- **M** – a matrix
- **{j** (*i},*) – positive integers

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.

Example

```
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 (*M*, *i*, *j*)
cofactor of a matrix

Parameters

- **M** – a matrix
- **{j** (*i},*) – positive integers

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

Example

```
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 (*mat*, *n*)
get nth power of a square matrix

Parameters

- **mat** – a square matrix
- **n** – an integer

{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.

SolveMatrix (*M*, *v*)
solve a linear system

Parameters

- **M** – a matrix
- **v** – a vector

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

Example

```
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 (*matrix*, *var*)
get characteristic polynomial of a matrix

Parameters

- **matrix** – a matrix
- **var** – a free variable

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

Example

```
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 (*matrix*)
get eigenvalues of a matrix

Parameters **matrix** – a square matrix

EigenValues returns the eigenvalues of a matrix. The eigenvalues x of a matrix M are the numbers such that $M*v=x*v$ 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).

Example

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

See also:

EigenVectors(), *CharacteristicEquation()*

EigenVectors (*A*, *eigenvalues*)
get eigenvectors of a matrix

Parameters

- **matrix** – a square matrix
- **eigenvalues** – list of eigenvalues as returned by {EigenValues}

{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.

Example

```
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(*matrix*)
get the sparsity of a matrix

Parameters **matrix** – a matrix

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.

Example

```
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(*A*)
find the Cholesky Decomposition

Parameters **A** – a square positive definite matrix

{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.

Example

```
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()*

IsScalar(*expr*)
test for a scalar

Parameters **expr** – a mathematical object

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

Example

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

See also:

IsList(), *IsVector()*, *IsMatrix()*

IsVector ([*pred*], *expr*)
test for a vector

Parameters

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

{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.

Example

```
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 ([*pred*], *expr*)
test for a matrix

Parameters

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

{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.

Example

```
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 ([*pred*], *expr*)
test for a square matrix

Parameters

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

{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.

Example

```
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 (*A*)
test for a Hermitian matrix

Parameters **A** – a square matrix

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.

Example

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

See also:

IsUnitary()

IsOrthogonal (*A*)
test for an orthogonal matrix

Parameters **A** – square matrix

{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).

Example

```
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(*A*)
test for a diagonal matrix

Parameters **A** – a matrix

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

Example

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

IsLowerTriangular(*A*)
test for a lower triangular matrix

Parameters **A** – a matrix

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.

Example

```
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(*A*)
test for a symmetric matrix

Parameters **A** – a matrix

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

Example

```
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(*A*)
test for a skew-symmetric matrix

Parameters **A** – a square matrix

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

Example

```
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(*A*)
test for a unitary matrix

Parameters **A** – a square matrix

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}).

Example

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

See also:

IsHermitian(), *IsSymmetric()*

IsIdempotent(*A*)
test for an idempotent matrix

Parameters **A** – a square matrix

{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.

Example

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

JacobianMatrix(*functions*, *variables*)
calculate the Jacobian matrix of n functions in n variables

Parameters

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

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.

Example

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

VandermondeMatrix(*vector*)
create the Vandermonde matrix

Parameters **vector** – an n-dimensional vector

The function {VandermondeMatrix} calculates the Vandermonde matrix of a vector. The (i,j)-th element of the Vandermonde matrix is defined as $i^(j-1)$.

Example

```
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 (*function*, *var*)
create the Hessian matrix

Parameters

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

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 $ Deriv(x[i]) Deriv(x[j]) f(x) $. 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.

Example

```
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 (*n*)
create a Hilbert matrix

Parameters **n,m** – positive integers

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.

Example

```
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(*n*)
create a Hilbert inverse matrix

Parameters **n** – positive integer

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.

Example

```
    In> PrettyForm(HilbertInverseMatrix(4))
    /                                       \
    | ( 16 )   ( -120 )  ( 240 )   ( -140 )  |
    |                                       |
    | ( -120 ) ( 1200 )  ( -2700 ) ( 1680 )  |
    |                                       |
    | ( 240 )  ( -2700 ) ( 6480 )  ( -4200 ) |
    |                                       |
    | ( -140 ) ( 1680 )  ( -4200 ) ( 2800 )  |
    \                                       /
```

See also:

HilbertMatrix()

ToeplitzMatrix(*N*)
create a Toeplitz matrix

Parameters **N** – an n-dimensional row vector

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.

Example

```
    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 (*func*, *var*)
create the Wronskian matrix

Parameters

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

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.

Example

```
    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 (*poly1*, *poly2*, *variable*)
calculate the Sylvester matrix of two polynomials

Parameters

- **poly1** – polynomial
- **poly2** – polynomial
- **variable** – variable to express the matrix for

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.

Example

```
    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()*

3.8 Operations on polynomials

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

Expand(*expr*)
Expand(*expr*, *var*)
Expand(*expr*, *varlist*)
transform a polynomial to an expanded form

Parameters

- **expr** – a polynomial expression
- **var** – a variable
- **varlist** – a list of variables

This command brings a polynomial in expanded form, in which polynomials are represented in the form $c_0 + c_1x + c_2x^2 + ... + c_nx^n$. 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.

Example

```
In> Expand((1+x)^5)
Out> x^5+5*x^4+10*x^3+10*x^2+5*x+1
In> Expand((1+x-y)^2, x);
Out> x^2+2*(1-y)*x+(1-y)^2
In> Expand((1+x-y)^2, {x,y})
Out> x^2+((-2)*y+2)*x+y^2-2*y+1
```

See also:

ExpandBrackets()

Degree(*expr*[, *var*])
degree of a polynomial

Parameters

- **expr** – a polynomial
- **var** – a variable occurring in {expr}

This command returns the degree of the polynomial `expr` with respect to the variable `var`. 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.

Example

```
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(*expr*, *var*, *order*)
coefficient of a polynomial

Parameters

- **expr** – a polynomial
- **var** – a variable occurring in {expr}
- **order** – integer or list of integers

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.

Example

```
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(*expr*)
content of a univariate polynomial

Parameters **expr** – univariate polynomial

This command determines the content of a univariate polynomial.

Example

```
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 (*expr*)
primitive part of a univariate polynomial

Parameters **expr** – univariate polynomial

This command determines the primitive part of a univariate polynomial. The primitive part is what remains after the content is divided out. So the product of the content and the primitive part equals the original polynomial.

Example

```
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 (*poly*)
leading coefficient of a polynomial

Parameters

- **poly** – a polynomial
- **var** – a variable

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.

Example

```
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 (*poly*)
monic part of a polynomial

Parameters

- **poly** – a polynomial
- **var** – a variable

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.

Example

```
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 (*p*)
return the square-free part of polynomial

Parameters **p** – a polynomial in {x}

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): $$ p[1]* ... * p[m] $$

Example

```
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()*

SquareFreeFactorize (*p*, *x*)
return square-free decomposition of polynomial

Parameters **p** – a polynomial in {x}

Given a polynomial p having square-free decomposition $$ p = p[1]^n[1] * ... * p[m]^n[m] $$ where $p[i]$ are square-free and $n[i+1]>n[i]$, return the list of pairs ($p[i]$, $n[i]$)

Example

```
In> Expand((x+1)^5)
Out> x^5+5*x^4+10*x^3+10*x^2+5*x+1
In> SquareFreeFactorize(%,x)
Out> {{x+1,5}}
```

See also:

Factor()

Horner (*expr*, *var*)
convert a polynomial into the Horner form

Parameters

- **expr** – a polynomial in {var}
- **var** – a variable

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.

Example

```
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 (*expr*)
expand all brackets

Parameters **expr** – an expression

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

Example

```
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 (*coeffs*, *x*)
fast evaluation of polynomials

Parameters

- **coeffs** – a list of coefficients
- **x** – expression

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.

OrthoP(n, x);

Legendre and Jacobi orthogonal polynomials

Parameters

- **n** – degree of polynomial
- **x** – point to evaluate polynomial at
- **{b** (*a},*) – parameters for Jacobi polynomial

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 $$P(n,a,b,x) = (2*n+a+b-1)/(2*n+a+b-2) $$ $$ ((a^2-b^2+x*(2*n+a+b-2)*(n+a+b))/(2*n*(n+a+b))) * P(n-1,a,b,x)$$ $$ -((n+a-1)*(n+b-1)*(2*n+a+b))/(n*(n+a+b)*(2*n+a+b-2))*P(n-2,a,b,x)$$ for $n > 1$, with $P(0,a,b,x) = 1$, $$P(1,a,b,x) = (a-b)/2+x*(1+(a+b)/2)$$.

OrthoH(n, x);

Hermite orthogonal polynomials

Parameters

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

This function evaluates the Hermite polynomial of degree {n} at the point {x}. The Hermite polynomials are orthogonal with respect to the weight function $Exp(-x^2/2)$ on the entire real axis. They satisfy the recurrence relation $$ H(n,x) = 2*x*H(n-1,x) - 2*(n-1)*H(n-2,x) $$ for $n > 1$, with $H(0,x) = 1$, $H(1,x) = 2*x$. Most of the work is performed by the internal function {OrthoPoly}.

Example

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

See also:

OrthoHSum(), *OrthoPoly()*

OrthoG(n, a, x);

Gegenbauer orthogonal polynomials

Parameters

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

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 $(1-x^2)^(a-1/2)$ on the interval [-1,1]. Hence they are connected to the Jacobi polynomials via $$ G(n, a, x) = P(n, a-1/2, a-1/2, x) $$. They satisfy the recurrence relation $$ G(n,a,x) = 2*(1+(a-1)/n)*x*G(n-1,a,x) $$ $$ -(1+2*(a-2)/n)*G(n-2,a,x) $$ for $n>1$, with $G(0,a,x) = 1$, $G(1,a,x) = 2*x$.

OrthoL(n, a, x);
Laguerre orthogonal polynomials

Parameters

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

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 $$ L(n,a,x) = (2+(a-1-x)/n)* L(n-1,a,x) $$ $$ -(1-(a-1)/n)*L(n-2,a,x) $$ for $n>1$, with $L(0,a,x) = 1$, $L(1,a,x) = a + 1 - x$.

OrthoT(n, x);
Chebyshev polynomials

Parameters

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

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 $(1-x^2)^(-1/2)$. Hence they are a special case of the Gegenbauer polynomials $G(n,a,x)$, with $a=0$. They satisfy the recurrence relations $$ T(n,x) = 2* x* T(n-1,x) - T(n-2,x) $$, $$ U(n,x) = 2* x* U(n-1,x) - U(n-2,x) $$ for $n > 1$, with $T(0,x) = 1$, $T(1,x) = x$, $U(0,x) = 1$, $U(1,x) = 2*x$.

Example

```
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(c, x);
sums of series of orthogonal polynomials

Parameters

- **c** – list of coefficients
- **{b** (*a},*) – parameters of specific polynomials
- **x** – point to evaluate polynomial at

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.

Example

```
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`(*name*, *n*, *par*, *x*)
internal function for constructing orthogonal polynomials

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

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. $A+B*x$; the second list contains the coefficients {A,B,C} in the recurrence relation, i.e. $P[n] = (A+B*x)*P[n-1]+C*P[n-2]$. (There are only 3 coefficients in the second list, because none of the polynomials use $C+D*x$ 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`(*name*, *c*, *par*, *x*)
internal function for computing series of orthogonal polynomials

Parameters

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

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()*

3.9 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 (*list*)

Returns the first element of a list

Parameters **list** – a list

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.

Example

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

See also:

Tail(), *Length()*

Tail (*list*)

Returns a list without its first element

Parameters **list** – a list

This function returns `list` without its first element.

Example

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

See also:

Head(), *Length()*

Length (*object*)

The length of a list or string

Parameters **object** – a list or string

Length returns the length of a list or string.

Example

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

See also:

Head(), *Tail()*, *Nth()*, *Count()*

Map (*fn*, *list*)
apply an *n*-ary function to all entries in a list

Parameters

- **fn** – to apply
- **list** – list of lists of arguments

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).

Example

```
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 (*fn*, *list*)
apply a unary function to all entries in a list

Parameters

- **fn** – function to apply
- **list** – list of arguments

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()*.

Example

```
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 (*var*, *n*)
vector of uniquely numbered variable names

Parameters

- **var** – free variable
- **n** – length of the vector

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.

Example

```
In> MakeVector(a,3)
Out> {a1,a2,a3};
```

See also:

RandomIntegerVector(), *ZeroVector()*

Select (*pred*, *list*)
select entries satisfying some predicate

Parameters

- **pred** – a predicate
- **list** – a list of elements to select from

`Select` returns a sublist of `list` which contains all the entries for which the predicate `pred` returns `True` when applied to this entry.

Example

```
In> Select("IsInteger",{a,b,2,c,3,d,4,e,f})
Out> {2,3,4};
```

See also:

Length(), *Find()*, *Count()*

Nth (*list*, *n*)
return the `n`-th element of a list

Parameters

- **list** – list to choose from
- **n** – index of the entry to pick

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]`.

Example

```
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()*

Reverse (*list*)
return the reversed list (without touching the original)

Parameters **list** – list to reverse

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 (*expr1*, *expr2*, ...)
construct a list

Parameters

- **expr1** –
- **expr2** –
- `...` – expressions making up the list

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, ...}`.

Example

```
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 (*list*)
convert a list to a function application

Parameters **list** – list to be converted

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.

Example

```
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 (*expr*)
convert a function application to a list

Parameters **expr** – expression to be converted

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.

Example

```
In> Listify(Cos(x));
Out> {Cos,x};
In> Listify(3*a);
Out> {*,3,a};
```

See also:

List(), *UnList()*, *IsAtom()*

Concat (*list1*, *list2*, ...)
concatenate lists

Parameters

- **list1** –
- **list2** –
- **...** – lists to concatenate

The lists `list1`, `list2`, ... are evaluated and concatenated. The resulting big list is returned.

Example

```
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 (*list*, *n*)
delete an element from a list

Parameters

- **list** – list from which an element should be removed
- **n** – index of the element to remove

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.

Example

```
In> Delete({a,b,c,d,e,f}, 4);
Out> {a,b,c,e,f};
```

See also:

DestructiveDelete(), *Insert()*, *Replace()*

Insert (*list*, *n*, *expr*)
insert an element into a list

Parameters

- **list** – list in which `expr` should be inserted
- **n** – index at which to insert
- **expr** – expression to insert in `list`

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.

Example

```
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()*

Replace (*list*, *n*, *expr*)
replace an entry in a list

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

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)`.

Example

```
In> Replace({a,b,c,d,e,f}, 4, x);
Out> {a,b,c,x,e,f};
```

See also:

Delete(), *Insert()*, *DestructiveReplace()*

FlatCopy (*list*)
copy the top level of a list

Parameters **list** – list to be copied

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).

The following shows a possible way to define a command that reverses a list nondestructively.

Example

```
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(*list*, *expr*)

test whether a list contains a certain element

Parameters

- **list** – list to examine
- **expr** – expression to look for in list

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.

Example

```
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(*list*, *expr*)

get the index at which a certain element occurs

Parameters

- **list** – the list to examine
- **expr** – expression to look for in list

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.

Example

```
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 (*list*, *expr*)
append an entry at the end of a list

Parameters

- **list** – list to append `expr` to
- **expr** – expression to append to the list

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.

Example

```
In> Append({a,b,c,d}, 1);
Out> {a,b,c,d,1};
```

See also:

Concat(), *:()*, *DestructiveAppend()*

RemoveDuplicates (*list*)
remove any duplicates from a list

Parameters **list** – list to act on

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.

Example

```
In> RemoveDuplicates({1,2,3,2,1});
Out> {1,2,3};
In> RemoveDuplicates({a,1,b,1,c,1});
Out> {a,1,b,c};
```

Swap (*list*, *i1*, *i2*)
swap two elements in a list

Parameters

- **list** – the list in which a pair of entries should be swapped
- **i2** (*i1,*) – indices of the entries in `list` to swap

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.

Example

```
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 (*list*, *expr*)
count the number of occurrences of an expression

Parameters

- **list** – the list to examine
- **expr** – expression to look for in `list`

This command counts the number of times that the expression `expr` occurs in `list` and returns this number.

Example

```
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()*

FillList (*expr*, *n*)
fill a list with a certain expression

Parameters

- **expr** – expression to fill the list with
- **n** – the length of the list to construct

This command creates a list of length `n` in which all slots contain the expression `expr` and returns this list.

Example

```
  In> FillList(x, 5);
Out> {x,x,x,x,x};
```

See also:

MakeVector(), *ZeroVector()*, *RandomIntegerVector()*

Drop (*list*, *n*)
Drop (*list*, *-n*)
Drop (*list*, *{m, n}*)
drop a range of elements from a list

Parameters

- **list** – list to act on
- **m** (*n,*) – indices

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`.

Example

```
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()*

Take (*list*, *n*)
Take (*list*, *-n*)
Take (*list*, *{m, n}*)
take a sublist from a list, dropping the rest

Parameters

- **list** – list to act on
- **m** (*n,*) – indices

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.

Example

```
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()*

Partition (*list*, *n*)
partition a list in sublists of equal length

Parameters

- **list** – list to partition
- **n** – length of partitions

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.

Example

```
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()*

Flatten (*expression*, *operator*)
flatten expression w.r.t. some operator

Parameters

- **expression** – an expression
- **operator** – string with the contents of an infix operator.

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.

Example

```
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 (*list*, *operator*, *identity*)
inverse operation of Flatten

Parameters

- **list** – list of objects the operator is to work on
- **operator** – infix operator
- **identity** – identity of the operator

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.

Example

```
In> UnFlatten({a,b,c},"+",0)
Out> a+b+c;
In> UnFlatten({a,b,c},"*",1)
Out> a*b*c;
```

See also:

Flatten()

Type (*expr*)
return the type of an expression

Parameters **expr** – expression to examine

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 `""`.

Example

```
In> Type({a,b,c});
Out> "List";
In> Type(a*(b+c));
Out> "*";
In> Type(123);
Out> "";
```

See also:

IsAtom(), *NrArgs()*

NrArgs(*expr*)
return number of top-level arguments

Parameters **expr** – expression to examine

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.

Example

```
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(*expr*)
VarListArith(*expr*)
VarListSome(*expr*, *list*)
list of variables appearing in an expression

Parameters

- **expr** – an expression
- **list** – a list of function atoms

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 `{+}`.

Example

```
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(*expr*)

list of functions used in an expression

Parameters **expr** – an expression

The command {FuncList(expr)} returns a list of all function atoms that appear in the expression {expr}. The expression is recursively traversed.

Example

```
In> FuncList(x+y*Cos(Ln(x)/x))
Out> {+,*,Cos,/,Ln};
```

See also:

VarList(), *HasExpr()*, *HasFunc()*

FuncListArith(*expr*)

list of functions used in an expression

Parameters **expr** – an expression

FuncListArith is defined through *FuncListSome()* to look only at arithmetic operations {+}, {-}, {*}, {/}.

Example

```
In> FuncListArith(x+y*Cos(Ln(x)/x))
Out> {+,*,Cos};
```

See also:

VarList(), *HasExpr()*, *HasFunc()*

FuncListSome(*expr*, *list*)

list of functions used in an expression

Parameters

- **expr** – an expression
- **list** – list of function atoms to be considered *transparent*

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}}.

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 {+}.

Example

```
In> FuncListSome({a+b*2,c/d},{List})
Out> {List,+,/};
```

See also:

VarList(), *HasExpr()*, *HasFunc()*

PrintList (*list*[, *padding*])
print list with padding

Parameters

- **list** – a list to be printed
- **padding** – (optional) a string

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, standard one is used (comma, space).

Example

```
In> PrintList({a,b,{c, d}}, `` .. ``)
Out> `` a ..  b .. { c ..  d}``;
```

See also:

Write(), *WriteString()*

Table (*body*, *var*, *from*, *to*, *step*)
evaluate while some variable ranges over interval

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

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.

Example

```
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(*list*)
print each entry in a list on a line

Parameters **list** – list to print

This functions writes out the list {list} in a better readable form, by printing every element in the list on a separate line.

Example

```
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()*

3.9.1 Destructive operations

DestructiveAppend(*list*, *expr*)
destructively append an entry to a list

Parameters

- **list** – list to append `expr` to
- **expr** – expression to append to the list

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.

Example

```
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()*

DestructiveDelete(*list*, *n*)
delete an element destructively from a list

Parameters

- **list** – list from which an element should be removed
- **n** – index of the element to remove

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.

Example

```
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(*list*, *n*, *expr*)
insert an element destructively into a list

Parameters

- **list** – list in which `expr` should be inserted
- **n** – index at which to insert
- **expr** – expression to insert in `list`

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.

Example

```
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()*

DestructiveReplace(*list*, *n*, *expr*)
replace an entry destructively in a list

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

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.

Example

```
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()*

DestructiveReverse(*list*)
reverse a list destructively

Parameters **list** – list to reverse

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.

Example

```
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()*

3.9.2 Set operations

Intersection (*l1*, *l2*)
return the intersection of two lists

Parameters **l2** (*l1,*) – two lists

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.

Example

```
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 (*l1*, *l2*)
return the union of two lists

Parameters **l2** (*l1,*) – two lists

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.

Example

```
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 (*l1*, *l2*)
return the difference of two lists

Parameters **{l2** (*l1},*) – two lists

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.

Example

```
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()*

3.9.3 Associative map

Assoc (*key*, *alist*)
return element stored in association list

Parameters

- **key** – string, key under which element is stored
- **alist** – association list to examine

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]}.

Example

```
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 (*alist*)
return the keys in an association list

Parameters **alist** – association list to examine

All the keys in the association list `alist` are assembled in a list and this list is returned.

Example

```
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 AssocDelete(alist, key) AssocDelete(alist, {key, value})

Parameters

- **alist** – association list
- **key** – string, association key
- **value** – value of the key to be deleted

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:

•to make sure that we are deleting the right value;

•if several values are stored on the same key, to delete the specified entry (see the last example).

At most one entry is deleted.

Example

```
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()*

3.9.4 Sorting

BubbleSort (*list*, *compare*)
sort a list

Parameters

- **list** – list to sort
- **compare** – function used to compare elements of {list}

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.

Example

```
In> BubbleSort({4,7,23,53,-2,1}, "<");
Out> {-2,1,4,7,23,53};
```

See also:

HeapSort()

HeapSort (*list*, *compare*)
sort a list

Parameters

- **list** – list to sort
- **compare** – function used to compare elements of {list}

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 {HeapSort} uses the `heapsort algorithm` and is much faster for large lists. The sorting time is proportional to $n * \ln(n)$ where n is the length of the list.

Example

```
In> HeapSort({4,7,23,53,-2,1}, ``>``);
Out> {53,23,7,4,1,-2};
```

See also:

BubbleSort()

3.9.5 Stack and queue operations

Push (*stack*, *expr*)
add an element on top of a stack

Parameters

- **stack** – a list (which serves as the stack container)
- **expr** – expression to push on `stack`

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.

Example

```
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 (*stack*, *n*)
remove an element from a stack

Parameters

- **stack** – a list (which serves as the stack container)
- **n** – index of the element to remove

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.

Example

```
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 (*stack*)
remove an element from the top of a stack

Parameters **stack** – a list (which serves as the stack container)

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.

Example

```
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 (*stack*)
remove an element from the bottom of a stack

Parameters **stack** – a list (which serves as the stack container)

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.

Example

```
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()*

Global stack

GlobalPop()
restore variables using a global stack

GlobalPush()

save variables using a global stack GlobalPop(var) GlobalPop() GlobalPush(expr)

Parameters

- **var** – atom, name of variable to restore from the stack
- **expr** – expression, value to save on the stack

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.

Example

```
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()*

3.10 Graphs

Graph (*edges*)
Graph (*vertices*, *edges*)
construct a graph

See also:

->(), *<->()*

vertex1 **->** vertex2
vertex1 **<->** vertex2
construct an edge

Vertices (*g*)
return list of graph vertices

See also:

Edges(), *Graph()*

Edges (*g*)
return list of graph edges

See also:

Vertices(), *Graph()*

AdjacencyList (*g*)
adjacency list

Parameters **g** – graph

Return adjacency list of graph g.

See also:

AdjacencyMatrix(), *Graph()*

AdjacencyMatrix (*g*)
adjacency matrix

Parameters **g** – graph

Return adjacency matrix of graph `g`.

See also:

AdjacencyList(), *Graph()*

BFS (*g*, *f*)
BFS (*g*, *v*, *f*)
traverse graph in breadth-first order

Traverse graph `g` in breadth-first order, starting from `v` if provided, or from the first vertex. `f` is called for every visited vertex.

See also:

DFS(), *Graph()*

DFS (*g*, *f*)
DFS (*g*, *v*, *f*)
traverse graph in depth-first order

Traverse graph `g` in depth-first order, starting from `v` if provided, or from the first vertex. `f` is called for every visited vertex.

See also:

BFS(), *Graph()*

3.11 Functional operators

These operators can help the user to program in the style of functional programming languages such as Miranda or Haskell.

item `:` list
prepend item to list, or concatenate strings

Parameters

- **item** – an item to be prepended to a list
- **list** – a list
- **string1** – a string
- **string2** – a string

The first form prepends "item" as the first entry to the list "list". The second form concatenates the strings "string1" and "string2".

Example

```
In> a:b:c:{}
Out> {a,b,c};
In> "This":"Is":"A":"String"
Out> "ThisIsAString";
```

See also:

Concat(), *ConcatStrings()*

fn `@` arglist
apply a function

Parameters

- **fn** – function to apply
- **arglist** – single argument, or a list of arguments

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.

Example

```
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()

fn **/@** list
apply a function to all entries in a list

Parameters

- **fn** – function to apply
- **list** – list of arguments

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.

Example

```
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()*

n **..** m
construct a list of consecutive integers

Parameters

- **n** – integer. the first entry in the list
- **m** – integer, the last entry in the list

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}".

NFunction(*"newname"*, *"funcname"*, *{arglist}*)
make wrapper for numeric functions

Parameters

- **"newname"** – name of new function
- **"funcname"** – name of an existing function

- **arglist** – symbolic list of arguments

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}.

Example

```
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()

expr **Where** x==v
substitute result into expression

Parameters

- **expr** – expression to evaluate
- **x** – variable to set
- **v** – value to substitute for variable

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}.

Example

```
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()*

eq1 **AddTo** eq2
add an equation to a set of equations or set of set of equations

Parameters **eq** – (set of) set of equations

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.

Example

```
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()*

3.12 Control flow functions

MaxEvalDepth(*n*)
set the maximum evaluation depth

Parameters **n** – new maximum evaluation depth

Use this command to set the maximum evaluation depth to `n`. The default value is 1000.

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.

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
11067214887500776740702102249872244986396757631391 71
62551893458351062936503742905713846280871969155149 39
71496078691355496484619708421492101247422837559083 64
30609294996716388253479753511833108789215412582914 23
92955373084335320859663305248773674411336138752;
```

Hold(*expr*)
keep expression unevaluated

Parameters **expr** – expression to keep unevaluated

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.

Example

```
In> Echo({ Hold(1+1), "=", 1+1 });
1+1 = 2
Out> True;
```

See also:

Eval(), *HoldArg()*, *UnList()*

Eval(*expr*)
force evaluation of expression

Parameters **expr** – expression to evaluate

This function explicitly requests an evaluation of the expression `expr`, and returns the result of this evaluation.

Example

```
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 (*pred*) expr
loop while a condition is met

Parameters

- **pred** – predicate deciding whether to keep on looping
- **expr** – expression to loop over

Keep on evaluating expr 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 expr is evaluated and then the predicate pred is evaluated again. If it is still True, the expressions expr 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.

Example

```
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 (*pred*) expr
loop until a condition is met

Parameters

- **pred** – predicate deciding whether to stop

- **expr** – expression to loop over

Keep on evaluating `expr` 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 `expr` 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 `expr` at least once, but *`While()`* may not evaluate it 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) expr;` 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.

Example

```
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 (*pred*, *then*[, *else*])
branch point

Parameters

- **pred** – predicate to test
- **then** – expression to evaluate if `pred` is `True`
- **else** – expression to evaluate if `pred` is `False`

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, the *`If()`* expression returns `False`.

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 (*str*)
pass a command to the shell

Parameters **str** – the command to call

The command contained in the string `str` is executed by the underlying operating system. 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.

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 () func(args)
Function (*funcname*, *{args}*) body
declare or define a function

Parameters

- **func(args)** – function declaration, e.g. `f(x,y)`
- **args** – list of atoms, formal arguments to the function
- **body** – expression comprising the body of the function

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 `args`. If the ellipsis `...` follows the last atom in `args`, a function with a variable number of arguments is declared (using *RuleBaseListed()*). Note that the ellipsis cannot be the only element of `args` and *must* be preceded by an atom.

A function with variable number of arguments can take more arguments than elements in `args`; 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 `funcname(_arg1, _arg2) <-- body`. The rule will be declared at precedence 1025. Any previous rules associated with `funcname` (with the same arity) will be discarded. More complicated functions (with more than one body) can be defined by adding more rules.

Example

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`()` func(args)
Macro(*funcname*, *{args}*) body
declare or define a macro

Parameters

- **func(args)** – function declaration, e.g. `f(x,y)`
- **args** – list of atoms, formal arguments to the function
- **body** – expression comprising the body of the function

This does the same as *Function()*, but for macros. One can define a macro easily with this function, instead of having to use *DefMacroRuleBase()*.

Example

The following example defines a looping function

```
In> Macro("myfor",{init,pred,inc,body}) [@init;While(@pred)[@body;@inc;];True;];
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()*

For(*init*, *pred*, *incr*) expr
C-style `for` loop

Parameters

- **init** – expression for performing the initialization
- **pred** – predicate deciding whether to continue the loop
- **incr** – expression to increment the counter
- **expr** – expression to loop over

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 `expr`, `incr`, and `pred`. Finally, `True` is returned.

This command is most often used in a form such as `For(i=1, i<=10, i++) expr`, which evaluates `expr` with `i` subsequently set to 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.

The expression `For(init, pred, incr) expr` is equivalent to `init; While(pred) [expr; incr;]`.

Example

```
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 (*var*, *list*) expr
loop over all entries in list

Parameters

- **var** – looping variable
- **list** – list of values to assign to `var`
- **expr** – expression to evaluate with different values of `var`

The expression `expr` 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`.

Example

```
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 (*fn*, *arglist*)
apply a function to arguments

Parameters

- **fn** – function to apply
- **arglist** – list of arguments

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.

Example

```
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(*expr*, *fn*)
apply a function to all top-level arguments

Parameters

- **expr** – an expression to work on
- **fn** – an operation to perform on each argument

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).

Example

```
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(*from*, *to*) expr
perform a substitution

Parameters

- **from** – expression to be substituted
- **to** – expression to substitute for "from"
- **expr** – expression in which the substitution takes place

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.

Example

```
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(*var*, *val*, *expr*)
temporary assignment during an evaluation

Parameters

- **var** – variable to assign to
- **val** – value to be assigned to "var"
- **expr** – expression to evaluate with "var" equal to "val"

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.

Example

```
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(), */:()*

expression **/:** patterns
local simplification rules

Parameters

- **expression** – an expression
- **patterns** – a list of patterns

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.

Example

```
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(*expression*)
show calling stack after an error occurs

Parameters **expression** – an expression to evaluate

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.

Example

```
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(*expr*)
evaluate with tracing enabled

Parameters **expr** – expression to trace

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.

Example

```
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(*template*) expr
turn on tracing for a particular function

Parameters

- **template** – template showing the operator to trace
- **expr** – expression to evaluate with tracing on

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.

Example

```
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 (*expr*)
measure the time taken by a function

Parameters **expr** – any expression

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> 6.907755278982137052053974364053092622803 3;
```

See also:

GetTime()

3.13 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}.

e1 != e2
test for "not equal"

Parameters **{e2** (*e1},*) – expressions to be compared

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)}.

Example

```
In> 1 != 2;
Out> True;
In> 1 != 1;
Out> False;
```

See also:

=()

e1 = e2
test for equality of expressions

Parameters **{e2** (*e1},*) – expressions to be compared

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.

Example

```
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 expr

logical negation

Parameters **expr** – a boolean expression

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.

Example

```
In> Not True
Out> False;
In> Not False
Out> True;
In> Not(a)
Out> Not a;
```

See also:

And(), Or()

a1 And a2

logical conjunction

Parameters **...**, **{a}** (*a}1,*) – boolean values (may evaluate to *True* or *False*)

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.

Example

```
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()`

a1 Or a2
logical disjunction

Parameters **..., {a}** (*a}1,*) – boolean expressions (may evaluate to *True* or *False*)

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.

Example

```
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(*var*, *expr*)
test whether expression depends on variable

Parameters

- **expr** – expression to test
- **var** – variable to look for in "expr"

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.

Example

```
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 :data:`True` because the
expression {D(x) a*x+b} evaluates to {a}, which does not depend on {x}.
```

See also:

Contains()

IsZeroVector(*list*)
test whether list contains only zeroes

Parameters **list** – list to compare against the zero vector

The only argument given to {IsZeroVector} should be a list. The result is *True* if the list contains only zeroes and *False* otherwise.

Example

```
In> IsZeroVector({0, x, 0});
Out> False;
In> IsZeroVector({x-x, 1 - D(x) x});
Out> True;
```

See also:

IsList(), *ZeroVector()*

IsNonObject (*expr*)
test whether argument is not an {Object()}

Parameters **expr** – the expression to examine

This function returns *True* if "expr" is not of the form {Object(...)} and *False* otherwise.

IsEven (*n*)
test for an even integer

Parameters **n** – integer to test

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.

Example

```
In> IsEven(4);
Out> True;
In> IsEven(-1);
Out> False;
```

See also:

IsOdd(), IsInteger()

IsOdd (*n*)
test for an odd integer

Parameters **n** – integer to test

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.

Example

```
In> IsOdd(4);
Out> False;
In> IsOdd(-1);
Out> True;
```

See also:

IsEven(), IsInteger()

IsEvenFunction (*expression*, *variable*)
Return true if function is an even function, False otherwise

Parameters

- **expression** – mathematical expression
- **variable** – variable

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.

Note: One can decompose a function into an even and an odd part $$ f(x) = f_{even}(x) + f_{odd}(x) $$ where $$ f_{even}(x) = (f(x)+f(-x))/2 $$ and $$ f_{odd}(x) = (f(x)-f(-x))/2 $$

IsFunction(*expr*)
test for a composite object

Parameters **expr** – expression to test

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.

Example

```
In> IsFunction(x+5);
Out> True;
In> IsFunction(x);
Out> False;
```

See also:

IsAtom(), *IsList()*, *Type()*

IsAtom(*expr*)
test for an atom

Parameters **expr** – expression to test

This function tests whether "expr" is an atom. Numbers, strings, and variables are all atoms.

Example

```
In> IsAtom(x+5);
Out> False;
In> IsAtom(5);
Out> True;
```

See also:

IsFunction(), *IsNumber()*, *IsString()*

IsString(*expr*)
test for an string

Parameters **expr** – expression to test

This function tests whether "expr" is a string. A string is a text within quotes, e.g. {"duh"}.

Example

```
In> IsString("duh");
Out> True;
In> IsString(duh);
Out> False;
```

See also:

IsAtom(), *IsNumber()*

IsNumber(*expr*)
test for a number

Parameters **expr** – expression to test

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*.

Example

```
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(*expr*)
test for a list

Parameters **expr** – expression to test

This function tests whether "expr" is a list. A list is a sequence between curly braces, e.g. {{2, 3, 5}}.

Example

```
In> IsList({2,3,5});
Out> True;
In> IsList(2+3+5);
Out> False;
```

See also:

IsFunction()

IsNumericList(*{list}*)
test for a list of numbers

Parameters **{list}** – a list

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(*var*)
test for a bound variable

Parameters **var** – variable to test

This function tests whether the variable "var" is bound, i.e. whether it has been assigned a value. The argument "var" is not evaluated.

Example

```
In> IsBound(x);
Out> False;
In> x := 5;
Out> 5;
In> IsBound(x);
Out> True;
```

See also:

IsAtom()

IsBoolean(*expression*)
test for a Boolean value

Parameters **expression** – an expression

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}.

Example

```
In> IsBoolean(a)
Out> False;
In> IsBoolean(True)
Out> True;
In> IsBoolean(a And b)
Out> True;
```

See also:

True(), *False()*

IsNegativeNumber(*n*)
test for a negative number

Parameters **n** – number to test

{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*.

Example

```
In> IsNegativeNumber(6);
Out> False;
In> IsNegativeNumber(-2.5);
Out> True;
```

See also:

IsNumber(), *IsPositiveNumber()*, *IsNotZero()*, *IsNegativeInteger()*, *IsNegativeReal()*

IsNegativeInteger(*n*)
test for a negative integer

Parameters **n** – integer to test

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*.

Example

```
In> IsNegativeInteger(31);
Out> False;
In> IsNegativeInteger(-2);
Out> True;
```

See also:

IsPositiveInteger(), *IsNonZeroInteger()*, *IsNegativeNumber()*

IsPositiveNumber(*n*)
test for a positive number

Parameters **n** – number to test

{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*.

Example

```
In> IsPositiveNumber(6);
Out> True;
In> IsPositiveNumber(-2.5);
Out> False;
```

See also:

IsNumber(), *IsNegativeNumber()*, *IsNotZero()*, *IsPositiveInteger()*, *IsPositiveReal()*

IsPositiveInteger(*n*)
test for a positive integer

Parameters **n** – integer to test

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*.

Example

```
In> IsPositiveInteger(31);
Out> True;
In> IsPositiveInteger(-2);
Out> False;
```

See also:

IsNegativeInteger(), *IsNonZeroInteger()*, *IsPositiveNumber()*

IsNotZero(*n*)
test for a nonzero number

Parameters **n** – number to test

{IsNotZero(n)} evaluates to *True* if {n} is not zero. In case {n} is not a number, the function returns *False*.

Example

```
In> IsNotZero(3.25);
Out> True;
In> IsNotZero(0);
Out> False;
```

See also:

IsNumber(), *IsPositiveNumber()*, *IsNegativeNumber()*, *IsNonZeroInteger()*

IsNonZeroInteger(*n*)
test for a nonzero integer

Parameters **n** – integer to test

This function tests whether the integer {n} is not zero. If {n} is not an integer, the result is *False*.

Example

```
In> IsNonZeroInteger(0)
Out> False;
In> IsNonZeroInteger(-2)
Out> True;
```

See also:

IsPositiveInteger(), *IsNegativeInteger()*, *IsNotZero()*

IsInfinity(*expr*)
test for an infinity

Parameters **expr** – expression to test

This function tests whether {expr} is an infinity. This is only the case if {expr} is either {Infinity} or {-Infinity}.

Example

```
In> IsInfinity(10^1000);
Out> False;
In> IsInfinity(-Infinity);
Out> True;
```

See also:

Integer()

IsPositiveReal(*expr*)
test for a numerically positive value

Parameters **expr** – expression to test

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.

Example

```
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(*expr*)
test for a numerically negative value

Parameters **expr** – expression to test

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.

Example

```
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(*expr*)

test for a constant

Parameters **expr** – some expression

{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.

Example

```
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(*z*)

test for a Gaussian integer

Parameters **z** – a complex or real number

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 $a+b*I$ is a Gaussian integer if and only if a and b are integers.

Example

```
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 (*x*, *expr*)
match an expression to a polynomial of degree one in a variable

Parameters

- **x** – variable to express the univariate polynomial in
- **expr** – expression to match

{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.

Example

```
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 (*expr*, *x*)
check for expression containing a subexpression

Parameters

- **expr** – an expression
- **x** – a subexpression to be found
- **list** – list of function atoms to be considered "transparent"

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 "{+}".

Example

```
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 (*expr*, *func*)
check for expression containing a function

Parameters

- **expr** – an expression
- **func** – a function atom to be found
- **list** – list of function atoms to be considered "transparent"

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 "{+}".

Example

```
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()*

3.14 Constants

3.14.1 Yacas-specific constants

%
previous result

% 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.

Example

```
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()

EndOfFile
end-of-file marker

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.

3.14.2 Mathematical constants

True
False
boolean constants representing true and false

True and *False* are typically a result of boolean expressions such as `2 < 3` or `True And False`.

See also:

`And(),Or(),Not()`

Infinity
constant representing mathematical infinity

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*.

Example

```
In> 2*Infinity
Out> Infinity;
In> 2<Infinity
Out> True;
```

Pi
mathematical constant, π

The *constant* represents the number π. It is available symbolically as `Pi` or numerically through `N(Pi)`.

This is a *cached constant* which is recalculated only when precision is increased.

Example

```
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

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.

Example

```
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

The *constant* represents the golden ratio

$$\phi := \frac{1+\sqrt{5}}{2} = 1.6180339887\ldots$$

It is available symbolically as `GoldenRatio` or numerically through `N(GoldenRatio)`.

This is a *cached constant* which is recalculated only when precision is increased.

Example

```
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

The *constant* represents the Catalan's constant

$$G := \beta(2) = \sum_{n=0}^{\infty} \frac{-1^n}{(2n+1)^2} = 0.9159655941\ldots$$

It is available symbolically as `Catalan` or numerically through `N(Catalan)`.

This is a *cached constant* which is recalculated only when precision is increased.

Example

```
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–Mascheroni constant γ

The *constant* represents the Euler–Mascheroni constant

$$\gamma := \lim_{n\to\infty}\left(-\ln(n) + \sum_{k=1}^{n}\frac{1}{k}\right) = 0.5772156649\ldots$$

It is available symbolically as `gamma` or numerically through `N(gamma)`.

This is a *cached constant* which is recalculated only when precision is increased.

Note: Euler's $\Gamma(x)$ function is the capitalized *Gamma()* in yacas.

Example

```
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()*

3.15 Variables

var **:=** expr
var[i] **:=** expr
varlist **:=** exprlist
fn **:=** expr
assign a variable or a list; define a function

var := expr {var1, var2, ...} := {expr1, expr2, ...} var[i] := expr fn(arg1, arg2, ...) := expr

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
- **arg2** (*arg1,*) – atoms, names of arguments of the new function {fn}

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) := x^2} removes any rules previously associated with {fn(x)} and defines the rule {fn(_x) <– x^2}. 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

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^21/5109094217170944000
-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^21/5109094217170944000
-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^21/5109094217170944000
0-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^21/5109094217170944000
0-x^23/25852016738884976640000+x^25/
15511210043330985984000000;
```

See also:

Set(), *Clear()*, [](), *Rule()*, *Infix()*, *Eval()*, *Function()*

Set (*var*, *exp*)
assignment

Parameters

- **var** – variable which should be assigned
- **exp** – expression to assign to the variable

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.

Example

```
In> Set(a, Sin(x)+3);
Out> True;
In> a;
Out> Sin(x)+3;
```

See also:

Clear(), *:=()*

Clear (*var*, ...)
undo an assignment

Parameters **var** – name of the variable to be cleared

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.

Example

```
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 (*var*, ...)
declare new local variables

Parameters **var** – name of the variable to be declared as local

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()*.

Example

```
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()*

var**++**
increment variable

Parameters **var** – variable to increment

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.

Example

```
In> x := 5;
Out> 5;
In> x++;
Out> True;
In> x;
Out> 6;
```

See also:

--(), *:=()*

var**--**
decrement variable

Parameters **var** – variable to decrement

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.

Example

```
In> x := 5;
Out> 5;
In> x--;
Out> True;
In> x;
Out> 4;
```

See also:

++(), *:=()*

Object(*"pred"*, *expr*)
create an incomplete type

Parameters

- **pred** – name of the predicate to apply
- **expr** – expression on which `pred` should be applied

This function returns "obj" as soon as "pred" returns *True* when applied on "obj". This is used to declare so-called incomplete types.

Example

```
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(*var*, *value*)
global variable is to be evaluated lazily

Parameters

- **var** – variable (held argument)
- **value** – value to be set to (evaluated before it is assigned)

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
```

Example

```
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

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.

Example

```
In> UniqueConstant()
Out> C9
In>  UniqueConstant()
Out> C10
```

See also:

LocalSymbols()

LocalSymbols(*var1*, *var2*, ...) expr
create unique local symbols with given prefix

Parameters

- **var2, ..** (*var1,*) – atoms, symbols to be made local
- **expr** – expression to execute

Given the symbols passed as the first arguments to *LocalSymbols()*, a set of unique local symbols will be created, typically of the form `$<symbol><number>`, where `symbol` was the symbol entered by the user, and `number` is a unique number. This scheme is 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.).

Example

```
In> LocalSymbols(a,b)a+b
Out> $a6+ $b6;
```

See also:

UniqueConstant()

3.16 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(*expr*)

print an expression in LISP-format

Parameters **expr** – expression to be printed in LISP-format

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.

Example

```
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(*item*)

high-level printing routine

Parameters

- **item** – the item to be printed
- **list** – a list of items to be printed

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*.

Example

```
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(*expr*)

print an expression nicely with ASCII art

Parameters **expr** – an expression

{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.

Example

```
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(*expr*)

print an evaluation nicely with ASCII art

Parameters **expr** – an expression

Show an evaluation in a nice way, using {PrettyPrinter'Set} to show 'input = output'.

Example

```
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(*expr*)
export expressions to $LaTeX$

Parameters **expr** – an expression to be exported

{TeXForm} returns a string containing a $LaTeX$ representation of the Yacas expression {expr}. Currently the exporter handles most expression types but not all.

CForm(*expr*)
export expression to C++ code

Parameters **expr** – expression to be exported

{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.

IsCFormable(*expr*)
check possibility to export expression to C++ code

Parameters

- **expr** – expression to be exported (this argument is not evaluated)
- **funclist** – list of "allowed" function atoms

{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.

Example

```
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 :data:`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(*expr*, ...)
low-level printing routine

Parameters **expr** – expression to be printed

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*.

Example

```
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(*string*)
low-level printing routine for strings

Parameters **string** – the string to be printed

The expression "string" is evaluated and written to the current output without quotation marks. The argument should be a string. WriteString always returns True.

Example

```
    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

Parameters **nr** – the number of spaces to print

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.

Example

```
    In> Space(5);
    Out> True;
```

See also:

Echo(), *Write()*, *NewLine()*

NewLine()
print one or more newline characters

Parameters **nr** – the number of newline characters to print

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.

Example

```
    In> NewLine();
    Out> True;
```

See also:

Echo(), *Write()*, *Space()*

FromFile(name) body
connect current input to a file

Parameters

- **name** – string, the name of the file to read
- **body** – expression to be evaluated

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.

Example

```
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(str) body;
connect current input to a string

Parameters

- **str** – a string containing the text to parse
- **body** – expression to be evaluated

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.

Example

```
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(name) body
connect current output to a file

Parameters

- **name** – string, the name of the file to write the result to
- **body** – expression to be evaluated

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.

Example

```
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() body

connect current output to a string

Parameters **body** – expression to be evaluated

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.

Example

```
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

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.

Example

```
In> FromString("2+5;") Read();
Out> 2+5;
In> FromString("") Read();
Out> EndOfFile;
```

See also:

FromFile(),FromString(),*LispRead()*,*ReadToken()*,*Write()*

ToStdout() body
select initial output stream for output

Parameters **body** – expression to be evaluated

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.

ReadCmdLineString(*prompt*)
read an expression from command line and return in string

Parameters **prompt** – string representing the prompt shown on screen

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.

Example

```
The following defines a function that when invoked keeps asking
for an expression (the <i>read</i> step), and then takes
the derivative of it (the <i>eval</i> step) and then
uses PrettyForm to display the result (the <i>print</i> 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

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.

Example

```
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;
```

See also:

FromFile(),FromString(),*Read()*,*ReadToken()*,*FullForm()*,*LispReadListed()*

LispReadListed()

read expressions in LISP syntax

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).

Example

```
In> FromString("(+ a b)")LispReadListed()
Out> {+,a,b};
```

See also:

FromFile(),FromString(),*Read()*,*ReadToken()*,*FullForm()*,*LispRead()*

ReadToken()

read a token from current input

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.

Example

```
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(*name*)
evaluate all expressions in a file

Parameters **name** – string, name of the file to load

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(*name*)
load a file, but not twice

Parameters **name** – string, name of the file to load

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(*name*)
load a {.def} file

Parameters **name** – string, name of the file (without {.def} suffix)

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(*name*)
find a file in the current path

Parameters **name** – string, name of the file or directory to find

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(*name*)
execute commands between {<?} and {?>} in file

Parameters **name** – string, name of the file to "patch"

{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

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").

Example

```
In> WriteString("First line" : Nl() : "Second line" : Nl());
First line
Second line
Out> True;
```

See also:

NewLine()

V(*expression*)
set verbose output mode

Parameters **expression** – expression to be evaluated in verbose mode

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*.

Example

```
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()*, *InVerboseMode()*

InVerboseMode()
check for verbose output mode

In verbose mode, {InVerboseMode()} will return *True*, otherwise it will return *False*.

Example

```
In> InVerboseMode()
Out> False
In> V(InVerboseMode())
Out> True
```

See also:

Echo(), *N()*, *OldSolve()*, *V()*

Plot2D (*f(x)*)
adaptive two-dimensional plotting

Parameters

- **f(x)** – unevaluated expression containing one variables (function to be plotted)
- **list** – list of functions to plot
- **{b** (*a},*) – numbers, plotting range in the x coordinate
- **option** – atom, option name
- **value** – atom, number or string (value of option)

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 <*yacasdoc://Algo/3/1/*>). The initial grid of {points+1} points is refined between any grid points a, b if the integral $Integrate(x,a,b)f(x)$ 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.)

See also:

V(), *NFunction()*, *Plot3DS()*

Plot3DS (*f(x, y)*)
three-dimensional (surface) plotting

Parameters

- **f(x,y)** – unevaluated expression containing two variables (function to be plotted)
- **list** – list of functions to plot
- **{b}, {c}, {d**(*a}*,) – numbers, plotting ranges in the x and y coordinates
- **option** – atom, option name
- **value** – atom, number or string (value of option)

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 <*yacasdoc://Algo/3/1/*>). 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.

Example

```
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 (*xmltext*)

convert XML strings to tag objects

Parameters **xmltext** – string containing some XML tokens

{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.

Example

```
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()

XmlTokenizer()
select the default syntax tokenizer for parsing the input

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:

- after switching to {XmlTokenizer} the {In>} prompt disappeared; the user typed {<a>} and the {Out>} prompt with the resulting expression appeared.
- The resulting expression is an atom with the string representation {<a>}; it is not a string.

See also:

OMRead(), *TrapError()*, *XmlExplodeTag()*, *ReadToken()*, FromFile(), FromString()

DefaultTokenizer()
select the default syntax tokenizer for parsing the input

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.

See also:

OMRead(), *TrapError()*, *XmlExplodeTag()*, *ReadToken()*, FromFile(), FromString()

OMForm(*expression*)

convert Yacas expression to OpenMath

Parameters **expression** – expression to convert

{OMForm} prints an OpenMath representation of the input parameter {expression} to standard output. 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()*

OMRead()
read OpenMath expression and convert to Yacas

Parameters **expression** – expression to convert

{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);
```

See also:

XmlTokenizer(), *XmlExplodeTag()*, *OMDef()*

OMDef(*yacasForm*, *cd*, *name*)
define translations from Yacas to OpenMath and vice-versa.

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

{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"/></OMOBJ> ")OMRead() 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 as {} , but is redefined now as EmptySet Out> True In> FromString("<OMOBJ><OMS cd="set1"

name=”emptyset”/></OMOBJ> ”)OMRead() 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.

Example

```
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></O
Out> Sqrt(16)
In> FromString("<OMOBJ><OMA><OMS cd=\"arith1\" name=\"root\"/><OMI>16</OMI><OMI>3</OMI></OMA></O
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=Righ
      |{ $, _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))OMRead()
Out> Limit(x,0,Right)1/x
In> %
Out> Infinity
```

See also:

OMRead(), *OMForm()*

3.17 String manipulation

StringMid'Set (*index*, *substring*, *string*)
change a substring

Parameters

- **index** – index of substring to get
- **substring** – substring to store
- **string** – string to store substring in

Set (change) a part of a string. It leaves the original alone, returning a new changed copy.

Example

```
In> StringMid'Set(3,"XY","abcdef")
Out> "abXYef";
```

See also:

StringMid'Get(), *Length()*

StringMid'Get (*index*, *length*, *string*)
retrieve a substring

Parameters

- **index** – index of substring to get
- **length** – length of substring to get
- **string** – string to get substring from

{StringMid'Get} returns a part of a string. Substrings can also be accessed using the {[]} operator.

Example

```
In> StringMid'Get(3,2,"abcdef")
Out> "cd";
In> "abcdefg"[2 .. 4]
Out> "bcd";
```

See also:

StringMid'Set(), *Length()*

Atom (*"string"*)
convert string to atom

Parameters **"string"** – a string

Returns an atom with the string representation given as the evaluated argument. Example: {Atom("foo");} returns {foo}.

Example

```
In> Atom("a")
Out> a;
```

See also:

String()

String (*atom*)
convert atom to string

Parameters **atom** – an atom

{String} is the inverse of {Atom}: turns {atom} into {"atom"}.

Example

```
In> String(a)
Out> "a";
```

See also:

Atom()

ConcatStrings (*strings*)
concatenate strings

Parameters **strings** – one or more strings

Concatenates strings.

Example

```
In> ConcatStrings("a","b","c")
Out> "abc";
```

See also:

Concat()

PatchString (*string*)
execute commands between {<?} and {?>} in strings

Parameters **string** – a string to patch

This function does the same as PatchLoad, but it works on a string instead of on the contents of a text file. See PatchLoad for more details.

Example

```
In> PatchString("Two plus three is <? Write(2+3); ?> ");
Out> "Two plus three is 5 ";
```

See also:

PatchLoad()

3.18 Probability and Statistics

3.18.1 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 (*p*)
Bernoulli distribution

Parameters **p** – number, probability of an event in a single trial

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 (*p*, *n*)
binomial distribution

Parameters

- **p** – number, probability to observe an event in single trial
- **n** – number of trials

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 (*m*)
Student's t distribution

Parameters **{m}** – integer, number of degrees of freedom

PDF (*dist*, *x*)
probability density function

Parameters

- **dist** – a distribution type
- **x** – a value of random variable

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()`

3.18.2 Statistics

ChiSquareTest (*observed*, *expected*)
Pearson's ChiSquare test

Parameters

- **observed** – list of observed frequencies
- **expected** – list of expected frequencies
- **params** – number of estimated parameters

`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 $n_i = np_i$ 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()`.

3.19 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(*n*)
test for a prime number

Parameters **n** – integer to test

IsComposite(*n*)
test for a composite number

Parameters **n** – positive integer

IsCoprime(*m*, *n*)
test if integers are coprime

Parameters

- **m** – positive integer
- **n** – positive integer
- **list** – list of positive integers

IsSquareFree(*n*)
test for a square-free number

Parameters **n** – positive integer

IsPrimePower(*n*)
test for a power of a prime number

Parameters **n** – integer to test

NextPrime(*i*)
generate a prime following a number

Parameters **i** – integer value

IsTwinPrime(*n*)
test for a twin prime

Parameters **n** – positive integer

IsIrregularPrime(*n*)
test for an irregular prime

Parameters **n** – positive integer

IsCarmichaelNumber(*n*)
test for a Carmichael number

Parameters **n** – positive integer

Factors(*x*)
factorization

Parameters **x** – integer or univariate polynomial

IsAmicablePair(*m*, *n*)
test for a pair of amicable numbers

Parameters

- **m** – positive integer
- **n** – positive integer

Factor(*x*)
factorization, in pretty form

Parameters **x** – integer or univariate polynomial

Divisors(*n*)
number of divisors

Parameters **n** – positive integer

DivisorsSum(*n*)
the sum of divisors

Parameters **n** – positive integer

ProperDivisors(*n*)
the number of proper divisors

Parameters **n** – positive integer

ProperDivisorsSum(*n*)
the sum of proper divisors

Parameters **n** – positive integer

Moebius(*n*)
the Moebius function

Parameters **n** – positive integer

CatalanNumber(*n*)
return the n-th Catalan Number

Parameters **n** – positive integer

FermatNumber(*n*)
return the n-th Fermat Number

Parameters **n** – positive integer

HarmonicNumber(*n*)
return the n-th Harmonic Number

Parameters

- **n** – positive integer
- **r** – positive integer

StirlingNumber1(*n*, *m*)
return the n,m-th Stirling Number of the first kind

Parameters

- **n** – positive integers
- **m** – positive integers

StirlingNumber1(*n*, *m*)
return the n, m-th Stirling Number of the second kind

Parameters

- **n** – positive integer
- **m** – positive integer

DivisorsList(*n*)
the list of divisors

Parameters **n** – positive integer

SquareFreeDivisorsList(*n*)
the list of square-free divisors

Parameters **n** – positive integer

MoebiusDivisorsList(*n*)
the list of divisors and Moebius values

Parameters **n** – positive integer

SumForDivisors(*var*, *n*, *expr*)
loop over divisors

Parameters

- **var** – atom, variable name
- **n** – positive integer
- **expr** – expression depending on var

RamanujanSum(*k*, *n*)
compute the Ramanujan's sum

Parameters

- **k** – positive integer
- **n** – positive integer

This function computes the Ramanujan's sum, i.e. the sum of the n-th powers of the k-th primitive roots of the unit:

$$\sum_{l=1}^{k} \frac{\exp(2ln\pi\imath)}{k}$$

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.

Todo

check the definition

PAdicExpand(*n*, *p*)
p-adic expansion

Parameters

- **n** – number or polynomial to expand
- **p** – base to expand in

IsQuadraticResidue(*m*, *n*)
functions related to finite groups

Parameters

- **m** – integer
- **n** – odd positive integer

GaussianFactors(*z*)
factorization in Gaussian integers

Parameters **z** – Gaussian integer

GaussianNorm(*z*)
norm of a Gaussian integer

Parameters **z** – Gaussian integer

IsGaussianUnit(*z*)
test for a Gaussian unit

Parameters **z** – a Gaussian integer

IsGaussianPrime(*z*)
test for a Gaussian prime

Parameters **z** – a complex or real number

GaussianGcd(*z*, *w*)
greatest common divisor in Gaussian integers

Parameters

- **z** – Gaussian integer
- **w** – Gaussian integer

3.20 Numerical methods

NIntegrate(*x*, *x0*, *x1*) expr
numerical integration

Parameters

- **x** – integration variable
- **x0** – lower integration limit
- **x1** – upper integration limit
- **expr** – integrand

Numerically integrate `expr` over `x` from `x0` to `x1`.

See also:

Integrate()

3.21 Functions related to programming in Yacas

3.21.1 Introduction

This document aims to be a reference for functions that are useful when programming in {Yacas}, but which are not necessarily useful when using {Yacas}. There is another document that describes the functions that are useful from a users point of view.

3.21.2 Programming

This chapter describes functions useful for writing Yacas scripts.

```
/* --- Start of comment
*/ --- end of comment
// --- Beginning of one-line comment

 /* comment */
 // comment
```

Introduce a comment block in a source file, similar to C++ comments. `//` makes everything until the end of the line a comment, while `/*` and `*/` may delimit a multi-line comment.

Example

```
a+b; // get result
a + /* add them */ b;
```

Prog(*expr1*, *expr2*, ...)
block of statements

param expr1 expression

The {Prog} and the {[...]} construct have the same effect: they evaluate all arguments in order and return the result of the last evaluated expression.

{Prog(a,b);} is the same as typing {[a;b;];} and is very useful for writing out function bodies. The {[...]} construct is a syntactically nicer version of the {Prog} call; it is converted into {Prog(...)} during the parsing stage.

Bodied(*op*, *precedence*)
declare op as *bodied function*

Parameters

- **op** – string, the name of a function
- **precedence** – nonnegative integer (evaluated)

Declares a special syntax for the function to be parsed as a *bodied function*.

For(pre, condition, post) statement;

Here the function `For` has 4 arguments and the last argument is placed outside the parentheses.

The `precedence` of a "bodied" function refers to how tightly the last argument is bound to the parentheses. This makes a difference when the last argument contains other operators. For example, when taking the derivative D(x) Sin(x)+Cos(x) both {Sin} and {Cos} are under the derivative because the bodied function {D} binds less tightly than the infix operator "{+}".

See also:

IsBodied(), *OpPrecedence()*

Infix(*op*[, *precedence*])
define function syntax (infix operator)

Parameters

- **op** – string, the name of a function
- **precedence** – nonnegative integer (evaluated)

Declares a special syntax for the function to be parsed as a bodied, infix, postfix, or prefix operator.

"Infix" functions must have two arguments and are syntactically placed between their arguments. Names of infix functions can be arbitrary, although for reasons of readability they are usually made of non-alphabetic characters.

See also:

IsBodied(), *OpPrecedence()*

Postfix(*op*[, *precedence*])
define function syntax (postfix operator)

Parameters

- **op** – string, the name of a function
- **precedence** – nonnegative integer (evaluated)

Declares a special syntax for the function to be parsed as a bodied, infix, postfix, or prefix operator.

"Postfix" functions must have one argument and are syntactically placed after their argument.

See also:

IsBodied(), *OpPrecedence()*

Prefix(*op*[, *precedence*])
define function syntax (prefix operator)

Parameters

- **op** – string, the name of a function
- **precedence** – nonnegative integer (evaluated)

Declares a special syntax for the function to be parsed as a bodied, infix, postfix, or prefix operator.

"Prefix" functions must have one argument and are syntactically placed before their argument.

Function name can be any string but meaningful usage and readability would require it to be either made up entirely of letters or entirely of non-letter characters (such as "+", ":" etc.). Precedence is optional (will be set to 0 by default).

Example

```
In> YY x := x+1;
CommandLine(1) : Error parsing expression

In> Prefix("YY", 2)
Out> True;
In> YY x := x+1;
Out> True;
In> YY YY 2*3
Out> 12;
In> Infix("##", 5)
```

```
Out> True;
In> a ## b ## c
Out> a##b##c;
```

Note that, due to a current parser limitation, a function atom that is declared prefix cannot be used by itself as an argument.

```
In> YY
CommandLine(1) : Error parsing expression
```

See also:

IsBodied(), *OpPrecedence()*

IsBodied(*op*)
check for function syntax

Parameters **op** – string, the name of a function

Check whether the function with given name {"op"} has been declared as a "bodied", infix, postfix, or prefix operator, and return *True* or *False*.

IsInfix(*op*)
check for function syntax

Parameters **op** – string, the name of a function

Check whether the function with given name {"op"} has been declared as a "bodied", infix, postfix, or prefix operator, and return *True* or *False*.

IsPostfix(*op*)
check for function syntax

Parameters **op** – string, the name of a function

Check whether the function with given name {"op"} has been declared as a "bodied", infix, postfix, or prefix operator, and return *True* or *False*.

IsPrefix(*op*)
check for function syntax

Parameters **op** – string, the name of a function

Check whether the function with given name {"op"} has been declared as a "bodied", infix, postfix, or prefix operator, and return *True* or *False*.

Example

```
In> IsInfix("+");
Out> True;
In> IsBodied("While");
Out> True;
In> IsBodied("Sin");
Out> False;
In> IsPostfix("!");
Out> True;
```

See also:

Bodied(), *OpPrecedence()*

OpPrecedence(*op*)
get operator precedence

Parameters **op** – string, the name of a function

Returns the precedence of the function named "op" which should have been declared as a bodied function or an infix, postfix, or prefix operator. Generates an error message if the string str does not represent a type of function that can have precedence.

For infix operators, right precedence can differ from left precedence. Bodied functions and prefix operators cannot have left precedence, while postfix operators cannot have right precedence; for these operators, there is only one value of precedence.

OpLeftPrecedence(*op*)
get operator precedence

Parameters **op** – string, the name of a function

Returns the precedence of the function named "op" which should have been declared as a bodied function or an infix, postfix, or prefix operator. Generates an error message if the string str does not represent a type of function that can have precedence.

For infix operators, right precedence can differ from left precedence. Bodied functions and prefix operators cannot have left precedence, while postfix operators cannot have right precedence; for these operators, there is only one value of precedence.

OpRightPrecedence(*op*)
get operator precedence

Parameters **op** (*string*) – name of a function

Returns the precedence of the function named "op" which should have been declared as a bodied function or an infix, postfix, or prefix operator. Generates an error message if the string str does not represent a type of function that can have precedence.

For infix operators, right precedence can differ from left precedence. Bodied functions and prefix operators cannot have left precedence, while postfix operators cannot have right precedence; for these operators, there is only one value of precedence.

Example

```
In> OpPrecedence("+")
Out> 6;
In> OpLeftPrecedence("!")
Out> 0;
```

RightAssociative(*op*)
declare associativity

Parameters **op** – string, the name of a function

This makes the operator right-associative. For example:

```
RightAssociative("*")
```

would make multiplication right-associative. Take care not to abuse this function, because the reverse, making an infix operator left-associative, is not implemented. (All infix operators are by default left-associative until they are declared to be right-associative.)

See also:

OpPrecedence()

LeftPrecedence(*op*, *precedence*)
set operator precedence

Parameters

- **op** – string, the name of a function
- **precedence** – nonnegative integer

{"op"} should be an infix operator. This function call tells the infix expression printer to bracket the left or right hand side of the expression if its precedence is larger than precedence.

This functionality was required in order to display expressions like {a-(b-c)} correctly. Thus, {a+b+c} is the same as {a+(b+c)}, but {a-(b-c)} is not the same as {a-b-c}.

Note that the left and right precedence of an infix operator does not affect the way Yacas interprets expressions typed by the user. You cannot make Yacas parse {a-b-c} as {a-(b-c)} unless you declare the operator "{-}" to be right-associative.

See also:

OpPrecedence(), *OpLeftPrecedence()*, *OpRightPrecedence()*, *RightAssociative()*

RightPrecedence()
set operator precedence(op, precedence)

Parameters

- **op** – string, the name of a function
- **precedence** – nonnegative integer

{"op"} should be an infix operator. This function call tells the infix expression printer to bracket the left or right hand side of the expression if its precedence is larger than precedence.

This functionality was required in order to display expressions like {a-(b-c)} correctly. Thus, {a+b+c} is the same as {a+(b+c)}, but {a-(b-c)} is not the same as {a-b-c}.

Note that the left and right precedence of an infix operator does not affect the way Yacas interprets expressions typed by the user. You cannot make Yacas parse {a-b-c} as {a-(b-c)} unless you declare the operator "{-}" to be right-associative.

See also:

OpPrecedence(), *OpLeftPrecedence()*, *OpRightPrecedence()*, *RightAssociative()*

RuleBase(*name*, *params*)
define function with a fixed number of arguments

Parameters

- **name** – string, name of function
- **params** – list of arguments to function

Define a new rules table entry for a function "name", with {params} as the parameter list. Name can be either a string or simple atom.

In the context of the transformation rule declaration facilities this is a useful function in that it allows the stating of argument names that can he used with HoldArg.

Functions can be overloaded: the same function can be defined with different number of arguments.

See also:

MacroRuleBase(), *RuleBaseListed()*, *MacroRuleBaseListed()*, *HoldArg()*, *Retract()*

RuleBaseListed(*name*, *params*)
define function with variable number of arguments

Parameters

- **name** – string, name of function
- **params** – list of arguments to function

The command {RuleBaseListed} defines a new function. It essentially works the same way as {RuleBase}, except that it declares a new function with a variable number of arguments. The list of parameters {params} determines the smallest number of arguments that the new function will accept. If the number of arguments passed to the new function is larger than the number of parameters in {params}, then the last argument actually passed to the new function will be a list containing all the remaining arguments.

A function defined using {RuleBaseListed} will appear to have the arity equal to the number of parameters in the {param} list, and it can accept any number of arguments greater or equal than that. As a consequence, it will be impossible to define a new function with the same name and with a greater arity.

The function body will know that the function is passed more arguments than the length of the {param} list, because the last argument will then be a list. The rest then works like a {RuleBase}-defined function with a fixed number of arguments. Transformation rules can be defined for the new function as usual.

Example

The definitions

```
RuleBaseListed("f",{a,b,c})
10 # f(_a,_b,{_c,_d}) <--
  Echo({"four args",a,b,c,d});
20 # f(_a,_b,c_IsList) <--
  Echo({"more than four args",a,b,c});
30 # f(_a,_b,_c) <-- Echo({"three args",a,b,c});
```

give the following interaction:

```
In> f(A)
Out> f(A);
In> f(A,B)
Out> f(A,B);
In> f(A,B,C)
three args A B C
Out> True;
In> f(A,B,C,D)
four args A B C D
Out> True;
In> f(A,B,C,D,E)
more than four args A B {C,D,E}
Out> True;
In> f(A,B,C,D,E,E)
more than four args A B {C,D,E,E}
Out> True;
```

The function {f} now appears to occupy all arities greater than 3:

```
In> RuleBase("f", {x,y,z,t});
CommandLine(1) : Rule base with this arity already defined
```

See also:

RuleBase(), *Retract()*, *Echo()*

Rule(*operator*, *arity*, *precedence*, *predicate*) body
define a rewrite rule

Parameters

- **"operator"** – string, name of function

- **arity** –
- **precedence** – integers
- **predicate** – function returning boolean
- **body** – expression, body of rule

Define a rule for the function "operator" with "arity", "precedence", "predicate" and "body". The "precedence" goes from low to high: rules with low precedence will be applied first.

The arity for a rules database equals the number of arguments. Different rules data bases can be built for functions with the same name but with a different number of arguments.

Rules with a low precedence value will be tried before rules with a high value, so a rule with precedence 0 will be tried before a rule with precedence 1.

HoldArg (*operator*, *parameters*)
mark argument as not evaluated

{"operator"} – string, name of a function {parameter} – atom, symbolic name of parameter

Specify that parameter should not be evaluated before used. This will be declared for all arities of "operator", at the moment this function is called, so it is best called after all {RuleBase} calls for this operator. "operator" can be a string or atom specifying the function name.

The {parameter} must be an atom from the list of symbolic arguments used when calling {RuleBase}.

See also:

RuleBase(), *HoldArgNr()*, *RuleBaseArgList()*

Retract (*function*, *arity*)
erase rules for a function

{"function"} – string, name of function {arity} – positive integer

Remove a rulebase for the function named {"function"} with the specific {arity}, if it exists at all. This will make Yacas forget all rules defined for a given function. Rules for functions with the same name but different arities are not affected.

Assignment {:=} of a function does this to the function being (re)defined.

See also:

RuleBaseArgList(), *RuleBase()*, *:=()*

UnFence (*operator*, *arity*)
change local variable scope for a function

{"operator"} – string, name of function {arity} – positive integers

When applied to a user function, the bodies defined for the rules for "operator" with given arity can see the local variables from the calling function. This is useful for defining macro-like procedures (looping and such).

The standard library functions {For} and {ForEach} use {UnFence}.

HoldArgNr (*function*, *arity*, *argNum*)
specify argument as not evaluated

{"function"} – string, function name {arity}, {argNum} – positive integers

Declares the argument numbered {argNum} of the function named {"function"} with specified {arity} to be unevaluated ("held"). Useful if you don't know symbolic names of parameters, for instance, when the function was not declared using an explicit {RuleBase} call. Otherwise you could use {HoldArg}.

See also:

HoldArg(), *RuleBase()*

RuleBaseArgList(*operator*, *arity*)
obtain list of arguments

{"operator"} – string, name of function {arity} – integer

Returns a list of atoms, symbolic parameters specified in the {RuleBase} call for the function named {"operator"} with the specific {arity}.

See also:

RuleBase(), *HoldArgNr()*, *HoldArg()*

MacroSet()
define rules in functions

MacroClear()
define rules in functions

MacroLocal()
define rules in functions

MacroRuleBase()
define rules in functions

MacroRuleBaseListed()
define rules in functions

MacroRule()
define rules in functions

These functions have the same effect as their non-macro counterparts, except that their arguments are evaluated before the required action is performed. This is useful in macro-like procedures or in functions that need to define new rules based on parameters.

Make sure that the arguments of {Macro}... commands evaluate to expressions that would normally be used in the non-macro versions!

See also:

Set(), *Clear()*, *Local()*, *RuleBase()*, *Rule()*, *Backquoting()*

Backquoting()
macro expansion (LISP-style backquoting)

{expression} – expression containing "{@var}" combinations to substitute the value of variable "{var}"

Backquoting is a macro substitution mechanism. A backquoted {expression} is evaluated in two stages: first, variables prefixed by {@} are evaluated inside an expression, and second, the new expression is evaluated.

To invoke this functionality, a backquote {`} needs to be placed in front of an expression. Parentheses around the expression are needed because the backquote binds tighter than other operators.

The expression should contain some variables (assigned atoms) with the special prefix operator {@}. Variables prefixed by {@} will be evaluated even if they are inside function arguments that are normally not evaluated (e.g. functions declared with {HoldArg}). If the {@var} pair is in place of a function name, e.g. "{@f(x)}", then at the first stage of evaluation the function name itself is replaced, not the return value of the function (see example); so at the second stage of evaluation, a new function may be called.

One way to view backquoting is to view it as a parametric expression generator. {@var} pairs get substituted with the value of the variable {var} even in contexts where nothing would be evaluated. This effect can be also achieved using {UnList} and {Hold} but the resulting code is much more difficult to read and maintain.

This operation is relatively slow since a new expression is built before it is evaluated, but nonetheless backquoting is a powerful mechanism that sometimes allows to greatly simplify code.

Example

This example defines a function that automatically evaluates to a number as soon as the argument is a number (a lot of functions do this only when inside a {N(...)} section).

```
In> Decl(f1,f2) := \
In>    `(@f1(x_IsNumber) <-- N(@f2(x)));
Out> True;
In> Decl(nSin,Sin)
Out> True;
In> Sin(1)
Out> Sin(1);
In> nSin(1)
Out> 0.8414709848;
```

This example assigns the expression {func(value)} to variable {var}. Normally the first argument of {Set} would be unevaluated.

```
In> SetF(var,func,value) := \
In>      `(Set(@var,@func(@value)));
Out> True;
In> SetF(a,Sin,x)
Out> True;
In> a
Out> Sin(x);
```

See also:

MacroSet(), *MacroLocal()*, *MacroRuleBase()*, *Hold()*, *HoldArg()*, *DefMacroRuleBase()*

DefMacroRuleBase (*name*, *params*)
define a function as a macro

{name} – string, name of a function {params} – list of arguments

{DefMacroRuleBase} is similar to {RuleBase}, with the difference that it declares a macro, instead of a function. After this call, rules can be defined for the function "{name}", but their interpretation will be different.

With the usual functions, the evaluation model is that of the applicative-order model of substitution, meaning that first the arguments are evaluated, and then the function is applied to the result of evaluating these arguments. The function is entered, and the code inside the function can not access local variables outside of its own local variables.

With macros, the evaluation model is that of the normal-order model of substitution, meaning that all occurrences of variables in an expression are first substituted into the body of the macro, and only then is the resulting expression evaluated in its calling environment. This is important, because then in principle a macro body can access the local variables from the calling environment, whereas functions can not do that.

As an example, suppose there is a function {square}, which squares its argument, and a function {add}, which adds its arguments. Suppose the definitions of these functions are:

```
add(x,y) <-- x+y;
```

and

```
square(x) <-- x*x;
```

In applicative-order mode (the usual way functions are evaluated), in the following expression

```
add(square(2),square(3))
```

first the arguments to {add} get evaluated. So, first {square(2)} is evaluated. To evaluate this, first {2} is evaluated, but this evaluates to itself. Then the {square} function is applied to it, {2*2}, which returns 4. The same is done for {square(3)}, resulting in {9}. Only then, after evaluating these two arguments, {add} is applied to them, which is equivalent to `add(4,9)` resulting in calling {4+9}, which in turn results in {13}.

In contrast, when {add} is a macro, the arguments to {add} are first expanded. So

```
add(square(2),square(3))
```

first expands to

```
square(2) + square(3)
```

and then this expression is evaluated, as if the user had written it directly. In other words, {square(2)} is not evaluated before the macro has been fully expanded.

Macros are useful for customizing syntax, and compilers can potentially greatly optimize macros, as they can be inlined in the calling environment, and optimized accordingly.

There are disadvantages, however. In interpreted mode, macros are slower, as the requirement for substitution means that a new expression to be evaluated has to be created on the fly. Also, when one of the parameters to the macro occur more than once in the body of the macro, it is evaluated multiple times.

When defining transformation rules for macros, the variables to be substituted need to be preceded by the {@} operator, similar to the back-quoting mechanism. Apart from that, the two are similar, and all transformation rules can also be applied to macros.

Macros can co-exist with functions with the same name but different arity. For instance, one can have a function {foo(a,b)} with two arguments, and a macro {foo(a,b,c)} with three arguments.

Example The following example defines a function {myfor}, and shows one use, referencing a variable {a} from the calling environment.

```
In> DefMacroRuleBase("myfor",{init,pred,inc,body})
Out> True;
In> myfor(_init,_pred,_inc,_body)<--[@init;While(@pred)[@body;@inc;];True;];
Out> True;
In> a:=10
Out> 10;
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:

RuleBase(), *Backquoting()*, *DefMacroRuleBaseListed()*

DefMacroRuleBaseListed(*name*, *params*)
define macro with variable number of arguments

{"name"} – string, name of function {params} – list of arguments to function

This does the same as {DefMacroRuleBase} (define a macro), but with a variable number of arguments, similar to {RuleBaseListed}.

See also:

RuleBase(), *RuleBaseListed()*, *Backquoting()*, *DefMacroRuleBase()*

ExtraInfo'Set(*expr*, *tag*)
ExtraInfo'Get(*expr*)
annotate objects with additional information

{expr} – any expression {tag} – tag information (any other expression)

Sometimes it is useful to be able to add extra tag information to "annotate" objects or to label them as having certain "properties". The functions {ExtraInfo'Set} and {ExtraInfo'Get} enable this.

The function {ExtraInfo'Set} returns the tagged expression, leaving the original expression alone. This means there is a common pitfall: be sure to assign the returned value to a variable, or the tagged expression is lost when the temporary object is destroyed.

The original expression is left unmodified, and the tagged expression returned, in order to keep the atomic objects small. To tag an object, a new type of object is created from the old object, with one added property (the tag). The tag can be any expression whatsoever.

The function {ExtraInfo'Get(x)} retrieves this tag expression from an object {x}. If an object has no tag, it looks the same as if it had a tag with value *False*.

No part of the Yacas core uses tags in a way that is visible to the outside world, so for specific purposes a programmer can devise a format to use for tag information. Association lists (hashes) are a natural fit for this, although it is not required and a tag can be any object (except the atom *False* because it is indistinguishable from having no tag information). Using association lists is highly advised since it is most likely to be the format used by other parts of the library, and one needs to avoid clashes with other library code. Typically, an object will either have no tag or a tag which is an associative list (perhaps empty). A script that uses tagged objects will check whether an object has a tag and if so, will add or modify certain entries of the association list, preserving any other tag information.

Note that {FlatCopy} currently does not copy the tag information (see examples).

Example

```
In> a:=2*b
Out> 2*b;
In> a:=ExtraInfo'Set(a,{{"type","integer"}})
Out> 2*b;
In> a
Out> 2*b;
In> ExtraInfo'Get(a)
Out> {{"type","integer"}};
In> ExtraInfo'Get(a)["type"]
Out> "integer";
In> c:=a
Out> 2*b;
In> ExtraInfo'Get(c)
Out> {{"type","integer"}};
In> c
Out> 2*b;
In> d:=FlatCopy(a);
Out> 2*b;
In> ExtraInfo'Get(d)
Out> False;
```

See also:

Assoc(), *:=()*

GarbageCollect()
do garbage collection on unused memory

{GarbageCollect} garbage-collects unused memory. The Yacas system uses a reference counting system for most objects, so this call is usually not necessary.

Reference counting refers to bookkeeping where in each object a counter is held, keeping track of the number of parts in the system using that object. When this count drops to zero, the object is automatically removed. Reference counting is not the fastest way of doing garbage collection, but it can be implemented in a very clean way with very little code.

Among the most important objects that are not reference counted are the strings. {GarbageCollect} collects these and disposes of them when they are not used any more.

{GarbageCollect} is useful when doing a lot of text processing, to clean up the text buffers. It is not highly needed, but it keeps memory use low.

FindFunction(*function*)
find the library file where a function is defined

{function} – string, the name of a function

This function is useful for quickly finding the file where a standard library function is defined. It is likely to only be useful for developers. The function {FindFunction} scans the {.def} files that were loaded at start-up. This means that functions that are not listed in {.def} files will not be found with {FindFunction}.

Example

```
In> FindFunction("Sum")
Out> "sums.rep/code.ys";
In> FindFunction("Integrate")
Out> "integrate.rep/code.ys";
```

See also:

Vi()

Secure(*body*)
guard the host OS

{body} – expression

{Secure} evaluates {body} in a "safe" environment, where files cannot be opened and system calls are not allowed. This can help protect the system when e.g. a script is sent over the Internet to be evaluated on a remote computer, which is potentially unsafe.

See also:

SystemCall()

3.21.3 Arbitrary-precision numerical programming

This chapter contains functions that help programming numerical calculations with arbitrary precision.

MultiplyNum(x, y[, ...])
optimized numerical multiplication

{x}, {y}, {z} – integer, rational or floating-point numbers to multiply

The function {MultiplyNum} is used to speed up multiplication of floating-point numbers with rational numbers. Suppose we need to compute $(p/q)*x$ where p, q are integers and x is a floating-point number. At high precision, it is faster to multiply x by an integer p and divide by an integer q than to compute p/q to high precision and then multiply by x. The function {MultiplyNum} performs this optimization.

The function accepts any number of arguments (not less than two) or a list of numbers. The result is always a floating-point number (even if {InNumericMode()} returns False).

See also:

MathMultiply()

CachedConstant (*cache*, *Cname*, *Cfunc*)
precompute multiple-precision constants

{cache} – atom, name of the cache {Cname} – atom, name of the constant {Cfunc} – expression that evaluates the constant

This function is used to create precomputed multiple-precision values of constants. Caching these values will save time if they are frequently used.

The call to {CachedConstant} defines a new function named {Cname()} that returns the value of the constant at given precision. If the precision is increased, the value will be recalculated as necessary, otherwise calling {Cname()} will take very little time.

The parameter {Cfunc} must be an expression that can be evaluated and returns the value of the desired constant at the current precision. (Most arbitrary-precision mathematical functions do this by default.)

The associative list {cache} contains elements of the form {{Cname, prec, value}}, as illustrated in the example. If this list does not exist, it will be created.

This mechanism is currently used by {N()} to precompute the values of Pi and $gamma$ (and the golden ratio through {GoldenRatio}, and {Catalan}). The name of the cache for {N()} is {CacheOfConstantsN}. The code in the function {N()} assigns unevaluated calls to {Internal'Pi()} and {Internal'gamma()} to the atoms {Pi} and {gamma} and declares them to be lazy global variables through {SetGlobalLazyVariable} (with equivalent functions assigned to other constants that are added to the list of cached constants).

The result is that the constants will be recalculated only when they are used in the expression under {N()}. In other words, the code in {N()} does the equivalent of

```
SetGlobalLazyVariable(mypi,Hold(Internal'Pi()));
SetGlobalLazyVariable(mygamma,Hold(Internal'gamma()));
```

After this, evaluating an expression such as {1/2+gamma} will call the function {Internal'gamma()} but not the function {Internal'Pi()}.

Example

```
In> CachedConstant( my'cache, Ln2, Internal'LnNum(2) )
Out> True;
In> Internal'Ln2()
Out> 0.6931471806;
In> V(N(Internal'Ln2(),20))
CachedConstant: Info: constant Ln2 is being recalculated at precision 20
Out> 0.69314718055994530942;
In> my'cache
Out> {{"Ln2",20,0.69314718055994530942}};
```

See also:

N(), *Builtin'Precision'Set()*, *Pi()*, *GoldenRatio()*, *Catalan()*, *gamma()*

NewtonNum (*func*, *x0*[, *prec0*[, *order*]])
low-level optimized Newton's iterations

{func} – a function specifying the iteration sequence {x0} – initial value (must be close enough to the root) {prec0} – initial precision (at least 4, default 5) {order} – convergence order (typically 2 or 3, default 2)

This function is an optimized interface for computing Newton's iteration sequences for numerical solution of equations in arbitrary precision.

{NewtonNum} will iterate the given function starting from the initial value, until the sequence converges within current precision. Initially, up to 5 iterations at the initial precision {prec0} is performed (the low precision is set for speed). The initial value {x0} must be close enough to the root so that the initial iterations converge. If the sequence does not produce even a single correct digit of the root after these initial iterations, an error message is printed. The default value of the initial precision is 5.

The {order} parameter should give the convergence order of the scheme. Normally, Newton iteration converges quadratically (so the default value is {order}=2) but some schemes converge faster and you can speed up this function by specifying the correct order. (Caution: if you give {order}=3 but the sequence is actually quadratic, the result will be silently incorrect. It is safe to use {order}=2.)

The verbose option {V} can be used to monitor the convergence. The achieved exact digits should roughly form a geometric progression.

Example

```
In> Builtin'Precision'Set(20)
Out> True;
In> NewtonNum({{x}, x+Sin(x)}, 3, 5, 3)
Out> 3.14159265358979323846;
```

See also:

Newton()

SumTaylorNum ()
optimized numerical evaluation of Taylor series

SumTaylorNum(x, NthTerm, order) SumTaylorNum(x, NthTerm, TermFactor, order) SumTaylorNum(x, ZerothTerm, TermFactor, order)

{NthTerm} – a function specifying n-th coefficient of the series {ZerothTerm} – value of the 0-th coefficient of the series {x} – number, value of the expansion variable {TermFactor} – a function specifying the ratio of n-th term to the previous one {order} – power of x in the last term

{SumTaylorNum} computes a Taylor series $Sum(k,0,n,a[k]*x^k)$ numerically. This function allows very efficient computations of functions given by Taylor series, although some tweaking of the parameters is required for good results.

The coefficients $a[k]$ of the Taylor series are given as functions of one integer variable (k). It is convenient to pass them to {SumTaylorNum} as closures. For example, if a function {a(k)} is defined, then

```
SumTaylorNum(x, {{k}, a(k)}, n)
```

computes the series $Sum(k, 0, n, a(k)*x^k)$.

Often a simple relation between successive coefficients $a[k-1]$, $a[k]$ of the series is available; usually they are related by a rational factor. In this case, the second form of {SumTaylorNum} should be used because it will compute the series faster. The function {TermFactor} applied to an integer $k>=1$ must return the ratio $a[k]$/$a[k-1]$. (If possible, the function {TermFactor} should return a rational number and not a floating-point number.) The function {NthTerm} may also be given, but the current implementation only calls {NthTerm(0)} and obtains all other coefficients by using {TermFactor}. Instead of the function {NthTerm}, a number giving the 0-th term can be given.

The algorithm is described elsewhere in the documentation. The number of terms {order}+1 must be specified and a sufficiently high precision must be preset in advance to achieve the desired accuracy. (The function {SumTaylorNum} does not change the current precision.)

Example

To compute 20 digits of $Exp(1)$ using the Taylor series, one needs 21 digits of working precision and 21 terms of the series.

```
In> Builtin'Precision'Set(21)
Out> True;
In> SumTaylorNum(1, {{k},1/k!}, 21)
Out> 2.718281828459045235351;
In> SumTaylorNum(1, 1, {{k},1/k}, 21)
Out> 2.71828182845904523535;
In> SumTaylorNum(1, {{k},1/k!}, {{k},1/k}, 21)
Out> 2.71828182845904523535;
In> RoundTo(N(Ln(%)),20)
Out> 1;
```

See also:

`Taylor()`

IntPowerNum(*x*, *n*, *mult*, *unity*)
optimized computation of integer powers

{x} – a number or an expression {n} – a non-negative integer (power to raise {x} to) {mult} – a function that performs one multiplication {unity} – value of the unity with respect to that multiplication

{IntPowerNum} computes the power x^n using the fast binary algorithm. It can compute integer powers with $n>=0$ in any ring where multiplication with unity is defined. The multiplication function and the unity element must be specified. The number of multiplications is no more than $2*Ln(n)/Ln(2)$.

Mathematically, this function is a generalization of {MathPower} to rings other than that of real numbers.

In the current implementation, the {unity} argument is only used when the given power {n} is zero.

Example

For efficient numerical calculations, the {MathMultiply} function can be passed:

```
In> IntPowerNum(3, 3, MathMultiply,1)
Out> 27;
```

Otherwise, the usual {*} operator suffices:

```
In> IntPowerNum(3+4*I, 3, *,1)
Out> Complex(-117,44);
In> IntPowerNum(HilbertMatrix(2), 4, *, Identity(2))
Out> {{289/144,29/27},{29/27,745/1296}};
```

Compute $Mod(3^100,7)$:

```
In> IntPowerNum(3,100,{{x,y},Mod(x*y,7)},1)
Out> 4;
```

See also:

MultiplyNum(), *MathPower()*, *MatrixPower()*

BinSplitNum(*n1*, *n2*, *a*, *b*, *c*, *d*)
computations of series by the binary splitting method

BinSplitData (*n1*, *n2*, *a*, *b*, *c*, *d*)
computations of series by the binary splitting method

BinSplitFinal (*{P*, *Q*, *B*, *T}*)
computations of series by the binary splitting method

{n1}, {n2} – integers, initial and final indices for summation {a}, {b}, {c}, {d} – functions of one argument, coefficients of the series {P}, {Q}, {B}, {T} – numbers, intermediate data as returned by {BinSplitData}

The binary splitting method is an efficient way to evaluate many series when fast multiplication is available and when the series contains only rational numbers. The function {BinSplitNum} evaluates a series of the form $$ S(n[1],n[2])=Sum(k,n[1],n[2], a(k)/b(k)*(p(0)/q(0)) * ... * p(k)/q(k)) $$. Most series for elementary and special functions at rational points are of this form when the functions $a(k)$, $b(k)$, $p(k)$, $q(k)$ are chosen appropriately.

The last four arguments of {BinSplitNum} are functions of one argument that give the coefficients $a(k)$, $b(k)$, $p(k)$, $q(k)$. In most cases these will be short integers that are simple to determine. The binary splitting method will work also for non-integer coefficients, but the calculation will take much longer in that case.

Note: the binary splitting method outperforms the straightforward summation only if the multiplication of integers is faster than quadratic in the number of digits. See <*the algorithm documentation*|*yacasdoc://Algo/3/14/*> for more information.

The two other functions are low-level functions that allow a finer control over the calculation. The use of the low-level routines allows checkpointing or parallelization of a binary splitting calculation.

The binary splitting method recursively reduces the calculation of $S(n[1],n[2])$ to the same calculation for the two halves of the interval [$n[1]$, $n[2]$]. The intermediate results of a binary splitting calculation are returned by {BinSplitData} and consist of four integers P, Q, B, T. These four integers are converted into the final answer S by the routine {BinSplitFinal} using the relation $$ S = T / (B*Q) $$.

Example

Compute the series for $e=Exp(1)$ using binary splitting. (We start from $n=1$ to simplify the coefficient functions.):

```
In> Builtin'Precision'Set(21)
Out> True;
In>  BinSplitNum(1,21, {{k},1}, {{k},1},{{k},1},{{k},k})
Out> 1.718281828459045235359;
In> N(Exp(1)-1)
Out> 1.718281828459045236;
In>  BinSplitData(1,21, {{k},1}, {{k},1},{{k},1},{{k},k})
Out> {1,51090942171709440000,1, 877886375325002400 22};
In> BinSplitFinal(%)
Out> 1.718281828459045235359;
```

See also:

SumTaylorNum()

MathSetExactBits (*x*)
manipulate precision of floating-point numbers

MathGetExactBits (*x*, *bits*)
manipulate precision of floating-point numbers

{x} – an expression evaluating to a floating-point number {bits} – integer, number of bits

Each floating-point number in Yacas has an internal precision counter that stores the number of exact bits in the mantissa. The number of exact bits is automatically updated after each arithmetic operation to reflect the gain

or loss of precision due to round-off. The functions {MathGetExactBits}, {MathSetExactBits} allow to query or set the precision flags of individual number objects.

{MathGetExactBits(x)} returns an integer number n such that {x} represents a real number in the interval [$x*(1-2^(-n))$, $x*(1+2^(-n))$] if $x!=0$ and in the interval [$-2^(-n)$, $2^(-n)$] if $x=0$. The integer n is always nonnegative unless {x} is zero (a "floating zero"). A floating zero can have a negative value of the number n of exact bits.

These functions are only meaningful for floating-point numbers. (All integers are always exact.) For integer {x}, the function {MathGetExactBits} returns the bit count of {x} and the function {MathSetExactBits} returns the unmodified integer {x}.

Todo

FIXME - these examples currently do not work because of bugs

Example

The default precision of 10 decimals corresponds to 33 bits:

```
In> MathGetExactBits(1000.123)
Out> 33;
In> x:=MathSetExactBits(10., 20)
Out> 10.;
In> MathGetExactBits(x)
Out> 20;
```

Prepare a "floating zero" representing an interval [-4, 4]:

```
In> x:=MathSetExactBits(0., -2)
Out> 0.;
In> x=0
Out> True;
```

See also:

Builtin'Precision'Set(), *Builtin'Precision'Get()*

InNumericMode()
determine if currently in numeric mode

NonN(*expr*)
calculate part in non-numeric mode

{expr} – expression to evaluate {prec} – integer, precision to use

When in numeric mode, {InNumericMode()} will return *True*, else it will return *False*. {Yacas} is in numeric mode when evaluating an expression with the function {N}. Thus when calling {N(expr)}, {InNumericMode()} will return *True* while {expr} is being evaluated.

{InNumericMode()} would typically be used to define a transformation rule that defines how to get a numeric approximation of some expression. One could define a transformation rule:

```
f(_x)_InNumericMode() <- [... some code to get a numeric approximation of f(x) ... ];
```

{InNumericMode()} usually returns *False*, so transformation rules that check for this predicate are usually left alone.

When in numeric mode, {NonN} can be called to switch back to non-numeric mode temporarily.

{NonN} is a macro. Its argument {expr} will only be evaluated after the numeric mode has been set appropriately.

Example

```
In> InNumericMode()
Out> False
In> N(InNumericMode())
Out> True
In> N(NonN(InNumericMode()))
Out> False
```

See also:

N(), *Builtin'Precision'Set()*, *Builtin'Precision'Get()*, *Pi()*, *CachedConstant()*

IntLog (*n*, *base*)
integer part of logarithm

{n}, {base} – positive integers

{IntLog} calculates the integer part of the logarithm of {n} in base {base}. The algorithm uses only integer math and may be faster than computing $$Ln(n)/Ln(base)$$ with multiple precision floating-point math and rounding off to get the integer part.

This function can also be used to quickly count the digits in a given number.

Example

Count the number of bits:

```
In> IntLog(257^8, 2)
Out> 64;
```

Count the number of decimal digits:

```
In> IntLog(321^321, 10)
Out> 804;
```

See also:

IntNthRoot(), *Div()*, *Mod()*, *Ln()*

IntNthRoot (*x*, *n*)
integer part of n-th root

{x}, {n} – positive integers

{IntNthRoot} calculates the integer part of the n-th root of x. The algorithm uses only integer math and may be faster than computing $x^(1/n)$ with floating-point and rounding.

This function is used to test numbers for prime powers.

Example

```
In> IntNthRoot(65537^111, 37)
Out> 281487861809153;
```

See also:

IntLog(), *MathPower()*, *IsPrimePower()*

NthRoot (*m*, *n*)
calculate/simplify nth root of an integer

{m} – a non-negative integer ($m>0$) {n} – a positive integer greater than 1 ($n>1$)

{NthRoot(m,n)} calculates the integer part of the n-th root $m^(1/n)$ and returns a list {{f,r}}. {f} and {r} are both positive integers that satisfy $f^n r = m$. In other words, f is the largest integer such that m divides f^n and r is the remaining factor.

For large {m} and small {n} {NthRoot} may work quite slowly. Every result {{f,r}} for given {m}, {n} is saved in a lookup table, thus subsequent calls to {NthRoot} with the same values {m}, {n} will be executed quite fast.

Example

```
In> NthRoot(12,2)
Out> {2,3};
In> NthRoot(81,3)
Out> {3,3};
In> NthRoot(3255552,2)
Out> {144,157};
In> NthRoot(3255552,3)
Out> {12,1884};
```

See also:

IntNthRoot(), *Factors()*, *MathPower()*

ContFracList (*frac*[, *depth*])
manipulate continued fractions

ContFracEval (*list*[, *rest*])
manipulate continued fractions

{frac} – a number to be expanded {depth} – desired number of terms {list} – a list of coefficients {rest} – expression to put at the end of the continued fraction

The function {ContFracList} computes terms of the continued fraction representation of a rational number {frac}. It returns a list of terms of length {depth}. If {depth} is not specified, it returns all terms.

The function {ContFracEval} converts a list of coefficients into a continued fraction expression. The optional parameter {rest} specifies the symbol to put at the end of the expansion. If it is not given, the result is the same as if {rest=0}.

Example

```
In> A:=ContFracList(33/7 + 0.000001)
Out> {4,1,2,1,1,20409,2,1,13,2,1,4,1,1,3,3,2};
In> ContFracEval(Take(A, 5))
Out> 33/7;
In> ContFracEval(Take(A,3), remainder)
Out> 1/(1/(remainder+2)+1)+4;
```

See also:

ContFrac(), *GuessRational()*

GuessRational (*x*[, *digits*])
find optimal rational approximations

NearRational (*x*[, *digits*])
find optimal rational approximations

BracketRational (*x*, *eps*)
find optimal rational approximations

{x} – a number to be approximated (must be already evaluated to floating-point) {digits} – desired number of decimal digits (integer) {eps} – desired precision

The functions {GuessRational(x)} and {NearRational(x)} attempt to find "optimal" rational approximations to a given value {x}. The approximations are "optimal" in the sense of having smallest numerators and denominators among all rational numbers close to {x}. This is done by computing a continued fraction representation of {x} and truncating it at a suitably chosen term. Both functions return a rational number which is an approximation of {x}.

Unlike the function {Rationalize()} which converts floating-point numbers to rationals without loss of precision, the functions {GuessRational()} and {NearRational()} are intended to find the best rational that is approximately equal to a given value.

The function {GuessRational()} is useful if you have obtained a floating-point representation of a rational number and you know approximately how many digits its exact representation should contain. This function takes an optional second parameter {digits} which limits the number of decimal digits in the denominator of the resulting rational number. If this parameter is not given, it defaults to half the current precision. This function truncates the continuous fraction expansion when it encounters an unusually large value (see example). This procedure does not always give the "correct" rational number; a rule of thumb is that the floating-point number should have at least as many digits as the combined number of digits in the numerator and the denominator of the correct rational number.

The function {NearRational(x)} is useful if one needs to approximate a given value, i.e. to find an "optimal" rational number that lies in a certain small interval around a certain value {x}. This function takes an optional second parameter {digits} which has slightly different meaning: it specifies the number of digits of precision of the approximation; in other words, the difference between {x} and the resulting rational number should be at most one digit of that precision. The parameter {digits} also defaults to half of the current precision.

The function {BracketRational(x,eps)} can be used to find approximations with a given relative precision from above and from below. This function returns a list of two rational numbers {{r1,r2}} such that $r1<x<r2$ and $Abs(r2-r1)<Abs(x*eps)$. The argument {x} must be already evaluated to enough precision so that this approximation can be meaningfully found. If the approximation with the desired precision cannot be found, the function returns an empty list.

Example

Start with a rational number and obtain a floating-point approximation:

```
In> x:=N(956/1013)
Out> 0.9437314906
In> Rationalize(x)
Out> 4718657453/5000000000;
In> V(GuessRational(x))
GuessRational: using 10 terms of the continued fraction
Out> 956/1013;
In> ContFracList(x)
Out> {0,1,16,1,3,2,1,1,1,1,508848,3,1,2,1,2,2};
```

The first 10 terms of this continued fraction correspond to the correct continued fraction for the original rational number:

```
In> NearRational(x)
Out> 218/231;
```

This function found a different rational number closeby because the precision was not high enough:

```
In> NearRational(x, 10)
Out> 956/1013;
```

Find an approximation to $Ln(10)$ good to 8 digits:

```
In> BracketRational(N(Ln(10)), 10^(-8))
Out> {12381/5377,41062/17833};
```

See also:

ContFrac(), *ContFracList()*, *Rationalize()*

TruncRadian(*r*)
remainder modulo $2 * Pi$

{r} – a number

{TruncRadian} calculates $Mod(r,2*Pi)$, returning a value between 0 and $2*Pi$. This function is used in the trigonometry functions, just before doing a numerical calculation using a Taylor series. It greatly speeds up the calculation if the value passed is a large number.

The library uses the formula $$TruncRadian(r) = r - Floor(r/(2*Pi))*2*Pi$$, where r and $2*Pi$ are calculated with twice the precision used in the environment to make sure there is no rounding error in the significant digits.

Example

```
In> 2*Internal'Pi()
Out> 6.283185307;
In> TruncRadian(6.28)
Out> 6.28;
In> TruncRadian(6.29)
Out> 0.0068146929;
```

See also:

Sin(), *Cos()*, *Tan()*

Builtin'Precision'Set(*n*)
set the precision

{n} – integer, new value of precision

This command sets the number of decimal digits to be used in calculations. All subsequent floating point operations will allow for at least {n} digits of mantissa.

This is not the number of digits after the decimal point. For example, {123.456} has 3 digits after the decimal point and 6 digits of mantissa. The number {123.456} is adequately computed by specifying {Builtin'Precision'Set(6)}.

The call {Builtin'Precision'Set(n)} will not guarantee that all results are precise to {n} digits.

When the precision is changed, all variables containing previously calculated values remain unchanged. The {Builtin'Precision'Set} function only makes all further calculations proceed with a different precision.

Also, when typing floating-point numbers, the current value of {Builtin'Precision'Set} is used to implicitly determine the number of precise digits in the number.

Example

```
In> Builtin'Precision'Set(10)
Out> True;
In> N(Sin(1))
Out> 0.8414709848;
In> Builtin'Precision'Set(20)
Out> True;
In> x:=N(Sin(1))
Out> 0.84147098480789650665;
```

The value {x} is not changed by a {Builtin'Precision'Set()} call:

```
In> [ Builtin'Precision'Set(10); x; ]
Out> 0.84147098480789650665;
```

The value {x} is rounded off to 10 digits after an arithmetic operation:

```
In> x+0.
Out> 0.8414709848;
```

In the above operation, {0.} was interpreted as a number which is precise to 10 digits (the user does not need to type {0.0000000000} for this to happen). So the result of {x+0.} is precise only to 10 digits.

See also:

Builtin'Precision'Get(), *N()*

Builtin'Precision'Get()
get the current precision

This command returns the current precision, as set by {Builtin'Precision'Set}.

Example

```
In> Builtin'Precision'Get();
Out> 10;
In> Builtin'Precision'Set(20);
Out> True;
In> Builtin'Precision'Get();
Out> 20;
```

See also:

Builtin'Precision'Set(), *N()*

3.21.4 Error reporting

This chapter contains commands useful for reporting errors to the user.

Check(*predicate*, *"error text"*)
report "hard" errors

TrapError(*expression*, *errorHandler*)
trap "hard" errors

GetCoreError()
get "hard" error string

{predicate} – expression returning *True* or *False* {"error text"} – string to print on error {expression} – expression to evaluate (causing potential error) {errorHandler} – expression to be called to handle error

If {predicate} does not evaluate to *True*, the current operation will be stopped, the string {"error text"} will be printed, and control will be returned immediately to the command line. This facility can be used to assure that some condition is satisfied during evaluation of expressions (guarding against critical internal errors).

A "soft" error reporting facility that does not stop the execution is provided by the function {Assert}.

Example In> [Check(1=0,"bad value"); Echo(OK);] In function "Check" : CommandLine(1) : "bad value"

Note that {OK} is not printed.

TrapError evaluates its argument {expression}, returning the result of evaluating {expression}. If an error occurs, {errorHandler} is evaluated, returning its return value in stead.

GetCoreError returns a string describing the core error. TrapError and GetCoreError can be used in combination to write a custom error handler.

See also:

Assert()

Assert (*pred*, *str*, *expr*)
Assert(pred, str) pred
Assert (*pred*)
signal "soft" custom error

Precedence: EVAL OpPrecedence("Assert")

{pred} – predicate to check {"str"} – string to classify the error {expr} – expression, error object

{Assert} is a global error reporting mechanism. It can be used to check for errors and report them. An error is considered to occur when the predicate {pred} evaluates to anything except *True*. In this case, the function returns *False* and an error object is created and posted to the global error tableau. Otherwise the function returns *True*.

Unlike the "hard" error function {Check}, the function {Assert} does not stop the execution of the program.

The error object consists of the string {"str"} and an arbitrary expression {expr}. The string should be used to classify the kind of error that has occurred, for example "domain" or "format". The error object can be any expression that might be useful for handling the error later; for example, a list of erroneous values and explanations. The association list of error objects is currently obtainable through the function {GetErrorTableau()}.

If the parameter {expr} is missing, {Assert} substitutes *True*. If both optional parameters {"str"} and {expr} are missing, {Assert} creates an error of class {"generic"}.

Errors can be handled by a custom error handler in the portion of the code that is able to handle a certain class of errors. The functions {IsError}, {GetError} and {ClearError} can be used.

Normally, all errors posted to the error tableau during evaluation of an expression should be eventually printed to the screen. This is the behavior of prettyprinters {DefaultPrint}, {Print}, {PrettyForm} and {TeXForm} (but not of the inline prettyprinter, which is enabled by default); they call {DumpErrors} after evaluating the expression.

Example

```
In> Assert("bad value", "must be zero") 1=0
Out> False;
In> Assert("bad value", "must be one") 1=1
Out> True;
In> IsError()
Out> True;
In> IsError("bad value")
Out> True;
In> IsError("bad file")
Out> False;
In> GetError("bad value");
Out> "must be zero";
In> DumpErrors()
Error: bad value: must be zero
Out> True;
```

No more errors left:

```
In> IsError()
Out> False;
In> DumpErrors()
Out> True;
```

See also:

IsError(), *DumpErrors()*, *Check()*, *GetError()*, *ClearError()*, *ClearErrors()*, *GetErrorTableau()*

DumpErrors()
simple error handlers

ClearErrors()
simple error handlers

{DumpErrors} is a simple error handler for the global error reporting mechanism. It prints all errors posted using {Assert} and clears the error tableau.

{ClearErrors} is a trivial error handler that does nothing except it clears the tableau.

See also:

Assert(), *IsError()*

IsError()
IsError(*str*)
check for custom error

{"str"} – string to classify the error

{IsError()} returns *True* if any custom errors have been reported using {Assert}. The second form takes a parameter {"str"} that designates the class of the error we are interested in. It returns *True* if any errors of the given class {"str"} have been reported.

See also:

GetError(), *ClearError()*, *Assert()*, *Check()*

GetError(*str*)
custom errors handlers

ClearError(*str*)
custom errors handlers

GetErrorTableau()
custom errors handlers

{"str"} – string to classify the error

These functions can be used to create a custom error handler.

{GetError} returns the error object if a custom error of class {"str"} has been reported using {Assert}, or *False* if no errors of this class have been reported.

{ClearError("str")} deletes the same error object that is returned by {GetError("str")}. It deletes at most one error object. It returns *True* if an object was found and deleted, and *False* otherwise.

{GetErrorTableau()} returns the entire association list of currently reported errors.

Example

```
In> x:=1
Out> 1;
In> Assert("bad value", {x,"must be zero"}) x=0
Out> False;
In> GetError("bad value")
Out> {1, "must be zero"};
In> ClearError("bad value");
Out> True;
```

```
In> IsError()
Out> False;
```

See also:

IsError(), *Assert()*, *Check()*, *ClearErrors()*

CurrentFile()
return current input file

CurrentLine()
return current line number on input

The functions {CurrentFile} and {CurrentLine} return a string with the file name of the current file and the current line of input respectively.

These functions are most useful in batch file calculations, where there is a need to determine at which line an error occurred. One can define a function:

```
tst() := Echo({CurrentFile(),CurrentLine()});
```

which can then be inserted into the input file at various places, to see how far the interpreter reaches before an error occurs.

See also:

Echo()

3.21.5 Built-in (core) functions

Yacas comes with a small core of built-in functions and a large library of user-defined functions. Some of these core functions are documented in this chapter.

It is important for a developer to know which functions are built-in and cannot be redefined or {Retract}-ed. Also, core functions may be somewhat faster to execute than functions defined in the script library. All core functions are listed in the file {corefunctions.h} in the {src/} subdirectory of the Yacas source tree. The declarations typically look like this:

```
SetCommand(LispSubtract, "MathSubtract");
```

Here {LispSubtract} is the Yacas internal name for the function and {MathSubtract} is the name visible to the Yacas language. Built-in bodied functions and infix operators are declared in the same file.

MathNot(*expression*)
built-in logical "not"

Returns "False" if "expression" evaluates to "True", and vice versa.

MathAnd()
built-in logical "and"

Lazy logical {And}: returns *True* if all args evaluate to *True*, and does this by looking at first, and then at the second argument, until one is *False*. If one of the arguments is *False*, {And} immediately returns *False* without evaluating the rest. This is faster, but also means that none of the arguments should cause side effects when they are evaluated.

MathOr()
built-in logical "or"

{MathOr} is the basic logical "or" function. Similarly to {And}, it is lazy-evaluated. {And(...)} and {Or(...)} do also exist, defined in the script library. You can redefine them as infix operators yourself, so you have the choice

of precedence. In the standard scripts they are in fact declared as infix operators, so you can write {expr1 And expr}.

BitAnd(*n*, *m*)
bitwise and operation

BitOr(*n*, *m*)
bitwise or operation

BitXor(*n*, *m*)
bitwise xor operation

These functions return bitwise "and", "or" and "xor" of two numbers.

Equals(*a*, *b*)
check equality

Compares evaluated {a} and {b} recursively (stepping into expressions). So "Equals(a,b)" returns "True" if the expressions would be printed exactly the same, and "False" otherwise.

GreaterThan(*a*, *b*)
comparison predicate

LessThan(*a*, *b*)
comparison predicate

{a}, {b} – numbers or strings

Comparing numbers or strings (lexicographically).

Example

```
In> LessThan(1,1)
Out> False;
In> LessThan("a","b")
Out> True;
```

MathExp()

MathLog()

MathPower()

MathSin()

MathCos()

MathTan()

MathArcSin()

MathArcCos()

MathArcTan()

MathSinh()

MathCosh()

MathTanh()

MathArcSinh()

MathArcCosh()

MathArcTanh()

MathGcd()

MathAdd()

MathSubtract()

MathMultiply()

MathDivide()

MathSqrt()

MathFloor()

MathCeil()

MathAbs()

MathMod()

MathDiv()

MathGcd(*n*, *m*)
Greatest Common Divisor

MathAdd(*x*, *y*)
(add two numbers)

MathSubtract(*x*, *y*)
(subtract two numbers)

MathMultiply(*x*, *y*)
(multiply two numbers)

MathDivide(*x*, *y*)
(divide two numbers)

MathSqrt(*x*)
(square root, must be x>=0)

MathFloor(*x*)
(largest integer not larger than x)

MathCeil(*x*)
(smallest integer not smaller than x)

MathAbs(*x*)
(absolute value of x, or |x|)

MathExp(*x*)
(exponential, base 2.718...)

MathLog(*x*)
(natural logarithm, for x>0)

MathPower(*x*, *y*)
(power, x ^ y)

MathSin(*x*)
(sine)

MathCos(*x*)
(cosine)

MathTan(*x*)
(tangent)

MathSinh (*x*)
(hyperbolic sine)

MathCosh (*x*)
(hyperbolic cosine)

MathTanh (*x*)
(hyperbolic tangent)

MathArcSin (*x*)
(inverse sine)

MathArcCos (*x*)
(inverse cosine)

MathArcTan (*x*)
(inverse tangent)

MathArcSinh (*x*)
(inverse hyperbolic sine)

MathArcCosh (*x*)
(inverse hyperbolic cosine)

MathArcTanh (*x*)
(inverse hyperbolic tangent)

MathDiv (*x*, *y*)
(integer division, result is an integer)

MathMod (*x*, *y*)
(remainder of division, or x mod y)

These commands perform the calculation of elementary mathematical functions. The arguments must be numbers. The reason for the prefix {Math} is that the library needs to define equivalent non-numerical functions for symbolic computations, such as {Exp}, {Sin} and so on.

Note that all functions, such as the {MathPower}, {MathSqrt}, {MathAdd} etc., accept integers as well as floating-point numbers. The resulting values may be integers or floats. If the mathematical result is an exact integer, then the integer is returned. For example, {MathSqrt(25)} returns the integer {5}, and {MathPower(2,3)} returns the integer {8}. In such cases, the integer result is returned even if the calculation requires more digits than set by {Builtin'Precision'Set}. However, when the result is mathematically not an integer, the functions return a floating-point result which is correct only to the current precision.

Example

```
In> Builtin'Precision'Set(10)
Out> True
In> Sqrt(10)
Out> Sqrt(10)
In> MathSqrt(10)
Out> 3.16227766
In> MathSqrt(490000*2^150)
Out> 264452523040700131966976000
In> MathSqrt(490000*2^150+1)
Out> 0.264452523e26
In> MathPower(2,3)
Out> 8
In> MathPower(2,-3)
Out> 0.125
```

FastLog(*x*)
(natural logarithm),

FastPower(*x*, *y*)

FastArcSin(*x*)
double-precision math functions

Versions of these functions using the C++ library. These should then at least be faster than the arbitrary precision versions.

ShiftLeft(*expr*, *bits*)
built-in bitwise shift left operation

ShiftRight(*expr*, *bits*)
built-in bitwise shift right operation

ShiftLeft(expr,bits) ShiftRight(expr,bits)

Shift bits to the left or to the right.

IsPromptShown()
test for the Yacas prompt option

Returns *False* if Yacas has been started with the option to suppress the prompt, and *True* otherwise.

GetTime(*expr*)
measure the time taken by an evaluation

{expr} – any expression

The function {GetTime(expr)} evaluates the expression {expr} and returns the time needed for the evaluation. The result is returned as a floating-point number of seconds. The value of the expression {expr} is lost.

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 {GetTime}.

Example

```
In> GetTime(Simplify((a*b)/(b*a)))
Out> 0.09;
```

See also:

Time()

3.21.6 Generic objects

Generic objects are objects that are implemented in C++, but can be accessed through the Yacas interpreter.

IsGeneric(*object*)
check for generic object

Returns *True* if an object is of a generic object type.

GenericTypeName(*object*)
get type name

Returns a string representation of the name of a generic object.

Example

```
In> GenericTypeName(Array'Create(10,1))
Out> "Array";
```

Array'Create (*size*, *init*)
create array

Parameters

- **size** – size of the array
- **init** – initial value

Creates an array with `size` elements, all initialized to the value `init`.

Array'Size (*array*)
array size

Parameters **array** – an array

Returns array size (number of elements in the array)

Array'Get (*array*, *index*)
fetch array element

Parameters

- **array** – an array
- **index** – an index

Returns the element of `array` at position `index`

Note: Array indices are one-based, which means that the first element is indexed by 1.

Arrays can also be accessed through the `[]` operators. So `array[index]` would return the same as `Array'Get(array, index)`.

Array'Set (*array*, *index*, *element*)
set array element

Sets the element at position index in the array passed to the value passed in as argument to element. Arrays are treated as base-one, so {index} set to 1 would set first element.

Arrays can also be accessed through the {[]} operators. So {array[index] := element} would do the same as {Array'Set(array, index,element)}.

Array'CreateFromList (*list*)
convert list to array

Creates an array from the contents of the list passed in.

Array'ToList (*array*)
convert array to list

Creates a list from the contents of the array passed in.

3.21.7 The Yacas test suite

This chapter describes commands used for verifying correct performance of Yacas.

Yacas comes with a test suite which can be found in the directory {tests/}. Typing

make test

on the command line after Yacas was built will run the test. This test can be run even before {make install}, as it only uses files in the local directory of the Yacas source tree. The default extension for test scripts is {.yts} (Yacas test script).

The verification commands described in this chapter only display the expressions that do not evaluate correctly. Errors do not terminate the execution of the Yacas script that uses these testing commands, since they are meant to be used in test scripts.

Verify (*question*, *answer*)
verifying equivalence of two expressions

TestYacas (*question*, *answer*)
verifying equivalence of two expressions

LogicVerify (*question*, *answer*)
verifying equivalence of two expressions

LogicTest (*variables*, *expr1*, *expr2*)
verifying equivalence of two expressions

{question} – expression to check for {answer} – expected result after evaluation {variables} – list of variables {exprN} – Some boolean expression

The commands {Verify}, {TestYacas}, {LogicVerify} and {LogicTest} can be used to verify that an expression is <I>equivalent</I> to a correct answer after evaluation. All three commands return *True* or *False*.

For some calculations, the demand that two expressions are <I>identical</I> syntactically is too stringent. The Yacas system might change at various places in the future, but $ 1+x $ would still be equivalent, from a mathematical point of view, to $ x+1 $.

The general problem of deciding that two expressions $ a $ and $ b $ are equivalent, which is the same as saying that $ a-b=0 $, is generally hard to decide on. The following commands solve this problem by having domain-specific comparisons.

The comparison commands do the following comparison types:

- {Verify} – verify for literal equality. This is the fastest and simplest comparison, and can be used, for example, to test that an expression evaluates to $ 2 $.
- {TestYacas} – compare two expressions after simplification as multivariate polynomials. If the two arguments are equivalent multivariate polynomials, this test succeeds. {TestYacas} uses {Simplify}. Note: {TestYacas} currently should not be used to test equality of lists.
- {LogicVerify} – Perform a test by using {CanProve} to verify that from {question} the expression {answer} follows. This test command is used for testing the logic theorem prover in Yacas.
- {LogicTest} – Generate a truth table for the two expressions and compare these two tables. They should be the same if the two expressions are logically the same.

Example

```
In> Verify(1+2,3)
Out> True;
In> Verify(x*(1+x),x^2+x)
******************
x*(x+1) evaluates to x*(x+1) which differs
  from x^2+x
******************
Out> False;
In> TestYacas(x*(1+x),x^2+x)
Out> True;
```

```
In> Verify(a And c Or b And Not c,a Or b)
******************
 a And c Or b And Not c evaluates to  a And c
  Or b And Not c which differs from  a Or b
******************
Out> False;
In> LogicVerify(a And c Or b And Not c,a Or b)
Out> True;
In> LogicVerify(a And c Or b And Not c,b Or a)
Out> True;
In> LogicTest({A,B,C},Not((Not A) And (Not B)),A Or B)
Out> True
In> LogicTest({A,B,C},Not((Not A) And (Not B)),A Or C)
******************
CommandLine: 1

$TrueFalse4({A,B,C},Not(Not A And Not B))
 evaluates to
{{{False,False},{True,True}},{{True,True},{True,True}}}
 which differs from
{{{False,True},{False,True}},{{True,True},{True,True}}}
******************
Out> False
```

See also:

Simplify(), *CanProve()*, *KnownFailure()*

KnownFailure (*test*)

Mark a test as a known failure

{test} – expression that should return *False* on failure

The command {KnownFailure} marks a test as known to fail by displaying a message to that effect on screen.

This might be used by developers when they have no time to fix the defect, but do not wish to alarm users who download Yacas and type {make test}.

Example

```
In> KnownFailure(Verify(1,2))
Known failure:
******************
1 evaluates to  1 which differs from  2
******************
Out> False;
In> KnownFailure(Verify(1,1))
Known failure:
Failure resolved!
Out> True;
```

See also:

Verify(), *TestYacas()*, *LogicVerify()*

RoundTo (*number*, *precision*)

Round a real-valued result to a set number of digits

{number} – number to round off {precision} – precision to use for round-off

The function {RoundTo} rounds a floating point number to a specified precision, allowing for testing for correctness using the {Verify} command.

Example

```
In> N(RoundTo(Exp(1),30),30)
Out> 2.718281821102301149519597865 52;
In> N(RoundTo(Exp(1),20),20)
Out> 2.71828182796964237096;
```

See also:

Verify(), *VerifyArithmetic()*, *VerifyDiv()*

VerifyArithmetic(*x*, *n*, *m*)
Special purpose arithmetic verifiers

RandVerifyArithmetic(*n*)
Special purpose arithmetic verifiers

VerifyDiv(*u*, *v*)
Special purpose arithmetic verifiers

{x}, {n}, {m}, {u}, {v} – integer arguments

The commands {VerifyArithmetic} and {VerifyDiv} test a mathematic equality which should hold, testing that the result returned by the system is mathematically correct according to a mathematically provable theorem.

{VerifyArithmetic} verifies for an arbitrary set of numbers $ x $, $ n $ and $ m $ that $$ (x^n-1)*(x^m-1) = x^(n+m)-(x^n)-(x^m)+1 $$.

The left and right side represent two ways to arrive at the same result, and so an arithmetic module actually doing the calculation does the calculation in two different ways. The results should be exactly equal.

{RandVerifyArithmetic(n)} calls {VerifyArithmetic} with random values, {n} times.

{VerifyDiv(u,v)} checks that $$ u = v*Div(u,v) + Mod(u,v) $$.

Example

```
In> VerifyArithmetic(100,50,60)
Out> True;
In> RandVerifyArithmetic(4)
Out> True;
In> VerifyDiv(x^2+2*x+3,x+1)
Out> True;
In> VerifyDiv(3,2)
Out> True;
```

See also:

Verify()

CHAPTER 4

Programming in Yacas

This part of the manual is a somewhat in-depth explanation of the Yacas programming language and environment. It assumes that you have worked through the introductory tutorial. You should consult the function reference about how to use the various Yacas functions mentioned here.

This document should get you started programming in Yacas. There are some basic explanations and hands-on tutorials.

4.1 The Yacas architecture

Yacas is designed as a small core engine that interprets a library of scripts. The core engine provides the syntax parser and a number of hard-wired functions, such as {Set()} or {MathExp()} which cannot be redefined by the user. The script library resides in the scripts directory "{scripts/}" and contains higher-level definitions of functions and constants. The library scripts are on equal footing with any code the user executes interactively or any files the user loads.

Generally, all core functions have plain names and almost all are not "bodied" or infix operators. The file {corefunctions.h} in the source tree lists declarations of all kernel functions callable from Yacas; consult it for reference. For many of the core functions, the script library already provides convenient aliases. For instance, the addition operator "{+}" is defined in the script {scripts/standard} while the actual addition of numbers is performed through the built-in function {MathAdd}.

4.1.1 Startup, scripts and {.def} files

When Yacas is first started or restarted, it executes the script {yacasinit.ys} in the scripts directory. This script may load some other scripts. In order to start up quickly, Yacas does not execute all other library scripts at first run or at restart. It only executes the file {yacasinit.ys} and all {.def} files in the scripts. The {.def} files tell the system where it can find definitions for various library functions. Library is divided into "packages" stored in "repository" directories. For example, the function {ArcTan} is defined in the {stdfuncs} package; the library file is {stdfuncs.rep/}{code.ys} and the {.def} file is {stdfuncs.rep}{/code.ys.def}. The function {ArcTan} mentioned in the {.def} file, therefore Yacas will know to load the package {stdfuncs} when the user invokes {ArcTan}. This way Yacas knows where to look for any given function without actually loading the file where the function is defined.

There is one exception to the strategy of delayed loading of the library scripts. Namely, the syntax definitions of infix, prefix, postfix and bodied functions, such as {Infix("*",4)} cannot be delayed (it is currently in the file {stdopers.ys}). If it were delayed, the Yacas parser would encounter {1+2} (typed by the user) and generate a syntax error before it has a chance to load the definition of the operator "{+}".

4.1.2 Object types

Yacas supports two basic kinds of objects: atoms and compounds. Atoms are (integer or real, arbitrary-precision) numbers such as {2.71828}, symbolic variables such as {A3} and character strings. Compounds include functions and expressions, e.g. {Cos(a-b)} and lists, e.g. {{1+a,2+b,3+c}}.

The type of an object is returned by the built-in function {Type}, for example:

```
In> Type(a);
Out> "";
In> Type(F(x));
Out> "F";
In> Type(x+y);
Out> "+";
In> Type({1,2,3});
Out> "List";
```

Internally, atoms are stored as strings and compounds as lists. (The Yacas lexical analyzer is case-sensitive, so {List} and {list} are different atoms.) The functions {String()} and {Atom()} convert between atoms and strings. A Yacas list {{1,2,3}} is internally a list {(List 1 2 3)} which is the same as a function call {List(1,2,3)} and for this reason the "type" of a list is the string {"List"}. During evaluation, atoms can be interpreted as numbers, or as variables that may be bound to some value, while compounds are interpreted as function calls.

Note that atoms that result from an {Atom()} call may be invalid and never evaluate to anything. For example, {Atom(3X)} is an atom with string representation "3X" but with no other properties.

Currently, no other lowest-level objects are provided by the core engine besides numbers, atoms, strings, and lists. There is, however, a possibility to link some externally compiled code that will provide additional types of objects. Those will be available in Yacas as "generic objects." For example, fixed-size arrays are implemented in this way.

4.2 Yacas evaluation scheme

Evaluation of an object is performed either explicitly by the built-in command {Eval()} or implicitly when assigning variables or calling functions with the object as argument (except when a function does not evaluate that argument). Evaluation of an object can be explicitly inhibited using {Hold()}. To make a function not evaluate one of its arguments, a {HoldArg(funcname, argname)} must be declared for that function.

Internally, all expressions are either atoms or lists (perhaps nested). Use {FullForm()} to see the internal form of an expression. A Yacas list expression written as {{a, b}} is represented internally as {(List a b)}, equivalently to a function call {List(a,b)}.

Evaluation of an atom goes as follows: if the atom is bound locally as a variable, the object it is bound to is returned, otherwise, if it is bound as a global variable then that is returned. Otherwise, the atom is returned unevaluated. Note that if an atom is bound to an expression, that expression is considered as final and is not evaluated again.

Internal lists of atoms are generally interpreted in the following way: the first atom of the list is some command, and the atoms following in the list are considered the arguments. The engine first tries to find out if it is a built-in command (core function). In that case, the function is executed. Otherwise, it could be a user-defined function (with a "rule database"), and in that case the rules from the database are applied to it. If none of the rules are applicable, or if no rules are defined for it, the object is returned unevaluated.

Application of a rule to an expression transforms it into a different expression to which other rules may be applicable. Transformation by matching rules continues until no more rules are applicable, or until a "terminating" rule is encountered. A "terminating" rule is one that returns {Hold()} or {UnList()} of some expression. Calling these functions gives an unevaluated expression because it terminates the process of evaluation itself.

The main properties of this scheme are the following. When objects are assigned to variables, they generally are evaluated (except if you are using the {Hold()} function) because assignment {var := value} is really a function call

to {Set(var, value)} and this function evaluates its second argument (but not its first argument). When referencing that variable again, the object which is its value will not be re-evaluated. Also, the default behavior of the engine is to return the original expression if it could not be evaluated. This is a desired behavior if evaluation is used for simplifying expressions.

One major design flaw in Yacas (one that other functional languages like LISP also have) is that when some expression is re-evaluated in another environment, the local variables contained in the expression to be evaluated might have a different meaning. In this case it might be useful to use the functions {LocalSymbols} and {TemplateFunction}. Calling

```
LocalSymbols(a,b)
a*b;
```

results in "{a}" and "{b}" in the multiplication being substituted with unique symbols that can not clash with other variables that may be used elsewhere. Use {TemplateFunction} instead of {Function} to define a function whose parameters should be treated as unique symbols.

Consider the following example:

```
In> f1(x):=Apply("+",{x,x});
Out> True
```

The function {f1} simply adds its argument to itself. Now calling this function with some argument:

```
In> f1(Sin(a))
Out> 2*Sin(a)
```

yields the expected result. However, if we pass as an argument an expression containing the variable {x}, things go wrong:

```
In> f1(Sin(x))
Out> 2*Sin(Sin(x))
```

This happens because within the function, {x} is bound to {Sin(x)}, and since it is passed as an argument to {Apply} it will be re-evaluated, resulting in {Sin(Sin(x))}. {TemplateFunction} solves this by making sure the arguments can not collide like this (by using {LocalSymbols}:

```
In> TemplateFunction("f2",{x}) Apply("+",{x,x});
Out> True
In> f2(Sin(a))
Out> 2*Sin(a)
In> f2(Sin(x))
Out> 2*Sin(x)
```

In general one has to be careful when functions like {Apply}, {Map} or {Eval} (or derivatives) are used.

4.3 Rules

Rules are special properties of functions that are applied when the function object is being evaluated. A function object could have just one rule bound to it; this is similar to a "subroutine" having a "function body" in usual procedural languages. However, Yacas function objects can also have several rules bound to them. This is analogous of having several alternative "function bodies" that are executed under different circumstances. This design is more suitable for symbolic manipulations.

A function is identified by its name as returned by {Type} and the number of arguments, or "arity". The same name can be used with different arities to define different functions: {f(x)} is said to "have arity 1" and {f(x,y)} has arity 2. Each of these functions may possess its own set of specific rules, which we shall call a "rule database" of a function.

Each function should be first declared with the built-in command {RuleBase} as follows:

```
RuleBase("FunctionName",{argument list});
```

So, a new (and empty) rule database for {f(x,y)} could be created by typing {RuleBase("f",{x,y})}. The names for the arguments "x" and "y" here are arbitrary, but they will be globally stored and must be later used in descriptions of particular rules for the function {f}. After the new rulebase declaration, the evaluation engine of Yacas will begin to really recognize {f} as a function, even though no function body or equivalently no rules have been defined for it yet.

The shorthand operator {:=} for creating user functions that we illustrated in the tutorial is actually defined in the scripts and it makes the requisite call to the {RuleBase()} function. After a {RuleBase()} call you can specify parsing properties for the function; for example, you could make it an infix or bodied operator.

Now we can add some rules to the rule database for a function. A rule simply states that if a specific function object with a specific arity is encountered in an expression and if a certain predicate is true, then Yacas should replace this function with some other expression. To tell Yacas about a new rule you can use the built-in {Rule} command. This command is what does the real work for the somewhat more aesthetically pleasing {... # ... <– ...} construct we have seen in the tutorial. You do not have to call {RuleBase()} explicitly if you use that construct.

Here is the general syntax for a {Rule()} call:

```
Rule("foo", arity, precedence, pred) body;
```

This specifies that for function {foo} with given {arity} ({foo(a,b)} has arity 2), there is a rule that if {pred} is true, then {body} should be evaluated, and the original expression replaced by the result. Predicate and body can use the symbolic names of arguments that were declared in the {RuleBase} call.

All rules for a given function can be erased with a call to {Retract(funcname, arity)}. This is useful, for instance, when too many rules have been entered in the interactive mode. This call undefines the function and also invalidates the {RuleBase} declaration.

You can specify that function arguments are not evaluated before they are bound to the parameter: {HoldArg("foo",a)} would then declare that the a arguments in both {foo(a)} and {foo(a,b)} should not be evaluated before bound to {a}. Here the argument name {a} should be the same as that used in the {RuleBase()} call when declaring these functions. Inhibiting evaluation of certain arguments is useful for procedures performing actions based partly on a variable in the expression, such as integration, differentiation, looping, etc., and will be typically used for functions that are algorithmic and procedural by nature.

Rule-based programming normally makes heavy use of recursion and it is important to control the order in which replacement rules are to be applied. For this purpose, each rule is given a precedence. Precedences go from low to high, so all rules with precedence 0 will be tried before any rule with precedence 1.

You can assign several rules to one and the same function, as long as some of the predicates differ. If none of the predicates are true, the function is returned with its arguments evaluated.

This scheme is slightly slower for ordinary functions that just have one rule (with the predicate *True*), but it is a desired behavior for symbolic manipulation. You can gradually build up your own functions, incrementally testing their properties.

4.3.1 Examples of using rules

As a simple illustration, here are the actual {RuleBase()} and {Rule()} calls needed to define the factorial function:

```
In> RuleBase("f",{n});
Out> True;
In> Rule("f", 1, 10, n=0) 1;
Out> True;
In> Rule("f", 1, 20, IsInteger(n) And n>0) n*f(n-1);
Out> True;
```

This definition is entirely equivalent to the one in the tutorial. {f(4)} should now return 24, while {f(a)} should return just {f(a)} if {a} is not bound to any value.

The {Rule} commands in this example specified two rules for function {f} with arity 1: one rule with precedence 10 and predicate {n=0}, and another with precedence 20 and the predicate that returns *True* only if {n} is a positive integer. Rules with lowest precedence get evaluated first, so the rule with precedence 10 will be tried before the rule with precedence 20. Note that the predicates and the body use the name "n" declared by the {RuleBase()} call.

After declaring {RuleBase()} for a function, you could tell the parser to treat this function as a postfix operator:

```
In> Postfix("f");
Out> True;
In> 4 f;
Out> 24;
```

There is already a function {Function} defined in the standard scripts that allows you to construct simple functions. An example would be

```
Function ("FirstOf", {list})  list[1] ;
```

which simply returns the first element of a list. This could also have been written as

```
Function("FirstOf", {list})
[
    list[1] ;
];
```

As mentioned before, the brackets {[ ]} are also used to combine multiple operations to be performed one after the other. The result of the last performed action is returned.

Finally, the function {FirstOf} could also have been defined by typing

```
FirstOf(list):=list[1] ;
```

4.4 Structured programming and control flow

Some functions useful for control flow are already defined in Yacas's standard library. Let's look at a possible definition of a looping function {ForEach}. We shall here consider a somewhat simple-minded definition, while the actual {ForEach} as defined in the standard script "controlflow" is a little more sophisticated.

```
Function("ForEach",{foreachitem,
  foreachlist,foreachbody})
[
   Local(foreachi,foreachlen);
   foreachlen:=Length(foreachlist);
   foreachi:=0;
   While (foreachi < foreachlen)
   [
     foreachi++;
     MacroLocal(foreachitem);
     MacroSet(foreachitem,
       foreachlist[foreachi]);
     Eval(foreachbody);
   ];
];

Bodied("ForEach");
UnFence("ForEach",3);
```

```
HoldArg("ForEach",foreachitem);
HoldArg("ForEach",foreachbody);
```

Functions like this should probably be defined in a separate file. You can load such a file with the command {Load("file")}. This is an example of a macro-like function. Let's first look at the last few lines. There is a {Bodied(...)} call, which states that the syntax for the function {ForEach()} is {ForEach(item,{list}) body;} – that is, the last argument to the command {ForEach} should be outside its brackets. {UnFence(...)} states that this function can use the local variables of the calling function. This is necessary, since the body to be evaluated for each item will probably use some local variables from that surrounding.

Finally, {HoldArg("function",argument)} specifies that the argument "{argument}" should not be evaluated before being bound to that variable. This holds for {foreachitem} and {foreachbody}, since {foreachitem} specifies a variable to be set to that value, and {foreachbody} is the expression that should be evaluated after that variable is set.

Inside the body of the function definition there are calls to {Local(...)}. {Local()} declares some local variable that will only be visible within a block {[...]}. The command {MacroLocal()} works almost the same. The difference is that it evaluates its arguments before performing the action on it. This is needed in this case, because the variable {foreachitem} is bound to a variable to be used as the loop iterator, and it is the variable it is bound to that we want to make local, not {foreachitem} itself. {MacroSet()} works similarly: it does the same as {Set()} except that it also first evaluates the first argument, thus setting the variable requested by the user of this function. The {Macro}... functions in the built-in functions generally perform the same action as their non-macro versions, apart from evaluating an argument it would otherwise not evaluate.

To see the function in action, you could type:

```
ForEach(i,{1,2,3}) [Write(i); NewLine();];
```

This should print 1, 2 and 3, each on a new line.

Note: the variable names "foreach..." have been chosen so they won't get confused with normal variables you use. This is a major design flaw in this language. Suppose there was a local variable {foreachitem}, defined in the calling function, and used in {foreachbody}. These two would collide, and the interpreter would use only the last defined version. In general, when writing a function that calls {Eval()}, it is a good idea to use variable names that can not collide with user's variables. This is generally the single largest cause of bugs when writing programs in Yacas. This issue should be addressed in the future.

4.5 Additional syntactic sugar

The parser is extended slightly to allow for fancier constructs.

- Lists, e.g. {{a,b}}. This then is parsed into the internal notation {(List a b)} , but will be printed again as {{a,b};}
- Statement blocks such as {[} statement1 {;} statement2{;];}. This is parsed into a Lisp object {(Prog} {(}statement1 {)} {(}statement2 {))}, and printed out again in the proper form.
- Object argument accessors in the form of {expr[index]}. These are mapped internally to {Nth(expr,index)}. The value of {index}=0 returns the operator of the object, {index}=1 the first argument, etc. So, if {expr} is {foo(bar)}, then {expr[0]} returns {foo}, and {expr[1]} returns {bar}. Since lists of the form {{...}} are essentially the same as {List(...)}, the same accessors can be used on lists.
- Function blocks such as

```
  While (i < 10)
    [
      Write(i);
```

```
        i:=i+1;
      ];
```

The expression directly following the {While(...)} block is added as a last argument to the {While(...)} call. So {While(a)b;} is parsed to the internal form {(While a b).}

This scheme allows coding the algorithms in an almost C-like syntax.

Strings are generally represented with quotes around them, e.g. "this is a string". Backslash {} in a string will unconditionally add the next character to the string, so a quote can be added with {"} (a backslash-quote sequence).

4.6 Using "Macro rules" (e.g. {NFunction})

The Yacas language allows to have rules whose definitions are generated at runtime. In other words, it is possible to write rules (or "functions") that, as a side-effect, will define other rules, and those other rules will depend on some parts of the expression the original function was applied to.

This is accomplished using functions {MacroRuleBase}, {MacroRule}, {MacroRulePattern}. These functions evaluate their arguments (including the rule name, predicate and body) and define the rule that results from this evaluation.

Normal, "non-Macro" calls such as {Rule()} will not evaluate their arguments and this is a desired feature. For example, suppose we defined a new predicate like this,

```
RuleBase("IsIntegerOrString, {x});
Rule("IsIntegerOrString", 1, 1, True)
    IsInteger(x) And IsString(x);
```

If the {Rule()} call were to evaluate its arguments, then the "body" argument, {IsInteger(x) And IsString(x)}, would be evaluated to *False* since {x} is an atom, so we would have defined the predicate to be always *False*, which is not at all what we meant to do. For this reason, the {Rule} calls do not evaluate their arguments.

Consider however the following situation. Suppose we have a function {f(arglist)} where {arglist} is its list of arguments, and suppose we want to define a function {Nf(arglist)} with the same arguments which will evaluate {f(arglist)} and return only when all arguments from {arglist} are numbers, and return unevaluated {Nf(arglist)} otherwise. This can of course be done by a usual rule such as

```
Rule("Nf", 3, 0, IsNumericList({x,y,z}))
  <-- "f" @ {x,y,z};
```

Here {IsNumericList} is a predicate that checks whether all elements of a given list are numbers. (We deliberately used a {Rule} call instead of an easier-to-read {<–} operator to make it easier to compare with what follows.)

However, this will have to be done for every function {f} separately. We would like to define a procedure that will define {Nf}, given any function {f}. We would like to use it like this:

```
NFunction("Nf", "f", {x,y,z});
```

After this function call we expect to be able to use the function {Nf}.

Here is how we could naively try to implement {NFunction} (and fail):

```
NFunction(new'name, old'name, arg'list) := [
  MacroRuleBase(new'name, arg'list);
  MacroRule(new'name, Length(arg'list), 0,
    IsNumericList(arg'list)
    )
  new'name @ arg'list;
];
```

Now, this just does not do anything remotely right. {MacroRule} evaluates its arguments. Since {arg'list} is an atom and not a list of numbers at the time we are defining this, {IsNumericList(arg'list)} will evaluate to *False* and the new rule will be defined with a predicate that is always *False*, i.e. it will be never applied.

The right way to figure this out is to realize that the {MacroRule} call evaluates all its arguments and passes the results to a {Rule} call. So we need to see exactly what {Rule()} call we need to produce and then we need to prepare the arguments of {MacroRule} so that they evaluate to the right values. The {Rule()} call we need is something like this:

```
Rule("actual new name", <actual # of args>, 0,
  IsNumericList({actual arg list})
)  "actual new name" @ {actual arg list};
```

Note that we need to produce expressions such as {"new name" @ arg'list} and not results of evaluation of these expressions. We can produce these expressions by using {UnList()}, e.g.

```
UnList({Atom("@"), "Sin", {x}})
```

produces

```
"Sin" @ {x};
```

but not {Sin(x)}, and

```
UnList({IsNumericList, {1,2,x}})
```

produces the expression

```
IsNumericList({1,2,x});
```

which is not further evaluated.

Here is a second version of {NFunction()} that works:

```
NFunction(new'name, old'name, arg'list) := [
  MacroRuleBase(new'name, arg'list);
  MacroRule(new'name, Length(arg'list), 0,
    UnList({IsNumericList, arg'list})
  )
    UnList({Atom("@"), old'name, arg'list});
];
```

Note that we used {Atom("@")} rather than just the bare atom {@} because {@} is a prefix operator and prefix operator names as bare atoms do not parse (they would be confused with applications of a prefix operator to what follows).

Finally, there is a more concise (but less general) way of defining {NFunction()} for functions with known number of arguments, using the backquoting mechanism. The backquote operation will first substitute variables in an expression, without evaluating anything else, and then will evaluate the resulting expression a second time. The code for functions of just one variable may look like this:

```
N1Function(new'name, old'name) :=
    `( @new'name(x_IsNumber) <-- @old'name(x) );
```

This executes a little slower than the above version, because the backquote needs to traverse the expression twice, but makes for much more readable code.

4.7 Macro expansion

Yacas supports macro expansion (back-quoting). An expression can be back-quoted by putting a \` in front of it. Within the back-quoted expression, all atoms that have a @ in front of them get replaced with the value of that atom (treated as a variable), and then the resulting expression is evaluated:

```
In> x:=y
Out> y
In> `(@x:=2)
Out> 2
In> x
Out> y
In> y
Out> 2
```

This is useful in cases where within an expression one sub-expression is not evaluated. For instance, transformation rules can be built dynamically, before being declared. This is a particularly powerful feature that allows a programmer to write programs that write programs. The idea is borrowed from Lisp.

As the above example shows, there are similarities with the `Macro...` functions, that serve the same purpose for specific expressions. For example, for the above code, one could also have called `MacroSet`:

```
In> MacroSet(x,3)
Out> True;
In> x
Out> y;
In> y
Out> 3;
```

The difference is that `MacroSet`, and in general the `Macro...` functions, are faster than their back-quoted counterparts. This is because with back-quoting, first a new expression is built before it is evaluated. The advantages of back-quoting are readability and flexibility (the number of `Macro...` functions is limited, whereas back-quoting can be used anywhere).

When an @ operator is placed in front of a function call, the function call is replaced:

```
In> plus:=Add
Out> Add;
In> `(@plus(1,2,3))
Out> 6;
```

Application of pure functions is also possible by using macro expansion:

```
In> pure:={{a,b},a+b};
Out> {{a,b},a+b};
In> ` @pure(2,3);
Out> 5;
```

Pure (nameless) functions are useful for declaring a temporary function, that has functionality depending on the current environment it is in, or as a way to call driver functions. In the case of drivers (interfaces to specific functionality), a variable can be bound to a function to be evaluated to perform a specific task. That way several drivers can be around, with one bound to the variables holding the functions that will be called.

4.8 Scope of variable bindings

When setting variables or retrieving variable values, variables are automatically bound global by default. You can explicitly specify variables to be local to a block such as a function body; this will make them invisible outside the

block. Blocks have the form {[} statement1{;} statement2{;} {]} and local variables are declared by the {Local()} function.

When entering a block, a new stack frame is pushed for the local variables; it means that the code inside a block doesn't see the local variables of the caller either! You can tell the interpreter that a function should see local variables of the calling environment; to do this, declare UnFence(funcname, arity) on that function.

4.9 Evaluation of expressions

When programming in some language, it helps to have a mental model of what goes on behind the scenes when evaluating expressions, or in this case simplifying expressions.

This section aims to explain how evaluation (and simplification) of expressions works internally, in {Yacas}.

4.10 The LISP heritage

4.10.1 Representation of expressions

Much of the inner workings is based on how LISP-like languages are built up. When an expression is entered, or composed in some fashion, it is converted into a prefix form much like you get in LISP:

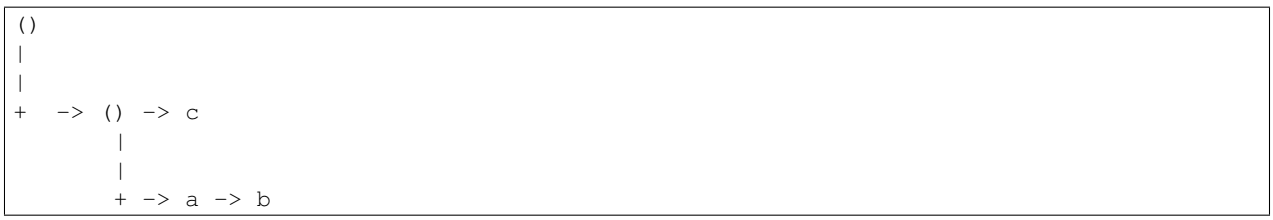
```
a+b    ->    (+ a b)
Sin(a) ->    (Sin a)
```

Here the sub-expression is changed into a list of so-called "atoms", where the first atom is a function name of the function to be invoked, and the atoms following are the arguments to be passed in as parameters to that function.

{Yacas} has the function {FullForm} to show the internal representation:

```
In> FullForm(a+b)
(+ a b )
Out> a+b;
In> FullForm(Sin(a))
(Sin a )
Out> Sin(a);
In> FullForm(a+b+c)
(+ (+ a b )c )
Out> a+b+c;
```

The internal representation is very close to what {FullForm} shows on screen. {a+b+c} would be {(+ (+ a b)c)} internally, or:



4.11 Evaluation

An expression like described above is done in the following manner: first the arguments are evaluated (if they need to be evaluated, {Yacas} can be told to not evaluate certain parameters to functions), and only then are these arguments passed in to the function for evaluation. They are passed in by binding local variables to the values, so these arguments are available as local values.

For instance, suppose we are evaluating `2*3+4`. This first gets changed to the internal representation `(+ (* 2 3 )4 )`. Then, during evaluation, the top expression refers to function "{+}". Its arguments are `(* 2 3)` and {4}. First `(* 2 3)` gets evaluated. This is a function call to the function `*` with arguments {2} and {3}, which evaluate to themselves. Then the function "{*}" is invoked with these arguments. The {Yacas} standard script library has code that accepts numeric input and performs the multiplication numerically, resulting in {6}.

The second argument to the top-level "{+}" is {4}, which evaluates to itself.

Now, both arguments to the "{+}" function have been evaluated, and the results are {6} and {4}. Now the "{+}" function is invoked. This function also has code in the script library to actually perform the addition when the arguments are numeric, so the result is 10:

```
In> FullForm(Hold(2*3+4))
(+ (* 2 3 )4 )
Out> 2*3+4;
In> 2*3+4
Out> 10;
```

Note that in {Yacas}, the script language does not define a "{+}" function in the core. This and other functions are all implemented in the script library. The feature "when the arguments to "{+}" are numeric, perform the numeric addition" is considered to be a "policy" which should be configurable. It should not be a part of the core language.

It is surprisingly difficult to keep in mind that evaluation is bottom up, and that arguments are evaluated before the function call is evaluated. In some sense, you might feel that the evaluation of the arguments is part of evaluation of the function. It is not. Arguments are evaluated before the function gets called.

Suppose we define the function {f}, which adds two numbers, and traces itself, as:

```
In> f(a,b):= \
In> [\
In> Local(result);\
In> Echo("Enter f with arguments ",a,b);\
In> result:=a+b;\
In> Echo("Leave f with result ",result);\
In> result;\
In> ];
Out> True;
```

Then the following interaction shows this principle:

```
In> f(f(2,3),4)
Enter f with arguments 2 3
Leave f with result 5
Enter f with arguments 5 4
Leave f with result 9
Out> 9;
```

The first Enter/Leave combination is for {f(2,3)}, and only then is the outer call to {f} entered.

This has important consequences for the way {Yacas} simplifies expressions: the expression trees are traversed bottom up, as the lowest parts of the expression trees are simplified first, before being passed along up to the calling function.

4.12 {Yacas}-specific extensions for CAS implementations

{Yacas} has a few language features specifically designed for use when implementing a CAS.

4.12.1 The transformation rules

Working with transformation rules is explained in the introduction and tutorial book. This section mainly deals with how {Yacas} works with transformation rules under the hood.

A transformation rule consists of two parts: a condition that an expression should match, and a result to be substituted for the expression if the condition holds. The most common way to specify a condition is a pattern to be matched to an expression.

A pattern is again simply an expression, stored in internal format:

```
In> FullForm(a_IsInteger+b_IsInteger*(_x))
(+ (_ a IsInteger )(* (_ b IsInteger )(_ x )))
Out> a _IsInteger+b _IsInteger*_x;
```

{Yacas} maintains structures of transformation rules, and tries to match them to the expression being evaluated. It first tries to match the structure of the pattern to the expression. In the above case, it tries to match to {a+b*x}. If this matches, local variables {a}, {b} and {x} are declared and assigned the sub-trees of the expression being matched. Then the predicates are tried on each of them: in this case, {IsInteger(a)} and {IsInteger(b)} should both return *True*.

Not shown in the above case, are post-predicates. They get evaluated afterwards. This post-predicate must also evaluate to *True*. If the structure of the expression matches the structure of the pattern, and all predicates evaluate to *True*, the pattern matches and the transformation rule is applied, meaning the right hand side is evaluated, with the local variables mentioned in the pattern assigned. This evaluation means all transformation rules are re-applied to the right-hand side of the expression.

Note that the arguments to a function are evaluated first, and only then is the function itself called. So the arguments are evaluated, and then the transformation rules applied on it. The main function defines its parameters also, so these get assigned to local variables also, before trying the patterns with their associated local variables.

Here is an example making the fact that the names in a pattern are local variables more explicit:

```
In> f1(_x,_a) <-- x+a
Out> True;
In> f2(_x,_a) <-- [Local(a); x+a;];
Out> True;
In> f1(1,2)
Out> 3;
In> f2(1,2)
Out> a+1;
```

4.12.2 Using different rules in different cases

In a lot of cases, the algorithm to be invoked depends on the type of the arguments. Or the result depends on the form of the input expression. This results in the typical "case" or "switch" statement, where the code to evaluate to determine the result depends on the form of the input expression, or the type of the arguments, or some other conditions.

{Yacas} allows to define several transformation rules for one and the same function, if the rules are to be applied under different conditions.

Suppose the function {f} is defined, a factorial function:

```
10 # f(0) <-- 1;
20 # f(n_IsPositiveInteger) <-- n*f(n-1);
```

Then interaction can look like:

```
In> f(3)
Out> 6;
In> f(a)
Out> f(a);
```

If the left hand side is matched by the expression being considered, then the right hand side is evaluated. A subtle but important thing to note is that this means that the whole body of transformation rules is thus re-applied to the right-hand side of the {<–} operator.

Evaluation goes bottom-up, evaluating (simplifying) the lowest parts of a tree first, but for a tree that matches a transformation rule, the substitution essentially means return the result of evaluating the right-hand side. Transformation rules are re-applied, on the right hand side of the transformation rule, and the original expression can be thought of as been substituted by the result of evaluating this right-hand side, which is supposed to be a "simpler" expression, or a result closer to what the user wants.

Internally, the function {f} is built up to resemble the following pseudo-code:

```
f(n)
{
   if (n = 1)
     return 1;
   else if (IsPositiveInteger(n))
     return n*f(n-1);
   else return f(n) unevaluated;
}
```

The transformation rules are thus combined into one big statement that gets executed, with each transformation rule being a if-clause in the statement to be evaluated. Transformation rules can be spread over different files, and combined in functional groups. This adds to the readability. The alternative is to write the full body of each function as one big routine, which becomes harder to maintain as the function becomes larger and larger, and hard or impossible to extend.

One nice feature is that functionality is easy to extend without modifying the original source code:

```
In> Ln(x*y)
Out> Ln(x*y);
In> Ln(_x*_y) <-- Ln(x) + Ln(y)
Out> True;
In> Ln(x*y)
Out> Ln(x)+Ln(y);
```

This is generally not advisable, due to the fact that it alters the behavior of the entire system. But it can be useful in some instances. For instance, when introducing a new function {f(x)}, one can decide to define a derivative explicitly, and a way to simplify it numerically:

```
In> f(_x)_InNumericMode() <-- Exp(x)
Out> True;
In> (Deriv(_x)f(_y)) <-- f(y)*(Deriv(x)y);
Out> True;
In> f(2)
Out> f(2);
In> N(f(2))
Out> 7.3890560989;
In> Exp(2)
Out> Exp(2);
In> N(Exp(2))
```

```
Out> 7.3890560989;
In> D(x)f(a*x)
Out> f(a*x)*a;
```

4.13 The "Evaluation is Simplification" hack

One of the ideas behind the {Yacas} scripting language is that evaluation is used for simplifying expressions. One consequence of this is that objects can be returned unevaluated when they can not be simplified further. This happens to variables that are not assigned, functions that are not defined, or function invocations where the arguments passed in as parameters are not actually handled by any code in the scripts. An integral that can not be performed by {Yacas} should be returned unevaluated:

```
In> 2+3
Out> 5;
In> a+b
Out> a+b;
In> Sin(a)
Out> Sin(a);
In> Sin(0)
Out> 0;
In> Integrate(x)Ln(x)
Out> x*Ln(x)-x;
In> Integrate(x)Ln(Sin(x))
Out> Integrate(x)Ln(Sin(x));
In> a!
Out> a!;
In> 3!
Out> 6;
```

Other languages usually do not allow evaluation of unbound variables, or undefined functions. In {Yacas}, these are interpreted as some yet undefined global variables or functions, and returned unevaluated.

4.14 Destructive operations

{Yacas} tries to keep as few copies of objects in memory as possible. Thus when assigning the value of one variable to another, a reference is copied, and both variables refer to the same memory, physically. This is relevant for programming; for example, one should use {FlatCopy} to actually make a new copy of an object. Another feature relevant to reference semantics is "destructive operations"; these are functions that modify their arguments rather than work on a copy. Destructive operations on lists are generally recognized because their name starts with "Destructive", e.g. {DestructiveDelete}. One other destructive operation is assignment of a list element through {list[index] := ...}.

Some examples to illustrate destructive operations on lists:

```
In> x1:={a,b,c}
Out> {a,b,c};
```

A list {x1} is created.

```
In> FullForm(x1)
(List a b c )
Out> {a,b,c};
In> x2:=z:x1
Out> {z,a,b,c};
```

A new list {x2} is {z} appended to {x1}. The {:} operation creates a copy of {x1} before appending, so {x1} is unchanged by this.

```
In> FullForm(x2)
(List z a b c )
Out> {z,a,b,c};
In> x2[1]:=y
Out> True;
```

We have modified the first element of {x2}, but {x1} is still the same.

```
In> x2
Out> {y,a,b,c};
In> x1
Out> {a,b,c};
In> x2[2]:=A
Out> True;
```

We have modified the second element of {x2}, but {x1} is still the same.

```
In> x2
Out> {y,A,b,c};
In> x1
Out> {a,b,c};
In> x2:=x1
Out> {A,b,c};
```

Now {x2} and {x1} refer to the same list.

```
In> x2[1]:=A
Out> True;
```

We have modified the first element of {x2}, and {x1} is also modified.

```
In> x2
Out> {A,b,c};
In> x1
Out> {A,b,c};
```

A programmer should always be cautious when dealing with destructive operations. Sometimes it is not desirable to change the original expression. The language deals with it this way because of performance considerations. Operations can be made non-destructive by using {FlatCopy}:

```
In> x1:={a,b,c}
Out> {a,b,c};
In> DestructiveReverse(x1)
Out> {c,b,a};
In> x1
Out> {a};
In> x1:={a,b,c}
Out> {a,b,c};
In> DestructiveReverse(FlatCopy(x1))
Out> {c,b,a};
In> x1
Out> {a,b,c};
```

{FlatCopy} copies the elements of an expression only at the top level of nesting. This means that if a list contains sub-lists, they are not copied, but references to them are copied instead:

```
In> dict1:={}
Out> {};
In> dict1["name"]:="John";
Out> True;
In> dict2:=FlatCopy(dict1)
Out> {{"name","John"}};
In> dict2["name"]:="Mark";
Out> True;
In> dict1
Out> {{"name","Mark"}};
```

A workaround for this is to use {Subst} to copy the entire tree:

```
In> dict1:={}
Out> {};
In> dict1["name"]:="John";
Out> True;
In> dict2:=Subst(a,a)(dict1)
Out> {{"name","John"}};
In> dict2["name"]:="Mark";
Out> True;
In> dict1
Out> {{"name","John"}};
In> dict2
Out> {{"name","Mark"}};
```

4.15 Coding style

4.15.1 Introduction

This chapter intends to describe the coding style and conventions applied in Yacas in order to make sure the engine always returns the correct result. This is an attempt at fending off such errors by combining rule-based programming with a clear coding style which should make help avoid these mistakes.

4.15.2 Interactions of rules and types

One unfortunate disadvantage of rule-based programming is that rules can sometimes cooperate in unwanted ways.

One example of how rules can produce unwanted results is the rule `a*0 <-- 0`. This would always seem to be true. However, when a is a vector, e.g. {a:={b,c,d}}, then `a*0` should actually return {{0,0,0}}, that is, a zero vector. The rule `a*0 <-- 0` actually changes the type of the expression from a vector to an integer! This can have severe consequences when other functions using this expressions as an argument expect a vector, or even worse, have a definition of how to work on vectors, and a different one for working on numbers.

When writing rules for an operator, it is assumed that the operator working on arguments, e.g. {Cos} or $\star$, will always have the same properties regardless of the arguments. The Taylor series expansion of $Cos(a)$ is the same regardless of whether a is a real number, complex number or even a matrix. Certain trigonometric identities should hold for the {Cos} function, regardless of the type of its argument.

If a function is defined which does not adhere to these rules when applied to another type, a different function name should be used, to avoid confusion.

By default, if a variable has not been bound yet, it is assumed to be a number. If it is in fact a more complex object, e.g. a vector, then you can declare it to be an "incomplete type" vector, using {Object("IsVector",x)} instead of {x}.

This expression will evaluate to {x} if and only if {x} is a vector at that moment of evaluation. Otherwise it returns unevaluated, and thus stays an incomplete type.

So this means the type of a variable is numeric unless otherwise stated by the user, using the "{Object}" command. No rules should ever work on incomplete types. It is just meant for delayed simplification.

The topic of implicit type of an object is important, since many rules need to assume something about their argument types.

4.15.3 Ordering of rules

The implementor of a rule set can specify the order in which rules should be tried. This can be used to let the engine try more specific rules (those involving more elements in the pattern) before trying less specific rules. Ordering of rules can be also explicitly given by precedence numbers. The Yacas engine will split the expression into subexpressions, and will try to apply all matching rules to a given subexpression in order of precedence.

A rule with precedence 100 is defined by the syntax such as

```
100 # f(_x + _y) <-- f(x) + f(y);
```

The problem mentioned above with a rule for vectors and scalars could be solved by making two rules:

- 1. $a*b$ (if b is a vector and a is a number) {<–} return vector of each component multiplied by a.
- 1. $a*0$ {<–} 0

So vector multiplication would be tried first.

The ordering of the precedence of the rules in the standard math scripts is currently:

- 50-60: Args are numbers: directly calculate. These are put in the beginning, so they are tried first. This is useful for quickly obtaining numeric results if all the arguments are numeric already, and symbolic transformations are not necessary.
- 100-199: tautologies. Transformations that do not change the type of the argument, and are always true.
- 200-399: type-specific transformations. Transformations for specific types of objects.
- 400-599: transformations on scalars (variables are assumed to be scalars). Meaning transformations that can potentially change the type of an argument.

4.16 Writing new library functions

When you implement new library functions, you need to make your new code compatible and consistent with the rest of the library. Here are some relevant considerations.

4.16.1 To evaluate or not to evaluate

Currently, a general policy in the library is that functions do nothing unless their arguments actually allow something to be evaluated. For instance, if the function expects a variable name but instead gets a list, or expects a list but instead gets a string, in most cases it seems to be a good idea to do nothing and return unevaluated. The unevaluated expression will propagate and will be easy to spot. Most functions can accomplish this by using type-checking predicates such as {IsInteger} in rules.

When dealing with numbers, Yacas tries to maintain exact answers as much as possible and evaluate to floating-point only when explicitly told so (using {N()}). The general evaluation strategy for numerical functions such as {Sin} or {Gamma} is the following:

- If {InNumericMode()} returns *True* and the arguments are numbers (perhaps complex numbers), the function should evaluate its result in floating-point to current precision.
- Otherwise, if the arguments are such that the result can be calculated exactly, it should be evaluated and returned. E.g. {Sin(Pi/2)} returns {1}.
- Otherwise the function should return unevaluated (but usually with its arguments evaluated).

Here are some examples of this behavior:

```
In> Sin(3)
Out> Sin(3);
In> Gamma(8)
Out> 5040;
In> Gamma(-11/2)
Out> (64*Sqrt(Pi))/10395;
In> Gamma(8/7)
Out> Gamma(8/7);
In> N(Gamma(8/7))
Out> 0.9354375629;
In> N(Gamma(8/7+x))
Out> Gamma(x+1.1428571428);
In> Gamma(12/6+x)
Out> Gamma(x+2);
```

To implement this behavior, {Gamma} and other mathematical functions usually have two variants: the "symbolic" one and the "numerical" one. For instance, there are {Sin} and {MathSin}, {Ln} and {Internal'LnNum}, {Gamma} and {Internal'GammaNum}. (Here {MathSin} happens to be a core function but it is not essential.) The "numerical" functions always evaluate to floating-point results. The "symbolic" function serves as a front-end; it evaluates when the result can be expressed exactly, or calls the "numerical" function if {InNumericMode()} returns *True*, and otherwise returns unevaluated.

The "symbolic" function usually has multiple rules while the "numerical" function is usually just one large block of number-crunching code.

4.16.2 Using {N()} and {InNumericMode()} in scripts

As a rule, {N()} should be avoided in code that implements basic numerical algorithms. This is because {N()} itself is implemented in the library and it may need to use some of these algorithms. Arbitrary-precision math can be handled by core functions such as {MathDivide}, {MathSin} and so on, without using {N()}. For example, if your code needs to evaluate $Sqrt(Pi)$ to many digits as an intermediate result, it is better to write {MathSqrt(Internal'Pi())} than {N(Sqrt(Pi))} because it makes for faster, more reliable code.

4.16.3 Using {Builtin'Precision'Set()}

The usual assumption is that numerical functions will evaluate floating-point results to the currently set precision. For intermediate calculations, a higher working precision is sometimes needed. In this case, your function should set the precision back to the original value at the end of the calculation and round off the result.

4.16.4 Using verbose mode

For routines using complicated algorithms, or when evaluation takes a long time, it is usually helpful to print some diagnostic information, so that the user can at least watch some progress. The current convention is that if {InVerboseMode()} returns *True*, functions may print diagnostic information. (But do not print too much!). Verbose mode is turned on by using the function {V(expression)}. The expression is evaluated in verbose mode.

4.16.5 Procedural programming or rule-based programming?

Two considerations are relevant to this decision. First, whether to use multiple rules with predicates or one rule with multiple {If()}s. Consider the following sample code for the "double factorial" function :math:n!! := n(n-2)...‘ written using predicates and rules:

```
1# 0 !! <-- 1;
1# 1 !! <-- 1;
2# (n_IsEven) !! <-- 2^(n/2)*n!;
3# (n_IsOdd) !! <-- n*(n-2)!!;
```

and an equivalent code with one rule:

```
n!! := If(n=0 Or n=1, 1,
  If(IsEven(n), 2^(n/2)*n!,
  If(IsOdd(n), n*(n-2)!!, Hold(n!!)))
);
```

(Note: This is not the way $n!!$ is implemented in the library.) The first version is a lot more clear. Yacas is very quick in rule matching and evaluation of predicates, so the first version is (marginally) faster. So it seems better to write a few rules with predicates than one rule with multiple {If()} statements.

The second question is whether to use recursion or loops. Recursion makes code more elegant but it is slower and limited in depth. Currently the default recursion depth of 1000 is enough for most casual calculations and yet catches infinite recursion errors relatively quickly. Because of clearer code, it seems better to use recursion in situations where the number of list elements will never become large. In numerical applications, such as evaluation of Taylor series, recursion usually does not pay off.

4.16.6 Reporting errors

Errors occurring because of invalid argument types should be reported only if absolutely necessary. (In the future there may be a static type checker implemented that will make explicit checking unnecessary.)

Errors of invalid values, e.g. a negative argument of real logarithm function, or a malformed list, mean that a human has probably made a mistake, so the errors need to be reported. "Internal errors", i.e. program bugs, certainly need to be reported.

There are currently two facilities for reporting errors: a "hard" one and a "soft" one.

The "hard" error reporting facility is the function {Check}. For example, if {x}={-1}, then

```
Check(x>0,"bad x");
```

will immediately halt the execution of a Yacas script and print the error messsage. This is implemented as a C++ exception. A drawback of this mechanism is that the Yacas stack unwinding is not performed by the Yacas interpreter, so global variables such as {InNumericMode()}, {Verbose}, {Builtin'Precision'Set()} may keep the intermediate values they had been assigned just before the error occurred. Also, sometimes it is better for the program to be able to catch the error and continue.

Todo

the above will hopefully be solved soon, as we can now trap exceptions in the scripts.

The "soft" error reporting is provided by the functions {Assert} and {IsError}, e.g.

```
Assert("domain", x) x>0;
If(IsError("domain"), ...);
```

The error will be reported but execution will continue normally until some other function "handles" the error (prints the error message or does something else appropriate for that error). Here the string {"domain"} is the "error type" and {x} will be the information object for this error. The error object can be any expression, but it is probably a good idea to choose a short and descriptive string for the error type.

The function {GetErrorTableau()} returns an associative list that accumulates all reported error objects. When errors are "handled", their objects should be removed from the list. The utility function {DumpErrors()} is a simple error handler that prints all errors and clears the list. Other handlers are {GetError} and {ClearError}. These functions may be used to handle errors when it is safe to do so.

The "soft" error reporting facility is safer and more flexible than the "hard" facility. However, the disadvantage is that errors are not reported right away and pointless calculations may continue for a while until an error is handled.

4.17 Advanced example 1: parsing expressions ({CForm})

In this chapter we show how Yacas represents expressions and how one can build functions that work on various types of expressions. Our specific example will be {CForm()}, a standard library function that converts Yacas expressions into C or C++ code. Although the input format of Yacas expressions is already very close to C and perhaps could be converted to C by means of an external text filter, it is instructive to understand how to use Yacas to parse its own expressions and produce the corresponding C code. Here we shall only design the core mechanism of {CForm()} and build a limited version that handles only expressions using the four arithmetic actions.

4.17.1 Recursive parsing of expression trees

As we have seen in the tutorial, Yacas represents all expressions as trees, or equivalently, as lists of lists. For example, the expression "{a+b+c+d+e}" is for Yacas a tree of depth 4 that could be visualized as

```
 "+"
a  "+"
  b  "+"
    c  "+"
      d   e
```

or as a nested list: `("+" a ("+" b ("+" c ("+" d e))))`.

Complicated expressions are thus built from simple ones in a general and flexible way. If we want a function that acts on sums of any number of terms, we only need to define this function on a single atom and on a sum of two terms, and the Yacas engine will recursively perform the action on the entire tree.

So our first try is to define rules for transforming an atom and for transforming sums and products. The result of {CForm()} will always be a string. We can use recursion like this:

```
In> 100 # CForm(a_IsAtom) <-- String(a);
Out> True;
In> 100 # CForm(_a + _b) <-- CForm(a) : \
  " + " : CForm(b);
Out> True;
In> 100 # CForm(_a * _b) <-- CForm(a) : \
  " * " : CForm(b);
Out> True;
```

We used the string concatenation operator "{:}" and we added spaces around the binary operators for clarity. All rules have the same precedence 100 because there are no conflicts in rule ordering so far: these rules apply in mutually exclusive cases. Let's try converting some simple expressions now:

```
In> CForm(a+b*c);
Out> "a + b * c";
In> CForm(a+b*c*d+e+1+f);
Out> "a + b * c * d + e + 1 + f";
```

With only three rules, we were able to process even some complicated expressions. How did it work? We could illustrate the steps Yacas went through when simplifying {CForm(a+b*c)} roughly like this:

```
CForm(a+b*c)
    ... apply 2nd rule
CForm(a) : " + " : CForm(b*c)
    ... apply 1st rule and 3rd rule
"a" : " + " : CForm(b) : " * " : CForm(c)
    ... apply 1st rule
"a" : " + " : "b" : " * " : "c"
    ... concatenate strings
"a + b * c"
```

4.17.2 Handling precedence of infix operations

It seems that recursion will do all the work for us. The power of recursion is indeed great and extensive use of recursion is built into the design of Yacas. We might now add rules for more operators, for example, the unary addition, subtraction and division:

```
100 # CForm(+ _a) <-- "+ " : CForm(a);
100 # CForm(- _a) <-- "- " : CForm(a);
100 # CForm(_a - _b) <-- CForm(a) : " - "
  : CForm(b);
100 # CForm(_a / _b) <-- CForm(a) : " / "
  : CForm(b);
```

However, soon we find that we forgot about operator precedence. Our simple-minded {CForm()} gives wrong C code for expressions like this:

```
In> CForm( (a+b) * c );
Out> "a + b * c";
```

We need to get something like `(a+b)*c` in this case. How would we add a rule to insert parentheses around subexpressions? A simple way out would be to put parentheses around every subexpression, replacing our rules by something like this:

```
100 # CForm(_a + _b) <-- "(" : CForm(a)
  : " + " : CForm(b) : ")";
100 # CForm(- _a) <-- "(- " : CForm(a)
  : ")";
```

and so on. This will always produce correct C code, e.g. in our case "((a+b)*c)", but generally the output will be full of unnecessary parentheses. It is instructive to find a better solution.

We could improve the situation by inserting parentheses only if the higher-order expression requires them; for this to work, we need to make a call such as {CForm(a+b)} aware that the enveloping expression has a multiplication by {c} around the addition {a+b}. This can be implemented by passing an extra argument to {CForm()} that will indicate the precedence of the enveloping operation. A compound expression that uses an infix operator must be bracketed if the precedence of that infix operator is higher than the precedence of the enveloping infix operation.

We shall define an auxiliary function also named "CForm" but with a second argument, the precedence of the enveloping infix operation. If there is no enveloping operation, we shall set the precedence to a large number, e.g. 60000, to indicate that no parentheses should be inserted around the whole expression. The new "CForm(expr, precedence)"

will handle two cases: either parentheses are necessary, or unnecessary. For clarity we shall implement these cases in two separate rules. The initial call to "CForm(expr)" will be delegated to "CForm(expr, precedence)".

The precedence values of infix operators such as "{+}" and "{*}" are defined in the Yacas library but may change in a future version. Therefore, we shall not hard-code these precedence values but instead use the function {OpPrecedence()} to determine them. The new rules for the "{+}" operation could look like this:

```
PlusPrec := OpPrecedence("+");
100 # CForm(_expr) <-- CForm(expr, 60000);
100 # CForm(_a + _b, _prec)_(PlusPrec>prec)
  <-- "(" : CForm(a, PlusPrec) : " + "
  : CForm(b, PlusPrec) : ")";
120 # CForm(_a + _b, _prec) <--
    CForm(a, PlusPrec) : " + "
    : CForm(b, PlusPrec);
```

and so on. We omitted the predicate for the last rule because it has a later precedence than the preceding rule.

The way we wrote these rules is unnecessarily repetitive but straightforward and it illustrates the central ideas of expression processing in Yacas. The standard library implements {CForm()} essentially in this way. In addition the library implementation supports standard mathematical functions, arrays and so on, and is somewhat better organized to allow easier extensions and avoid repetition of code.

4.18 Yacas programming pitfalls

No programming language is without programming pitfalls, and {Yacas} has its fair share of pitfalls.

4.18.1 All rules are global

All rules are global, and a consequence is that rules can clash or silently shadow each other, if the user defines two rules with the same patterns and predicates but different bodies.

For example:

```
In> f(0) <-- 1
Out> True;
In> f(x_IsConstant) <-- Sin(x)/x
Out> True;
```

This can happen in practice, if care is not taken. Here two transformation rules are defined which both have the same precedence (since their precedence was not explicitly set). In that case {Yacas} gets to decide which one to try first. Such problems can also occur where one transformation rule (possibly defined in some other file) has a wrong precedence, and thus masks another transformation rule. It is necessary to think of a scheme for assigning precedences first. In many cases, the order in which transformation rules are applied is important.

In the above example, because {Yacas} gets to decide which rule to try first, it is possible that f(0) invokes the second rule, which would then mask the first so the first rule is never called. Indeed, in {Yacas} version 1.0.51,

```
In> f(0)
Out> Undefined;
```

The order the rules are applied in is undefined if the precedences are the same. The precedences should only be the same if order does not matter. This is the case if, for instance, the two rules apply to different argument patters that could not possibly mask each other.

The solution could have been either:

```
In> 10 # f(0) <-- 1
Out> True;
In> 20 # f(x_IsConstant) <-- Sin(x)/x
Out> True;
In> f(0)
Out> 1;
```

or

```
In> f(0) <-- 1
Out> True;
In> f(x_IsConstant)_(x != 0) <-- Sin(x)/x
Out> True;
In> f(0)
Out> 1;
```

So either the rules should have distinct precedences, or they should have mutually exclusive predicates, so that they do not collide.

4.18.2 Objects that look like functions

An expression that looks like a "function", for example {AbcDef(x,y)}, is in fact either a call to a "core function" or to a "user function", and there is a huge difference between the behaviors. Core functions immediately evaluate to something, while user functions are really just symbols to which evaluation rules may or may not be applied.

For example:

```
In> a+b
Out> a+b;
In> 2+3
Out> 5;
In> MathAdd(a,b)
In function "MathAdd" :
bad argument number 1 (counting from 1)
The offending argument a evaluated to a
CommandLine(1) : Invalid argument

In> MathAdd(2,3)
Out> 5;
```

The {+} operator will return the object unsimplified if the arguments are not numeric. The {+} operator is defined in the standard scripts. {MathAdd}, however, is a function defined in the "core" to performs the numeric addition. It can only do this if the arguments are numeric and it fails on symbolic arguments. (The {+} operator calls {MathAdd} after it has verified that the arguments passed to it are numeric.)

A core function such as {MathAdd} can never return unevaluated, but an operator such as "{+}" is a "user function" which might or might not be evaluated to something.

A user function does not have to be defined before it is used. A consequence of this is that a typo in a function name or a variable name will always go unnoticed. For example:

```
In> f(x_IsInteger,y_IsInteger) <-- Mathadd(x,y)
Out> True;
In> f(1,2)
Out> Mathadd(1,2);
```

Here we made a typo: we should have written {MathAdd}, but wrote {Mathadd} instead. {Yacas} happily assumed that we mean a new and (so far) undefined "user function" {Mathadd} and returned the expression unevaluated.

In the above example it was easy to spot the error. But this feature becomes more dangerous when it this mistake is made in a part of some procedure. A call that should have been made to an internal function, if a typo was made, passes silently without error and returns unevaluated. The real problem occurs if we meant to call a function that has side-effects and we not use its return value. In this case we shall not immediately find that the function was not evaluated, but instead we shall encounter a mysterious bug later.

4.18.3 Guessing when arguments are evaluated and when not

If your new function does not work as expected, there is a good chance that it happened because you did not expect some expression which is an argument to be passed to a function to be evaluated when it is in fact evaluated, or vice versa.

For example:

```
In> p:=Sin(x)
Out> Sin(x);
In> D(x)p
Out> Cos(x);
In> y:=x
Out> x;
In> D(y)p
Out> 0;
```

Here the first argument to the differentiation function is not evaluated, so {y} is not evaluated to {x}, and {D(y)p} is indeed 0.

4.18.4 The confusing effect of {HoldArg}

The problem of distinguishing evaluated and unevaluated objects becomes worse when we need to create a function that does not evaluate its arguments.

Since in {Yacas} evaluation starts from the bottom of the expression tree, all "user functions" will appear to evaluate their arguments by default. But sometimes it is convenient to prohibit evaluation of a particular argument (using {HoldArg} or {HoldArgNr}).

For example, suppose we need a function {A(x,y)} that, as a side-effect, assigns the variable {x} to the sum of {x} and {y}. This function will be called when {x} already has some value, so clearly the argument {x} in {A(x,y)} should be unevaluated. It is possible to make this argument unevaluated by putting {Hold()} on it and always calling {A(Hold(x), y)}, but this is not very convenient and easy to forget. It would be better to define {A} so that it always keeps its first argument unevaluated.

If we define a rule base for {A} and declare {HoldArg},

```
Function() A(x,y);
HoldArg("A", x);
```

then we shall encounter a difficulty when working with the argument {x} inside of a rule body for {A}. For instance, the simple-minded implementation

```
A(_x, _y) <-- (x := x+y);
```

does not work:

```
In> [ a:=1; b:=2; A(a,b);]
Out> a+2;
```

In other words, the {x} inside the body of {A(x,y)} did not evaluate to {1} when we called the function {:=}. Instead, it was left unevaluated as the atom {x} on the left hand side of {:=}, since {:=} does not evaluate its left argument. It however evaluates its right argument, so the {y} argument was evaluated to {2} and the {x+y} became {a+2}.

The evaluation of {x} in the body of {A(x,y)} was prevented by the {HoldArg} declaration. So in the body, {x} will just be the atom {x}, unless it is evaluated again. If you pass {x} to other functions, they will just get the atom {x}. Thus in our example, we passed {x} to the function {:=}, thinking that it will get {a}, but it got an unevaluated atom {x} on the left side and proceeded with that.

We need an explicit evaluation of {x} in this case. It can be performed using {Eval}, or with backquoting, or by using a core function that evaluates its argument. Here is some code that illustrates these three possibilities:

```
A(_x, _y) <-- [ Local(z); z:=Eval(x); z:=z+y; ]
```

(using explicit evaluation) or

```
A(_x, _y) <-- `(@x := @x + y);
```

(using backquoting) or

```
A(_x, _y) <-- MacroSet(x, x+y);
```

(using a core function {MacroSet} that evaluates its first argument).

However, beware of a clash of names when using explicit evaluations (as explained above). In other words, the function {A} as defined above will not work correctly if we give it a variable also named {x}. The {LocalSymbols} call should be used to get around this problem.

Another caveat is that when we call another function that does not evaluate its argument, we need to substitute an explicitly evaluated {x} into it. A frequent case is the following: suppose we have a function {B(x,y)} that does not evaluate {x}, and we need to write an interface function {B(x)} which will just call {B(x,0)}. We should use an explicit evaluation of {x} to accomplish this, for example

```
B(_x) <-- `B(@x,0);
```

or

```
B(_x) <-- B @ {x, 0};
```

Otherwise {B(x,y)} will not get the correct value of its first parameter {x}.

4.18.5 Special behavior of {Hold}, {UnList} and {Eval}

When an expression is evaluated, all matching rules are applied to it repeatedly until no more rules match. Thus an expression is "completely" evaluated. There are, however, two cases when recursive application of rules is stopped at a certain point, leaving an expression not "completely" evaluated:

- The expression which is the result of a call to a Yacas core function is not evaluated further, even if some rules apply to it.
- The expression is a variable that has a value assigned to it; for example, the variable {x} might have the expression {y+1} as the value. That value is not evaluated again, so even if {y} has been assigned another value, say, {y=2} a Yacas expression such as {2*x+1} will evaluate to {2*(y+1)+1} and not to {7}. Thus, a variable can have some unevaluated expression as its value and the expression will not be re-evaluated when the variable is used.

The first possibility is mostly without consequence because almost all core functions return a simple atom that does not require further evaluation. However, there are two core functions that can return a complicated expression: {Hold} and {UnList}. Thus, these functions can produce arbitrarily complicated Yacas expressions that will be left unevaluated. For example, the result of

```
UnList({Sin, 0})
```

is the same as the result of

```
Hold(Sin(0))
```

and is the unevaluated expression {Sin(0)} rather than {0}.

Typically you want to use {UnList} because you need to construct a function call out of some objects that you have. But you need to call {Eval(UnList(...))} to actually evaluate this function call. For example:

```
In> UnList({Sin, 0})
Out> Sin(0);
In> Eval(UnList({Sin, 0}))
Out> 0;
```

In effect, evaluation can be stopped with {Hold} or {UnList} and can be explicitly restarted by using {Eval}. If several levels of un-evaluation are used, such as {Hold(Hold(...))}, then the same number of {Eval} calls will be needed to fully evaluate an expression.

```
In> a:=Hold(Sin(0))
Out> Sin(0);
In> b:=Hold(a)
Out> a;
In> c:=Hold(b)
Out> b;
In> Eval(c)
Out> a;
In> Eval(Eval(c))
Out> Sin(0);
In> Eval(Eval(Eval(c)))
Out> 0;
```

A function {FullEval} can be defined for "complete" evaluation of expressions, as follows:

```
LocalSymbols(x,y)
[
  FullEval(_x) <-- FullEval(x,Eval(x));
  10 # FullEval(_x,_x) <-- x;
  20 # FullEval(_x,_y) <-- FullEval(y,Eval(y));
];
```

Then the example above will be concluded with:

```
In> FullEval(c);
Out> 0;
```

4.18.6 Correctness of parameters to functions is not checked

Because {Yacas} does not enforce type checking of arguments, it is possible to call functions with invalid arguments. The default way functions in {Yacas} should deal with situations where an action can not be performed, is to return the expression unevaluated. A function should know when it is failing to perform a task. The typical symptoms are errors that seem obscure, but just mean the function called should have checked that it can perform the action on the object.

For example:

```
In> 10 # f(0) <-- 1;
Out> True;
In> 20 # f(_n) <-- n*f(n-1);
Out> True;
In> f(3)
Out> 6;
In> f(1.3)
CommandLine(1): Max evaluation stack depth reached.
```

Here, the function {f} is defined to be a factorial function, but the function fails to check that its argument is a positive integer, and thus exhausts the stack when called with a non-integer argument. A better way would be to write

```
In> 20 # f(n_IsPositiveInteger) <-- n*f(n-1);
```

Then the function would have returned unevaluated when passed a non-integer or a symbolic expression.

4.18.7 Evaluating variables in the wrong scope

There is a subtle problem that occurs when {Eval} is used in a function, combined with local variables. The following example perhaps illustrates it:

```
In> f1(x):=[Local(a);a:=2;Eval(x);];
Out> True;
In> f1(3)
Out> 3;
In> f1(a)
Out> 2;
```

Here the last call should have returned {a}, but it returned {2}, because {x} was assigned the value {a}, and {a} was assigned locally the value of {2}, and {x} gets re-evaluated. This problem occurs when the expression being evaluated contains variables which are also local variables in the function body. The solution is to use the {LocalSymbols} function for all local variables defined in the body.

The following illustrates this:

```
In> f2(x):=LocalSymbols(a)[Local(a);a:=2;Eval(x);];
Out> True;
In> f1(3)
Out> 3;
In> f2(a)
Out> a;
```

Here {f2} returns the correct result. {x} was assigned the value {a}, but the {a} within the function body is made distinctly different from the one referred to by {x} (which, in a sense, refers to a global {a}), by using {LocalSymbols}.

This problem generally occurs when defining functions that re-evaluate one of its arguments, typically functions that perform a loop of some sort, evaluating a body at each iteration.

4.19 Debugging in Yacas

4.19.1 Introduction

When writing a code segment, it is generally a good idea to separate the problem into many small functions. Not only can you then reuse these functions on other problems, but it makes debugging easier too.

For debugging a faulty function, in addition to the usual trial-and-error method and the "print everything" method, Yacas offers some trace facilities. You can try to trace applications of rules during evaluation of the function ({TraceRule()}, {TraceExp()}) or see the stack after an error has occurred ({TraceStack()}).

There is also an interactive debugger, which shall be introduced in this chapter.

Finally, you may want to run a debugging version of Yacas. This version of the executable maintains more information about the operations it performs, and can report on this.

This chapter will start with the interactive debugger, as it is the easiest and most useful feature to use, and then proceed to explain the trace and profiling facilities. Finally, the internal workings of the debugger will be explained. It is highly customizable (in fact, most of the debugging code is written in Yacas itself), so for bugs that are really difficult to track one can write custom code to track it.

4.19.2 The trace facilities

The trace facilities are:

- {TraceExp} : traces the full expression, showing all calls to user- or system-defined functions, their arguments, and the return values. For complex functions this can become a long list of function calls.
- {TraceRule} : traces one single user-defined function (rule). It shows each invocation, the arguments passed in, and the returned values. This is useful for tracking the behavior of that function in the environment it is intended to be used in.
- {TraceStack} : shows a few last function calls before an error has occurred.
- {Profile} : report on statistics (number of times functions were called, etc.). Useful for performance analysis.

The online manual pages (e.g. {?TraceStack}) have more information about the use of these functions.

An example invocation of {TraceRule} is

```
In> TraceRule(x+y)2+3*5+4;
```

Which should then show something to the effect of

```
  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;
```

4.19.3 Custom evaluation facilities

Yacas supports a special form of evaluation where hooks are placed when evaluation enters or leaves an expression.

This section will explain the way custom evaluation is supported in {Yacas}, and will proceed to demonstrate how it can be used by showing code to trace, interactively step through, profile, and write custom debugging code.

Debugging, tracing and profiling has been implemented in the debug.rep/ module, but a simplification of that code will be presented here to show the basic concepts.

4.19.4 The basic infrastructure for custom evaluation

The name of the function is {CustomEval}, and the calling sequence is:

```
CustomEval(enter,leave,error,expression);
```

Here, {expression} is the expression to be evaluated, {enter} some expression that should be evaluated when entering an expression, and {leave} an expression to be evaluated when leaving evaluation of that expression.

The {error} expression is evaluated when an error occurred. If an error occurs, this is caught high up, the {error} expression is called, and the debugger goes back to evaluating {enter} again so the situation can be examined. When the debugger needs to stop, the {error} expression is the place to call {CustomEval'Stop()} (see explanation below).

The {CustomEval} function can be used to write custom debugging tools. Examples are:

- a trace facility following entering and leaving functions
- interactive debugger for stepping through evaluation of an expression.
- profiler functionality, by having the callback functions do the bookkeeping on counts of function calls for instance.

In addition, custom code can be written to for instance halt evaluation and enter interactive mode as soon as some very specific situation occurs, like "stop when function foo is called while the function bar is also on the call stack and the value of the local variable x is less than zero".

As a first example, suppose we define the functions TraceEnter(), TraceLeave() and {TraceExp()} as follows:

```
TraceStart() := [indent := 0;];
TraceEnter() :=
[
   indent++;
   Space(2*indent);
   Echo("Enter ",CustomEval'Expression());
];
TraceLeave() :=
[
   Space(2*indent);
   Echo("Leave ",CustomEval'Result());
   indent--;
];
Macro(TraceExp,{expression})
[
   TraceStart();
   CustomEval(TraceEnter(),
              TraceLeave(),
              CustomEval'Stop(),@expression);
];
```

allows us to have tracing in a very basic way. We can now call:

```
In> TraceExp(2+3)
  Enter 2+3
    Enter 2
    Leave 2
    Enter 3
    Leave 3
    Enter IsNumber(x)
      Enter x
      Leave 2
    Leave True
```

```
    Enter IsNumber(y)
      Enter y
      Leave 3
    Leave True
    Enter True
    Leave True
    Enter MathAdd(x,y)
      Enter x
      Leave 2
      Enter y
      Leave 3
    Leave 5
  Leave 5
Out> 5;
```

This example shows the use of {CustomEval'Expression} and {CustomEval'Result}. These functions give some extra access to interesting information while evaluating the expression. The functions defined to allow access to information while evaluating are:

- {CustomEval'Expression()} - return expression currently on the top call stack for evaluation.
- {CustomEval'Result()} - when the {leave} argument is called this function returns what the evaluation of the top expression will return.
- {CustomEval'Locals()} - returns a list with the current local variables.
- {CustomEval'Stop()} - stop debugging execution

A simple interactive debugger

The following code allows for simple interactive debugging:

```
DebugStart():=
[
   debugging:=True;
   breakpoints:={};
];
DebugRun():= [debugging:=False;];
DebugStep():=[debugging:=False;nextdebugging:=True;];
DebugAddBreakpoint(fname_IsString) <--
   [ breakpoints := fname:breakpoints;];
BreakpointsClear() <-- [ breakpoints := {};];
Macro(DebugEnter,{})
[
   Echo(">>> ",CustomEval'Expression());
   If(debugging = False And
      IsFunction(CustomEval'Expression()) And
      Contains(breakpoints,
      Type(CustomEval'Expression())),
        debugging:=True);
   nextdebugging:=False;
   While(debugging)
   [
      debugRes:=
        Eval(FromString(
          ReadCmdLineString("Debug> "):";")
          Read());
      If(debugging,Echo("DebugOut> ",debugRes));
   ];
   debugging:=nextdebugging;
```

```
];
Macro(DebugLeave,{})
[
   Echo(CustomEval'Result(),
        " <-- ",CustomEval'Expression());
];
Macro(Debug,{expression})
[
   DebugStart();
   CustomEval(DebugEnter(),
              DebugLeave(),
              debugging:=True,@expression);
];
```

This code allows for the following interaction:

```
In> Debug(2+3)
>>> 2+3
Debug>
```

The console now shows the current expression being evaluated, and a debug prompt for interactive debugging. We can enter {DebugStep()}, which steps to the next expression to be evaluated:

```
Debug> DebugStep();
>>> 2
Debug>
```

This shows that in order to evaluate {2+3} the interpreter first needs to evaluate {2}. If we step further a few more times, we arrive at:

```
>>> IsNumber(x)
Debug>
```

Now we might be curious as to what the value for {x} is. We can dynamically obtain the value for {x} by just typing it on the command line.

```
>>> IsNumber(x)
Debug> x
DebugOut> 2
```

{x} is set to {2}, so we expect {IsNumber} to return *True*. Stepping again:

```
Debug> DebugStep();
>>> x
Debug> DebugStep();
2  <-- x
True  <-- IsNumber(x)
>>> IsNumber(y)
```

So we see this is true. We can have a look at which local variables are currently available by calling {CustomEval'Locals()}:

```
Debug> CustomEval'Locals()
DebugOut> {arg1,arg2,x,y,aLeftAssign,aRightAssign}
```

And when bored, we can proceed with {DebugRun()} which will continue the debugger until finished in this case (a more sophisticated debugger can add breakpoints, so running would halt again at for instance a breakpoint).

```
Debug> DebugRun()
>>> y
```

```
3  <-- y
True  <-- IsNumber(y)
>>> True
True  <-- True
>>> MathAdd(x,y)
>>> x
2  <-- x
>>> y
3  <-- y
5  <-- MathAdd(x,y)
5  <-- 2+3
Out> 5;
```

The above bit of code also supports primitive breakpointing, in that one can instruct the evaluator to stop when a function will be entered. The debugger then stops just before the arguments to the function are evaluated. In the following example, we make the debugger stop when a call is made to the {MathAdd} function:

```
In> Debug(2+3)
>>> 2+3
Debug> DebugAddBreakpoint("MathAdd")
DebugOut> {"MathAdd"}
Debug> DebugRun()
>>> 2
2  <-- 2
>>> 3
3  <-- 3
>>> IsNumber(x)
>>> x
2  <-- x
True  <-- IsNumber(x)
>>> IsNumber(y)
>>> y
3  <-- y
True  <-- IsNumber(y)
>>> True
True  <-- True
>>> MathAdd(x,y)
Debug>
```

The arguments to {MathAdd} can now be examined, or execution continued.

One great advantage of defining much of the debugger in script code can be seen in the {DebugEnter} function, where the breakpoints are checked, and execution halts when a breakpoint is reached. In this case the condition for stopping evaluation is rather simple: when entering a specific function, stop. However, nothing stops a programmer from writing a custom debugger that could stop on any condition, halting at e very special case.

4.19.5 Profiling

A simple profiler that counts the number of times each function is called can be written such:

```
ProfileStart():=
[
   profilefn:={};
];
10 # ProfileEnter()
     _(IsFunction(CustomEval'Expression())) <--
[
```

```
   Local(fname);
   fname:=Type(CustomEval'Expression());
   If(profilefn[fname]=Empty,profilefn[fname]:=0);
   profilefn[fname] := profilefn[fname]+1;
];
Macro(Profile,{expression})
[
   ProfileStart();
   CustomEval(ProfileEnter(),True,
              CustomEval'Stop(),@expression);
   ForEach(item,profilefn)
     Echo("Function ",item[1]," called ",
          item[2]," times");
];
```

which allows for the interaction:

```
In> Profile(2+3)
Function MathAdd called 1  times
Function IsNumber called 2  times
Function + called 1  times
Out> True;
```

4.20 Advanced example 2: implementing a non-commutative algebra

We need to understand how to simplify expressions in Yacas, and the best way is to try writing our own algebraic expression handler. In this chapter we shall consider a simple implementation of a particular non-commutative algebra called the Heisenberg algebra. This algebra was introduced by Dirac to develop quantum field theory. We won't explain any physics here, but instead we shall to delve somewhat deeper into the workings of Yacas.

4.20.1 The problem

Suppose we want to define special symbols $A(k)$ and $B(k)$ that we can multiply with each other or by a number, or add to each other, but not commute with each other, i.e. $A(k)*B(k) != B(k)*A(k)$. Here k is merely a label to denote that $A(1)$ and $A(2)$ are two different objects. (In physics, these are called "creation" and "annihilation" operators for "bosonic quantum fields".) Yacas already assumes that the usual multiplication operator "{*}" is commutative. Rather than trying to redefine {*}, we shall introduce a special multiplication sign "{**}" that we shall use with the objects $A(k)$ and $B(k)$; between usual numbers this would be the same as normal multiplication. The symbols $A(k)$, $B(k)$ will never be evaluated to numbers, so an expression such as {2 ** A(k1) ** B(k2) ** A(k3)} is just going to remain like that. (In physics, commuting numbers are called "classical quantities" or "c-numbers" while non-commuting objects made up of A(k) and B(k) are called "quantum quantities" or "q-numbers".) There are certain commutation relations for these symbols: the A's commute between themselves, $A(k)*A(l) = A(l)*A(k)$, and also the B's, $B(k)*B(l) = B(l)*B(k)$. However, the A's don't commute with the B's: $A(k)*B(l) - B(l)*A(k) = delta(k-l)$. Here the "{delta}" is a "classical" function (called the "Dirac $delta$-function") but we aren't going to do anything about it, just leave it symbolic.

We would like to be able to manipulate such expressions, expanding brackets, collecting similar terms and so on, while taking care to always keep the non-commuting terms in the correct order. For example, we want Yacas to automatically simplify {2**B(k1)**3**A(k2)} to {6**B(k1)**A(k2)}. Our goal is not to implement a general package to tackle complicated non-commutative operations; we merely want to teach Yacas about these two kinds of "quantum objects" called {A(k)} and {B(k)}, and we shall define one function that a physicist would need to apply to these objects. This function applied to any given expression containing A's and B's will compute something called a "vacuum expectation value", or "VEV" for short, of that expression. This function has "classical", i.e. commuting, values

and is defined as follows: VEV of a commuting number is just that number, e.g. $VEV(4) = 4$, $VEV(delta(k-l)) = delta(k-l)$; and $VEV(X*A(k)) = 0$, $VEV(B(k)*X) = 0$ where X is any expression, commutative or not. It is straightforward to compute VEV of something that contains A's and B's: one just uses the commutation relations to move all B's to the left of all A's, and then applies the definition of VEV, simply throwing out any remaining q-numbers.

4.20.2 First steps

The first thing that comes to mind when we start implementing this in Yacas is to write a rule such as

```
10 # A(_k)**B(_l) <-- B(l)**A(k)
  + delta(k-l);
```

However, this is not going to work right away. In fact this will immediately give a syntax error because Yacas doesn't know yet about the new multiplication $**$. Let's fix that: we shall define a new infix operator with the same precedence as multiplication.

```
RuleBase("**", {x,y});
Infix("**", OpPrecedence("*"));
```

Now we can use this new multiplication operator in expressions, and it doesn't evaluate to anything – exactly what we need. But we find that things don't quite work:

```
In> A(_k)**B(_l) <-- B(l)**A(k)+delta(k-l);
Out> True;
In> A(x)**B(y)
Out> B(l)**A(k)+delta(k-l);
```

Yacas doesn't grok that {delta(k)}, {A(k)} and {B(k)} are functions. This can be fixed by declaring

```
RuleBase("A", {k});
RuleBase("B", {k});
RuleBase("delta", {k});
```

Now things work as intended:

```
In> A(y)**B(z)*2
Out> 2*(B(z)**A(y)+delta(y-z));
```

4.20.3 Structure of expressions

Are we done yet? Let's try to calculate more things with our A's and B's:

```
In> A(k)*2**B(l)
Out> 2*A(k)**B(l);
In> A(x)**A(y)**B(z)
Out> A(x)**A(y)**B(z);
In> (A(x)+B(x))**2**B(y)*3
Out> 3*(A(x)+B(x))**2**B(y);
```

After we gave it slightly more complicated input, Yacas didn't fully evaluate expressions containing the new {**} operation: it didn't move constants {2} and {3} together, didn't expand brackets, and, somewhat mysteriously, it didn't apply the rule in the first line above – although it seems like it should have. Before we hurry to fix these things, let's think some more about how Yacas represents our new expressions. Let's start with the first line above:

```
In> FullForm( A(k)*2**B(l) )
(** (* 2 (A k ))(B l ))
Out> 2*A(k)**B(l);
```

What looks like {2*A(k)**B(l)} on the screen is really {(2*A(k)) ** B(l)} inside Yacas. In other words, the commutation rule didn't apply because there is no subexpression of the form {A(...)**B(...)} in this expression. It seems that we would need many rules to exhaust all ways in which the adjacent factors {A(k)} and {B(l)} might be divided between subexpressions. We run into this difficulty because Yacas represents all expressions as trees of functions and leaves the semantics to us. To Yacas, the "{*}" operator is fundamentally no different from any other function, so {(a*b)*c} and {a*(b*c)} are two basically different expressions. It would take a considerable amount of work to teach Yacas to recognize all such cases as identical. This is a design choice and it was made by the author of Yacas to achieve greater flexibility and extensibility.

A solution for this problem is not to write rules for all possible cases (there are infinitely many cases) but to systematically reduce expressions to a canonical form. "Experience has shown that" (a phrase used when we can't come up with specific arguments) symbolic manipulation of unevaluated trees is not efficient unless these trees are forced to a pattern that reflects their semantics.

We should choose a canonical form for all such expressions in a way that makes our calculations – namely, the function {VEV()} – easier. In our case, our expressions contain two kinds of ingredients: normal, commutative numbers and maybe a number of noncommuting symbols {A(k)} and {B(k)} multiplied together with the "{**}" operator. It will not be possible to divide anything by $A(k)$ or $B(k)$ – such division is undefined.

A possible canonical form for expressions with A's and B's is the following. All commutative numbers are moved to the left of the expression and grouped together as one factor; all non-commutative products are simplified to a single chain, all brackets expanded. A canonical expression should not contain any extra brackets in its non-commutative part. For example, (A(x)+B(x)*x)**B(y)*y**A(z) should be regrouped as a sum of two terms, (y)**(A(x)**(B(y))**A(z)) and (x*y)**(B(x)**(B(y))**A(z)). Here we wrote out all parentheses to show explicitly which operations are grouped. (We have chosen the grouping of non-commutative factors to go from left to right, however this does not seem to be an important choice.) On the screen this will look simply {y ** A(x) ** B(y)} and {x*y** B(x) ** B(y) ** A(z)} because we have defined the precedence of the "**" operator to be the same as that of the normal multiplication, so Yacas won't insert any more parentheses.

This canonical form will allow Yacas to apply all the usual rules on the commutative factor while cleanly separating all non-commutative parts for special treatment. Note that a commutative factor such as {2*x} will be multiplied by a single non-commutative piece with "{**}".

The basic idea behind the "canonical form" is this: we should define our evaluation rules in such a way that any expression containing {A(k)} and {B(k)} will be always automatically reduced to the canonical form after one full evaluation. All functions on our new objects will assume that the object is already in the canonical form and should return objects in the same canonical form.

4.20.4 Implementing the canonical form

Now that we have a design, let's look at some implementation issues. We would like to write evaluation rules involving the new operator "{**}" as well as the ordinary multiplications and additions involving usual numbers, so that all "classical" numbers and all "quantum" objects are grouped together separately. This should be accomplished with rules that expand brackets, exchange the bracketing order of expressions and move commuting factors to the left. For now, we shall not concern ourselves with divisions and subtractions.

First, we need to distinguish "classical" terms from "quantum" ones. For this, we shall define a predicate {IsQuantum()} recursively, as follows:

```
/* Predicate IsQuantum(): will return
   True if the expression contains A(k)
   or B(k) and False otherwise */
10 # IsQuantum(A(_x)) <-- True;
```

```
10 # IsQuantum(B(_x)) <-- True;
    /* Result of a binary operation may
      be Quantum */
20 # IsQuantum(_x + _y) <-- IsQuantum(x)
  Or IsQuantum(y);
20 # IsQuantum(+ _y) <-- IsQuantum(y);
20 # IsQuantum(_x * _y) <-- IsQuantum(x)
  Or IsQuantum(y);
20 # IsQuantum(_x ** _y) <-- IsQuantum(x)
  Or IsQuantum(y);
    /* If none of the rules apply, the
      object is not Quantum */
30 # IsQuantum(_x) <-- False;
```

Now we shall construct rules that implement reduction to the canonical form. The rules will be given precedences, so that the reduction proceeds by clearly defined steps. All rules at a given precedence benefit from all simplifications at earlier precedences.

```
/* First, replace * by ** if one of the
   factors is Quantum to guard against
   user error */
10 # (_x * _y)_(IsQuantum(x) Or
  IsQuantum(y)) <-- x ** y;
    /* Replace ** by * if neither of the
      factors is Quantum */
10 # (_x ** _y)_(Not(IsQuantum(x) Or
 IsQuantum(y))) <-- x * y;
    /* Now we are guaranteed that ** is
      used between Quantum values */
    /* Expand all brackets involving
      Quantum values */
15 # (_x + _y) ** _z <-- x ** z + y ** z;
15 # _z ** (_x + _y) <-- z ** x + z ** y;
    /* Now we are guaranteed that there are
      no brackets next to "**" */
    /* Regroup the ** multiplications
      toward the right */
20 # (_x ** _y) ** _z <-- x ** (y ** z);
    /* Move classical factors to the left:
      first, inside brackets */
30 # (x_IsQuantum ** _y)_(Not(IsQuantum(y)))
  <-- y ** x;
    /* Then, move across brackets:
      y and z are already ordered
      by the previous rule */
    /* First, if we have Q ** (C ** Q) */
35 # (x_IsQuantum ** (_y ** _z))
  _(Not(IsQuantum(y))) <-- y ** (x ** z);
    /* Second, if we have C ** (C ** Q) */
35 # (_x ** (_y ** _z))_(Not(IsQuantum(x)
  Or IsQuantum(y))) <-- (x*y) ** z;
```

After we execute this in Yacas, all expressions involving additions and multiplications are automatically reduced to the canonical form. Extending these rules to subtractions and divisions is straightforward.

4.20.5 Implementing commutation relations

But we still haven't implemented the commutation relations. It is perhaps not necessary to have commutation rules automatically applied at each evaluation. We shall define the function {OrderBA()} that will bring all B's to the left of all A's by using the commutation relation. (In physics, this is called "normal-ordering".) Again, our definition will be recursive. We shall assign it a later precedence than our quantum evaluation rules, so that our objects will always be in canonical form. We need a few more rules to implement the commutation relation and to propagate the ordering operation down the expression tree:

```
/* Commutation relation */
40 # OrderBA(A(_k) ** B(_l))
  <-- B(l)**A(k) + delta(k-l);
40 # OrderBA(A(_k) ** (B(_l) ** _x))
  <-- OrderBA(OrderBA(A(k)**B(l)) ** x);
    /* Ordering simple terms */
40 # OrderBA(_x)_(Not(IsQuantum(x))) <-- x;
40 # OrderBA(A(_k)) <-- A(k);
40 # OrderBA(B(_k)) <-- B(k);
    /* Sums of terms */
40 # OrderBA(_x + _y) <-- OrderBA(x)
  + OrderBA(y);
    /* Product of a classical and
      a quantum value */
40 # OrderBA(_x ** _y)_(Not(IsQuantum(x)))
  <-- x ** OrderBA(y);
    /* B() ** X : B is already at left,
      no need to order it */
50 # OrderBA(B(_k) ** _x)<-- B(k)
  ** OrderBA(x);
    /* A() ** X : need to order X first */
50 # OrderBA(A(_k) ** _x) <-- OrderBA(A(k)
  ** OrderBA(x));
```

These rules seem to be enough for our purposes. Note that the commutation relation is implemented by the first two rules; the first one is used by the second one which applies when interchanging factors A and B separated by brackets. This inconvenience of having to define several rules for what seems to be "one thing to do" is a consequence of tree-like structure of expressions in Yacas. It is perhaps the price we have to pay for conceptual simplicity of the design.

4.20.6 Avoiding infinite recursion

However, we quickly discover that our definitions don't work. Actually, we have run into a difficulty typical of rule-based programming:

```
In> OrderBA(A(k)**A(l))
Error on line 1 in file [CommandLine]
Line error occurred on:
>>>
Max evaluation stack depth reached.
Please use MaxEvalDepth to increase the
  stack size as needed.
```

This error message means that we have created an infinite recursion. It is easy to see that the last rule is at fault: it never stops applying itself when it operates on a term containing only A's and no B's. When encountering a term such as {A()**X}, the routine cannot determine whether {X} has already been normal-ordered or not, and it unnecessarily keeps trying to normal-order it again and again. We can circumvent this difficulty by using an auxiliary ordering function that we shall call {OrderBAlate()}. This function will operate only on terms of the form {A()**X}

and only after {X} has been ordered. It will not perform any extra simplifications but instead delegate all work to {OrderBA()}.

```
50 # OrderBA(A(_k) ** _x) <-- OrderBAlate(
  A(k) ** OrderBA(x));
55 # OrderBAlate(_x + _y) <-- OrderBAlate(
  x) + OrderBAlate(y);
55 # OrderBAlate(A(_k) ** B(_l)) <--
  OrderBA(A(k)**B(l));
55 # OrderBAlate(A(_k) ** (B(_l) ** _x))
  <-- OrderBA(A(k)**(B(l)**x));
60 # OrderBAlate(A(_k) ** _x) <-- A(k)**x;
65 # OrderBAlate(_x) <-- OrderBA(x);
```

Now {OrderBA()} works as desired.

4.20.7 Implementing VEV()

Now it is easy to define the function {VEV()}. This function should first execute the normal-ordering operation, so that all B's move to the left of A's. After an expression is normal-ordered, all of its "quantum" terms will either end with an $A(k)$ or begin with a $B(k)$, or both, and {VEV()} of those terms will return 0. The value of {VEV()} of a non-quantum term is just that term. The implementation could look like this:

```
100 # VEV(_x) <-- VEVOrd(OrderBA(x));
    /* Everything is expanded now,
      deal term by term */
100 # VEVOrd(_x + _y) <-- VEVOrd(x)
  + VEVOrd(y);
    /* Now cancel all quantum terms */
110 # VEVOrd(x_IsQuantum) <-- 0;
    /* Classical terms are left */
120 # VEVOrd(_x) <-- x;
```

To avoid infinite recursion in calling {OrderBA()}, we had to introduce an auxiliary function {VEVOrd()} that assumes its argument to be ordered.

Finally, we try some example calculations to test our rules:

```
In> OrderBA(A(x)*B(y))
Out> B(y)**A(x)+delta(x-y);
In> OrderBA(A(x)*B(y)*B(z))
Out> B(y)**B(z)**A(x)+delta(x-z)**B(y)
  +delta(x-y)**B(z);
In> VEV(A(k)*B(l))
Out> delta(k-l);
In> VEV(A(k)*B(l)*A(x)*B(y))
Out> delta(k-l)*delta(x-y);
In> VEV(A(k)*A(l)*B(x)*B(y))
Out> delta(l-y)*delta(k-x)+delta(l-x)
  *delta(k-y);
```

Things now work as expected. Yacas's {Simplify()} facilities can be used on the result of {VEV()} if it needs simplification.

CHAPTER 5

The Yacas Book of Algorithms

This book is a detailed description of the algorithms used in the Yacas system for exact symbolic and arbitrary-precision numerical computations. Very few of these algorithms are new, and most are well-known. The goal of this book is to become a compendium of all relevant issues of design and implementation of these algorithms.

5.1 Adaptive function plotting

Here we consider plotting of functions $y = f(x)$.

There are two tasks related to preparation of plots of functions: first, to produce the numbers required for a plot, and second, to draw a plot with axes, symbols, a legend, perhaps additional illustrations and so on. Here we only concern ourselves with the first task, that of preparation of the numerical data for a plot. There are many plotting programs that can read a file with numbers and plot it in any desired manner.

Generating data for plots of functions generally does not require high-precision calculations. However, we need an algorithm that can be adjusted to produce data to different levels of precision. In some particularly ill-behaved cases, a precise plot will not be possible and we would not want to waste time producing data that is too accurate for what it is worth.

A simple approach to plotting would be to divide the interval into many equal subintervals and to evaluate the function on the resulting grid. Precision of the plot can be adjusted by choosing a larger or a smaller number of points.

However, this approach is not optimal. Sometimes a function changes rapidly near one point but slowly everywhere else. For example, $f(x) = \frac{1}{x}$ changes very quickly at small x. Suppose we need to plot this function between 0 and 100. It would be wasteful to use the same subdivision interval everywhere: a finer grid is only required over a small portion of the plotting range near $x = 0$.

The adaptive plotting routine `Plot2D'adaptive()` uses a simple algorithm to select the optimal grid to approximate a function of one argument $f(x)$. The algorithm repeatedly subdivides the grid intervals near points where the existing grid does not represent the function well enough. A similar algorithm for adaptive grid refinement could be used for numerical integration. The idea is that plotting and numerical integration require the same kind of detailed knowledge about the behavior of the function.

The algorithm first splits the interval into a specified initial number of equal subintervals, and then repeatedly splits each subinterval in half until the function is well enough approximated by the resulting grid. The integer parameter {depth} gives the maximum number of binary splittings for a given initial interval; thus, at most 2^depth additional grid points will be generated. The function {Plot2D'adaptive} should return a list of pairs of points {{{x1,y1}, {x2,y2}, ...}} to be used directly for plotting.

The adaptive plotting algorithm works like this:

1. Given an interval (a, c), we split it in half, $b := (a + c)/2$ and first compute $f(x)$ at five grid points a, $a_1 := (a + b)/2$, b, $b_1 := (b + c)/2$, c.

2. If currently $depth <= 0$, return this list of five points and values because we cannot refine the grid any more.
3. Otherwise, check that the function does not oscillate too rapidly on the interval $[a, c]$. The formal criterion is that the five values are all finite and do not make a "zigzag" pattern such as $(1, 3, 2, 3, 1)$. More formally, we use the following procedure: For each three consecutive values, write "1" if the middle value is larger than the other two, or if it is smaller than the other two, or if one of them is not a number (e.g. `Infinity` or `Undefined}`. If we have at most two ones now, then we consider the change of values to be "slow enough". Otherwise it is not "slow enough". In this case we need to refine the grid; go to step 5. Otherwise, go to step 4.
4. Check that the function values are smooth enough through the interval. Smoothness is controlled by a parameter ϵ. The meaning of the parameter ϵ is the (relative) error of the numerical approximation of the integral of $f(x)$ by the grid. A good heuristic value of ϵ is 1/(the number of pixels on the screen) because it means that no pixels will be missing in the area under the graph. For this to work we need to make sure that we are actually computing the area *under* the graph; so we define $g(x) := f(x) - f[0]$ where $f[0]$ is the minimum of the values of $f(x)$ on the five grid points a, a_1, b, b_1, and c; the function $g(x)$ is nonnegative and has the minimum value 0. Then we compute two different Newton-Cotes quadratures for $\int_b^{b_1} g(x)\,\mathrm{d}x$ using these five points. (Asymmetric quadratures are chosen to avoid running into an accidental symmetry of the function; the first quadrature uses points a, a_1, b, b_1 and the second quadrature uses b, b_1, c.) If the absolute value of the difference between these quadratures is less than $epsilon$ * (value of the second quadrature), then we are done and we return the list of these five points and values.
5. Otherwise, we need to refine the grid. We compute `Plot2D'adaptive()` recursively for the two halves of the interval, that is, for $[a, b]$ and $[b, c]$. We also decrease $depth$ by 1 and multiply ϵ by 2 because we need to maintain a constant *absolute* precision and this means that the relative error for the two subintervals can be twice as large. The resulting two lists for the two subintervals are concatenated (excluding the double value at point b) and returned.

This algorithm works well if the initial number of points and the $depth$ parameter are large enough. These parameters can be adjusted to balance the available computing time and the desired level of detail in the resulting plot.

Singularities in the function are handled by the step 3. Namely, the change in the sequence a, a_1, b, b_1, c is always considered to be "too rapid" if one of these values is a non-number (e.g. `Infinity` or `Undefined`). Thus, the interval immediately adjacent to a singularity will be plotted at the highest allowed refinement level. When preparing the plotting data, the singular points are simply not printed to the data file, so that a plotting programs does not encounter any problems.

5.1.1 Newton-Cotes quadratures

The meaning of Newton-Cotes quadrature coefficients is that an integral of a function $f(x)$ is approximated by a sum

$$\int_{a_0}^{a_n} f(x)\,\mathrm{d}x \approx h \sum_{k=0}^{n} c_k f(a_k)$$

where a_k are the grid points, $h := a_1 - a_0$ is the grid step, and c_k are the quadrature coefficients. It may seem surprising, but these coefficients c_k are independent of the function $f(x)$ and can be precomputed in advance for a given grid a_k. [The quadrature coefficients do depend on the relative separations of the grid. Here we assume a uniform grid with a constant step $h = a_k - a_{k-1}$. Quadrature coefficients can also be found for non-uniform grids.]

The coefficients c_k for grids with a constant step h can be found, for example, by solving the following system of equations,

$$\sum_{k=0}^{n} c_k k^p = \frac{n^{p+1}}{p+1}$$

for $p = 0, 1, \ldots, n$. This system of equations means that the quadrature correctly gives the integrals of $p+1$ functions $f(x) = x^p$, $p = 0, 1, \ldots, n$ over the interval $(0, n)$. The solution of this system always exists and gives quadrature

coefficients as rational numbers. For example, the well-known Simpson quadrature $c_0 = 1/3, c_1 = 4/3, c_2 = 1/3$ is obtained with $n = 2$. An example of using this quadrature is the approximation

$$\int_0^2 f(x)\,\mathrm{d}x \approx (f(0) + f(2))/3 + 4/3 * f(1)$$

5.1.2 Newton-Cotes quadratures for partial intervals

In the same way it is possible to find quadratures for the integral over a subinterval rather than over the whole interval of x. In the current implementation of the adaptive plotting algorithm, two quadratures are used: the 3-point quadrature ($n = 2$) and the 4-point quadrature ($n = 3$) for the integral over the first subinterval, $\int_{a_0}^{a_1}(x, a[0], a[1])f(x)\,\mathrm{d}x$. Their coefficients are $\left(\frac{5}{12}, \frac{2}{3}, -\frac{1}{12}\right)$ and $\left(\frac{3}{8}, \frac{19}{24}, -\frac{5}{24}, \frac{1}{24}\right)$. An example of using the first of these subinterval quadratures would be the approximation

$$\int_0^2 f(x)\,\mathrm{d}x \approx \frac{5}{12}f(0) + \frac{2}{3}f(1) - \frac{1}{12}f(2).$$

These quadratures are intentionally chosen to be asymmetric to avoid an accidental cancellation when the function $f(x)$ itself is symmetric. (Otherwise the error estimate could accidentally become exactly zero.)

5.2 Surface plotting

Here we consider plotting of functions $z = f(x, y)$.

The task of surface plotting is to obtain a picture of a two-dimensional surface as if it were a solid object in three dimensions. A graphical representation of a surface is a complicated task. Sometimes it is required to use particular coordinates or projections, to colorize the surface, to remove hidden lines and so on. We shall only be concerned with the task of obtaining the data for a plot from a given function of two variables $f(x, y)$. Specialized programs can take a text file with the data and let the user interactively produce a variety of surface plots.

The currently implemented algorithm in the function `Plot3DS()` is very similar to the adaptive plotting algorithm for two-dimensional plots. A given rectangular plotting region $a_1 \le x \le a_2$, $b_1 \le y \le b_2$ is subdivided to produce an equally spaced rectangular grid of points. This is the initial grid which will be adaptively refined where necessary. The refinement algorithm will divide a given rectangle in four quarters if the available function values indicate that the function does not change smoothly enough on that rectangle.

The criterion of a "smooth enough" change is very similar to the procedure outlined in the previous section. The change is "smooth enough" if all points are finite, nonsingular values, and if the integral of the function over the rectangle is sufficiently well approximated by a certain low-order "cubature" formula.

The two-dimensional integral of the function is estimated using the following 5-point Newton-Cotes cubature:

1/12	0	1/12
0	2/3	0
1/12	0	1/12

An example of using this cubature would be the approximation

$$\int_0^1 \mathrm{d}x \int_0^1 \mathrm{d}y f(x, y) \approx \frac{f(0,0) + f(0,1) + f(1,0) + f(1,1)}{12} + \frac{2}{3}f\left(\frac{1}{2}, \frac{1}{2}\right)$$

Similarly, an 8-point cubature with zero sum is used to estimate the error:

-1/3	2/3	1/6
-1/6	-2/3	-1/2
1/2	0	1/3

This set of coefficients was intentionally chosen to be asymmetric to avoid possible exact cancellations when the function itself is symmetric.

One minor problem with adaptive surface plotting is that the resulting set of points may not correspond to a rectangular grid in the parameter space (x, y). This is because some rectangles from the initial grid will need to be bisected more times than others. So, unless adaptive refinement is disabled, the function `Plot3DS()` produces a somewhat disordered set of points. However, most surface plotting programs require that the set of data points be a rectangular grid in the parameter space. So a smoothing and interpolation procedure is necessary to convert a non-gridded set of data points ("scattered" data) to a gridded set.

5.3 Parametric plots

Currently, parametric plots are not directly implemented in Yacas. However, it is possible to use Yacas to obtain numerical data for such plots. One can then use external programs to produce actual graphics.

A two-dimensional parametric plot is a line in a two-dimensional space, defined by two equations such as $x = f(t), y = g(t)$. Two functions f, g and a range of the independent variable t, for example, $t_1 \leq t \leq t_2$, need to be specified.

Parametric plots can be used to represent plots of functions in non-Euclidean coordinates. For example, to plot the function $\rho = \cos(4\phi)^2$ in polar coordinates (ρ, ϕ), one can rewrite the Euclidean coordinates through the polar coordinates, $x = \rho\cos(\phi), y = \rho\sin(\phi)$, and use the equivalent parametric plot with ϕ as the parameter: $x = \cos(4\phi)^2\cos(\phi), y = \cos(4\phi)^2\sin(\phi)$.

Sometimes higher-dimensional parametric plots are required. A line plot in three dimensions is defined by three functions of one variable, for example, $x = f(t), y = g(t), z = h(t)$, and a range of the parameter t. A surface plot in three dimensions is defined by three functions of two variables each, for example, $x = f(u, v), y = g(u, v), z = h(u, v)$, and a rectangular domain in the (u, v) space.

The data for parametric plots can be generated separately using the same adaptive plotting algorithms as for ordinary function plots, as if all functions such as $f(t)$ or $g(u, v)$ were unrelated functions. The result would be several separate data sets for the $x, y, \ldots$ coordinates. These data sets could then be combined using an interactive plotting program.

5.4 The cost of arbitrary-precision computations

A computer algebra system absolutely needs to be able to perform computations with very large *integer* numbers. Without this capability, many symbolic computations (such as exact GCD of polynomials or exact solution of polynomial equations) would be impossible.

A different question is whether a CAS really needs to be able to evaluate, say, 10,000 digits of the value of a Bessel function of some 10,000-digit complex argument. It seems likely that no applied problem of natural sciences would need floating-point computations of special functions with such a high precision. However, arbitrary-precision computations are certainly useful in some mathematical applications; e.g. some mathematical identities can be first guessed by a floating-point computation with many digits and then proved.

Very high precision computations of special functions *might* be useful in the future. But it is already quite clear that computations with moderately high precision (say, 50 or 100 decimal digits) are useful for applied problems. For example, to obtain the leading asymptotic of an analytic function, we could expand it in series and take the first term. But we need to check that the coefficient at what we think is the leading term of the series does not vanish. This coefficient could be a certain "exact" number such as $(\cos(355) + 1)^2$. This number is "exact" in the sense that it is made of integers and elementary functions. But we cannot say *a priori* that this number is nonzero. The problem of "zero determination" (finding out whether a certain "exact" number is zero) is known to be algorithmically unsolvable if we allow transcendental functions. The only practical general approach seems to be to compute the number in question with many digits. Usually a few digits are enough, but occasionally several hundred digits are needed.

Implementing an efficient algorithm that computes 100 digits of $\sin(\frac{3}{7})$ already involves many of the issues that would also be relevant for a 10,000 digit computation. Modern algorithms allow evaluations of all elementary functions in time that is asymptotically logarithmic in the number of digits P and linear in the cost of long multiplication (usually denoted $M(P)$). Almost all special functions can be evaluated in time that is asymptotically linear in P and in $M(P)$. (However, this asymptotic cost sometimes applies only to very high precision, e.g., $P > 1000$, and different algorithms need to be implemented for calculations in lower precision.)

In yacas we strive to implement all numerical functions to arbitrary precision. All integer or rational functions return exact results, and all floating-point functions return their value with P correct decimal digits (assuming sufficient precision of the arguments). The current value of P is accessed as `Builtin'Precision'Get()` and may be changed by `Builtin'Precision'Set()`.

Implementing an arbitrary-precision floating-point computation of a function $f(x)$, such as $f(x) = \exp(x)$, typically needs the following:

- An algorithm that will compute $f(x)$ for a given value x to a user-specified precision of P (decimal) digits. Often, several algorithms must be implemented for different subdomains of the (x,:math:*P*) space.
- An estimate of the computational cost of the algorithm(s), as a function of x and P. This is needed to select the best algorithm for given x, P.
- An estimate of the round-off error. This is needed to select the "working precision" which will typically be somewhat higher than the precision of the final result.

In calculations with machine precision where the number of digits is fixed, the problem of round-off errors is quite prominent. Every arithmetic operation causes a small loss of precision; as a result, a few last digits of the final value are usually incorrect. But if we have an arbitrary precision capability, we can always increase precision by a few more digits during intermediate computations and thus eliminate all round-off error in the final result. We should, of course, take care not to increase the working precision unnecessarily, because any increase of precision means slower calculations. Taking twice as many digits as needed and hoping that the result is precise is not a good solution.

Selecting algorithms for computations is the most non-trivial part of the implementation. We want to achieve arbitrarily high precision, so we need to find either a series, or a continued fraction, or a sequence given by explicit formula, that converges to the function in a controlled way. It is not enough to use a table of precomputed values or a fixed approximation formula that has a limited precision.

In the last 30 years, the interest in arbitrary-precision computations grew and many efficient algorithms for elementary and special functions were published. Most algorithms are iterative. Almost always it is very important to know in advance how many iterations are needed for given x, P. This knowledge allows to estimate the computational cost, in terms of the required precision P and of the cost of long multiplication $M(P)$, and choose the best algorithm.

Typically all operations will fall into one of the following categories (sorted by the increasing cost):

- addition, subtraction: linear in P;
- multiplication, division, integer power, integer root: linear in $M(P)$;
- elementary functions: $\exp(x)$, $\ln(x)$, $\sin(x)$, $\arctan(x)$ etc.: $M(P)\ln(P)$ or slower by some powers of $\ln(P)$;
- transcendental functions: $erf(x)$, $\gamma(x)$ etc.: typically $PM(P)$ or slower.

The cost of long multiplication $M(P)$ is between $O(P^2)$ for low precision and $O(P\ln(P))$ for very high precision. In some cases, a different algorithm should be chosen if the precision is high enough to allow $M(P)$ faster than $O(P^2)$.

Some algorithms also need storage space (e.g. an efficient algorithm for summation of the Taylor series uses $O(\ln(P))$ temporary P-digit numbers).

Below we shall normally denote by P the required number of decimal digits. The formulae frequently contain conspicuous factors of $\ln(10)$, so it will be clear how to obtain analogous expressions for another base. (Most implementations use a binary base rather than a decimal base since it is more convenient for many calculations.)

5.5 Estimating convergence of a series

Analyzing convergence of a power series is usually not difficult. Here is a worked-out example of how we could estimate the required number of terms in the power series

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!} + O(x^{n+1})$$

if we need P decimal digits of precision in the result. To be specific, assume that $|x| < 1$. (A similar calculation can be done for any other bound on x.)

Suppose we truncate the series after n-th term and the series converges "well enough" after that term. Then the error will be approximately equal to the first term we dropped. (This is what we really mean by "converges well enough" and this will generally be the case in all applications, because we would not want to use a series that does not converge well enough.)

The term we dropped is $\frac{x^{n+1}}{(n+1)!}$. To estimate $n!$ for large n, one can use the inequality

$$e^{e-1} * (n/e)^n < n! < (n/e)^{n+1}$$

(valid for all $n \geq 47$) which provides tight bounds for the growth of the factorial, or a weaker inequality which is somewhat easier to use,

$$(n/e)^n < n! < ((n+1)/e)^{n+1}$$

(valid for all $n \geq 6$). The latter inequality is sufficient for most purposes.

If we use the upper bound on $n!$ from this estimate, we find that the term we dropped is bounded by

$$x^{n+1}/(n+1)! < (e/(n+2))^{n+2}.$$

We need this number to be smaller than 10^{-P}. This leads to an inequality

$$(e/(n+2))^(n+2) < 10^{-P},$$

which we now need to solve for n. The left hand side decreases with growing n. So it is clear that the inequality will hold for large enough n, say for $n \geq n_0$ where n_0 is an unknown (integer) value. We can take a logarithm of both sides, replace n with n_0 and obtain the following equation for n_0:

$$(n_0 + 2)\ln((n_0 + 2)/e) = P * \ln(10).$$

This equation cannot be solved exactly in terms of elementary functions; this is a typical situation in such estimates. However, we do not really need a very precise solution for n_0; all we need is an estimate of its integer part. This is also a typical situation. It is acceptable if our approximate value of n_0 comes out a couple of units higher than necessary, because a couple of extra terms of the Taylor series will not significantly slow down the algorithm (but it is important that we do not underestimate n_0). Finally, we are mostly interested in having a good enough answer for large values of P.

We can try to guess the result. The largest term on the LHS grows as $n_0 \ln(n_0)$ and it should be approximately equal to $P\ln(10)$; but $\ln(n_0)$ grows very slowly, so this gives us a hint that n_0 is proportional to $P\ln(10)$. As a first try, we set $n_0 = P\ln(10) - 2$ and compare the RHS with the LHS; we find that we have overshot by a factor $\ln(P) - 1 + \ln(\ln(10))$, which is not a large factor. We can now compensate and divide n_0 by this factor, so our second try is

$$n0 = (P\ln(10))/(\ln(P) - 1 + \ln(\ln(10))) - 2.$$

(This approximation procedure is equivalent to solving the equation

$$x = (P * \ln(10))/(\ln(x) - 1)$$

by direct iteration, starting from $x = P\ln(10)$.) If we substitute our second try for n_0 into the equation, we shall find that we undershot a little bit (i.e. the LHS is a little smaller than the RHS), but our n_0 is now smaller than it should be by a quantity that is smaller than 1 for large enough P. So we should stop at this point and simply add 1 to this approximate answer. We should also replace $\ln(\ln(10)) - 1$ by 0 for simplicity (this is safe because it will slightly increase n_0.)

Our final result is that it is enough to take

$$n = (P * \ln(10))/\ln(P) - 1$$

terms in the Taylor series to compute $\exp(x)$ for $|x| < 1$ to P decimal digits. (Of course, if x is much smaller than 1, many fewer terms will suffice.)

5.6 Estimating the round-off error

5.6.1 Unavoidable round-off errors

As the required precision P grows, an arbitrary-precision algorithm will need more iterations or more terms of the series. So the round-off error introduced by every floating-point operation will increase. When doing arbitrary-precision computations, we can always perform all calculations with a few more digits and compensate for round-off error. It is however imperative to know in advance how many more digits we need to take for our "working precision". We should also take that increase into account when estimating the total cost of the method. (In most cases this increase is small.)

Here is a simple estimate of the normal round-off error in a computation of n terms of a power series. Suppose that the sum of the series is of order 1, that the terms monotonically decrease in magnitude, and that adding one term requires two multiplications and one addition. If all calculations are performed with absolute precision $\epsilon = 10^{-P}$, then the total accumulated round-off error is $3n\epsilon$. If the relative error is $3n\epsilon$, it means that our answer is something like $a * (1 + 3n\epsilon)$ where a is the correct answer. We can see that out of the total P digits of this answer, only the first k decimal digits are correct, where $k = -\ln(3n\epsilon)/\ln(10)$. In other words, we have lost

$$P - k = \ln(3n)/\ln(10)$$

digits because of accumulated round-off error. So we found that we need $\ln(3 * n)/\ln(10)$ extra decimal digits to compensate for this round-off error.

This estimate assumes several things about the series (basically, that the series is "well-behaved"). These assumptions must be verified in each particular case. For example, if the series begins with some large terms but converges to a very small value, this estimate is wrong (see the next subsection).

In the previous exercise we found the number of terms n for $\exp(x)$. So now we know how many extra digits of working precision we need for this particular case.

Below we shall have to perform similar estimates of the required number of terms and of the accumulated round-off error in our analysis of the algorithms.

5.6.2 Catastrophic round-off error

Sometimes the round-off error of a particular method of computation becomes so large that the method becomes highly inefficient.

Consider the computation of $\sin(x)$ by the truncated Taylor series

$$\sin(x) \approx \sum_{k=0}^{N-1} (-1)^k x^{2k+1}/(2k+1)!),$$

when x is large. We know that this series converges for all x, no matter how large. Assume that $x = 10^M$ with $M \geq 1$, and that we need P decimal digits of precision in the result.

First, we determine the necessary number of terms N. The magnitude of the sum is never larger than 1. Therefore we need the N-th term of the series to be smaller than 10^{-P}. The inequality is $(2N+1)! > 10^{P+M(2N+1)}$. We obtain that $2N + 2 > e10^M$ is a necessary condition, and if P is large, we find approximately

$$2N + 2 \approx ((P - M)\ln(10))/(\ln(P - M) - 1 - M\ln(10)).$$

However, taking enough terms does not yet guarantee a good result. The terms of the series grow at first and then start to decrease. The sum of these terms is, however, small. Therefore there is some cancellation and we need to increase the working precision to avoid the round-off. Let us estimate the required working precision.

We need to find the magnitude of the largest term of the series. The ratio of the next term to the previous term is $x/(2k(2k+1))$ and therefore the maximum will be when this ratio becomes equal to 1, i.e. for $2k \approx \sqrt{x}$. Therefore the largest term is of order $x^{\sqrt{x}}/\sqrt{x}!$ and so we need about $M/2\sqrt{x}$ decimal digits before the decimal point to represent this term. But we also need to keep at least P digits after the decimal point, or else the round-off error will erase the significant digits of the result. In addition, we will have unavoidable round-off error due to $O(P)$ arithmetic operations. So we should increase precision again by $P + \ln(P)/\ln(10)$ digits plus a few guard digits.

As an example, to compute $\sin(10)$ to $P = 50$ decimal digits with this method, we need a working precision of about 60 digits, while to compute $\sin(10000)$ we need to work with about 260 digits. This shows how inefficient the Taylor series for $\sin(x)$ becomes for large arguments x. A simple transformation $x = 2\pi n + x'$ would have reduced x to at most 7, and the unnecessary computations with 260 digits would be avoided. The main cause of this inefficiency is that we have to add and subtract extremely large numbers to get a relatively small result of order 1.

We find that the method of Taylor series for $\sin(x)$ at large x is highly inefficient because of round-off error and should be complemented by other methods. This situation seems to be typical for Taylor series.

5.7 Basic arbitrary-precision arithmetic

Yacas uses an internal math library (the `yacasnumbers` library) which comes with the source code. This reduces the dependencies of the yacas system and improves portability. The internal math library is simple and does not necessarily use the most optimal algorithms.

If P is the number of digits of precision, then multiplication and division take $M(P) = O(P^2)$ operations in the internal math. (Of course, multiplication and division by a short integer takes time linear in P.) Much faster algorithms (Karatsuba, Toom-Cook, FFT multiplication, Newton-Raphson division etc.) are implemented in {gmp}, {CLN} and some other libraries. The asymptotic cost of multiplication for very large precision is $M(P) \approx O(P^{1.6})$ for the Karatsuba method and $M(P) = O(P\ln(P)\ln(\ln(P)))$ for the FFT method. In the estimates of computation cost in this book we shall assume that $M(P)$ is at least linear in P and maybe a bit slower.

The costs of multiplication may be different in various arbitrary-precision arithmetic libraries and on different computer platforms. As a rough guide, one can assume that the straightforward $O(P^2)$ multiplication is good until 100-200 decimal digits, the asymptotically fastest method of FFT multiplication is good at the precision of about 5,000 or more decimal digits, and the Karatsuba multiplication is best in the middle range.

Warning: calculations with internal yacas math using precision exceeding 10,000 digits are currently impractically slow.

In some algorithms it is necessary to compute the integer parts of expressions such as $a\ln(b)/\ln(10)$ or $a\ln(10)/\ln(2)$, where a, b are short integers of order $O(P)$. Such expressions are frequently needed to estimate the number of terms in the Taylor series or similar parameters of the algorithms. In these cases, it is important that the result is not underestimated. However, it would be wasteful to compute $1000\ln(10)/\ln(2)$ in great precision only to discard most of that information by taking the integer part of that number. It is more efficient to approximate such constants from above by short rational numbers, for example, $\ln(10)/\ln(2) < 28738/8651$ and $\ln(2) < 7050/10171$. The

error of such an approximation will be small enough for practical purposes. The function `BracketRational()` can be used to find optimal rational approximations.

The function `IntLog()` (see below) efficiently computes the integer part of a logarithm (for an integer base, not a natural logarithm). If more precision is desired in calculating $\ln(a)/\ln(b)$ for integer a, b, one can compute $IntLog(a^k, b)$ for some integer k and then divide by k.

5.7.1 How many digits of $\sin(\exp(\exp(1000)))$ do we need?

Arbitrary-precision math is not omnipotent against overflow. Consider the problem of representing very large numbers such as $x = \exp(\exp(1000))$. Suppose we need a floating-point representation of the number x with P decimal digits of precision. In other words, we need to express $x \approx M10^E$, where the mantissa $1 < M < 10$ is a floating-point number and the exponent E is an integer, chosen so that the relative precision is 10^{-P}. How much effort is needed to find M and E?

The exponent E is easy to obtain:

$$E = \lfloor \ln(x)/\ln(10) \rfloor = \lfloor \exp(1000)/\ln(10) \rfloor \approx 8.55 * 10^{433}.$$

To compute the integer part $\lfloor y \rfloor$ of a number y exactly, we need to approximate y with at least $\ln(y)/\ln(10)$ floating-point digits. In our example, we find that we need 434 decimal digits to represent E.

Once we found E, we can write $x = 10^{E+m}$ where $m = \exp(1000)/\ln(10) - E$ is a floating-point number, $0 < m < 1$. Then $M = 10^m$. To find M with P (decimal) digits, we need m with also at least P digits. Therefore, we actually need to evaluate $\exp(1000)/\ln(10)$ with $434 + P$ decimal digits before we can find P digits of the mantissa of x. We ran into a perhaps surprising situation: one needs a high-precision calculation even to find the first digit of x, because it is necessary to find the exponent E exactly as an integer, and E is a rather large integer. A normal double-precision numerical calculation would give an overflow error at this point.

Suppose we have found the number $x = \exp(\exp(1000))$ with some precision. What about finding $\sin(x)$? Now, this is extremely difficult, because to find even the first digit of $\sin(x)$ we have to evaluate x with *absolute* error of at most 0.5. We know, however, that the number x has approximately $10^4 34$ digits *before* the decimal point. Therefore, we would need to calculate x with at least that many digits. Computations with $10^4 34$ digits is clearly far beyond the capability of modern computers. It seems unlikely that even the sign of $\sin(\exp(\exp(1000)))$ will be obtained in the near future [1].

Suppose that N is the largest integer that our arbitrary-precision facility can reasonably handle. (For yacas internal math library, N is about $10^1 0000$.) Then it follows that numbers x of order 10^N can be calculated with at most one (1) digit of floating-point precision, while larger numbers cannot be calculated with any precision at all.

It seems that very large numbers can be obtained in practice only through exponentiation or powers. It is unlikely that such numbers will arise from sums or products of reasonably-sized numbers in some formula [2].

If numbers larger than 10^N are created only by exponentiation operations, then special exponential notation could be used to represent them. For example, a very large number z could be stored and manipulated as an unevaluated exponential $z = \exp(M10^E)$ where $M \geq 1$ is a floating-point number with P digits of mantissa and E is an integer, $\ln(N) < E < N$. Let us call such objects "exponentially large numbers" or "exp-numbers" for short.

In practice, we should decide on a threshold value N and promote a number to an exp-number when its logarithm exceeds N.

Note that an exp-number z might be positive or negative, e.g. $z = -\exp(M10^E)$.

[1] It seems even less likely that the sign of $\sin(\exp(\exp(1000)))$ would be of any use to anybody even if it could be computed.

[2] A factorial function can produce rapidly growing results, but exact factorials $n!$ for large n are well represented by the Stirling formula which involves powers and exponentials. For example, suppose a program operates with numbers x of size N or smaller; a number such as 10^N can be obtained only by multiplying $O(N)$ numbers x together. But since N is the largest representable number, it is certainly not feasible to perform $O(N)$ sequential operations on a computer. However, it is feasible to obtain N-th power of a small number, since it requires only $O(Ln(N))$ operations.

Arithmetic operations can be applied to the exp-numbers. However, exp-large arithmetic is of limited use because of an almost certainly critical loss of precision. The power and logarithm operations can be meaningfully performed on exp-numbers z. For example, if $z = \exp(M10^E)$ and p is a normal floating-point number, then $z^p = \exp(pM10^E)$ and $\ln(z) = M10^E$. We can also multiply or divide two exp-numbers. But it makes no sense to multiply an exp-number z by a normal number because we cannot represent the difference between z and say $2.52 * z$. Similarly, adding z to anything else would result in a total underflow, since we do not actually know a single digit of the decimal representation of z. So if z_1 and z_2 are exp-numbers, then $z_1 + z_2$ is simply equal to either $z1$ or $z2$ depending on which of them is larger.

We find that an exp-number z acts as an effective "infinity" compared with normal numbers. But exp-numbers cannot be used as a device for computing limits: the unavoidable underflow will almost certainly produce wrong results. For example, trying to verify

$$\lim_{x \to 0} \frac{\exp(x) - 1}{x} = 1$$

by substituting $x = 1/z$ with some exp-number z gives 0 instead of 1.

Taking a logarithm of an exp-number brings it back to the realm of normal, representable numbers. However, taking an exponential of an exp-number results in a number which is not representable even as an exp-number. This is because an exp-number z needs to have its exponent E represented exactly as an integer, but $\exp(z)$ has an exponent of order $O(z)$ which is not a representable number. The monstrous number $\exp(z)$ could be only written as $\exp(\exp(M10^E))$, i.e. as a "doubly exponentially large" number, or "2-exp-number" for short. Thus we obtain a hierarchy of iterated exp-numbers. Each layer is "unrepresentably larger" than the previous one.

The same considerations apply to very small numbers of the order 10^{-N} or smaller. Such numbers can be manipulated as "exponentially small numbers", i.e. expressions of the form $Exp(-M * 10^E)$ with floating-point mantissa $M \geq 1$ and integer E satisfying $\ln(N) < E < N$. Exponentially small numbers act as an effective zero compared with normal numbers.

Taking a logarithm of an exp-small number makes it again a normal representable number. However, taking an exponential of an exp-small number produces 1 because of underflow. To obtain a "doubly exponentially small" number, we need to take a reciprocal of a doubly exponentially large number, or take the exponent of an exponentially large negative power. In other words, $\exp(-M10^E)$ is exp-small, while $\exp(-\exp(M10^E))$ is 2-exp-small.

The practical significance of exp-numbers is rather limited. We cannot obtain even a single significant digit of an exp-number. A "computation" with exp-numbers is essentially a floating-point computation with logarithms of these exp-numbers. A practical problem that needs numbers of this magnitude can probably be restated in terms of more manageable logarithms of such numbers. In practice, exp-numbers could be useful not as a means to get a numerical answer, but as a warning sign of critical overflow or underflow [3].

5.8 Sparse representations

5.8.1 The sparse tree data structure

Yacas has a sparse tree object for use as a storage for storing (key,value) pairs for which the following properties hold:

- `(key, value1) + (key, value2) = (key, value1+value2)`
- `(key1, value1) * (key2, value2) = (key1+key2, value1*value2)`

The last is optional. For multivariate polynomials (described elsewhere) both hold, but for matrices, only the addition property holds. The function {MultiplyAddSparseTrees} (described below) should not be used in these cases.

[3] Yacas currently does not implement exp-numbers or any other guards against overflow and underflow. If a decimal exponential becomes too large, an incorrect answer may result.

5.8.2 Internal structure

A key is defined to be a list of integer numbers ($ n[1] $, ..., $ n[m] $). Thus for a two-dimensional key, one item in the sparse tree database could be reflected as the (key,value) pair { {{1,2},3} }, which states that element {(1,2)} has value {3}. (Note: this is not the way it is stored in the database!).

The storage is recursive. The sparse tree begins with a list of objects { {n1,tree1} } for values of {n1} for the first item in the key. The {tree1} part then contains a sub-tree for all the items in the database for which the value of the first item in the key is {n1}.

The above single element could be created with:

```
In> r:=CreateSparseTree({1,2},3)
Out> {{1,{{2,3}}}};
```

{CreateSparseTree} makes a database with exactly one item. Items can now be obtained from the sparse tree with {SparseTreeGet}.:

```
In> SparseTreeGet({1,2},r)
Out> 3;
In> SparseTreeGet({1,3},r)
Out> 0;
```

And values can also be set or changed:

```
In> SparseTreeSet({1,2},r,Current+5)
Out> 8;
In> r
Out> {{1,{{2,8}}}};
In> SparseTreeSet({1,3},r,Current+5)
Out> 5;
In> r
Out> {{1,{{3,5},{2,8}}}};
```

The variable {Current} represents the current value, and can be used to determine the new value. {SparseTreeSet} destructively modifies the original, and returns the new value. If the key pair was not found, it is added to the tree.

The sparse tree can be traversed, one element at a time, with {SparseTreeScan}:

```
In> SparseTreeScan(Hold({{k,v},Echo({k,v})}),2,r)
{1,3} 5
{1,2} 8
```

An example of the use of this function could be multiplying a sparse matrix with a sparse vector, where the entire matrix can be scanned with {SparseTreeScan}, and each non-zero matrix element $ A[i][j] $ can then be multiplied with a vector element $ v[j] $, and the result added to a sparse vector $ w[i] $, using the {SparseTreeGet} and {SparseTreeSet} functions. Multiplying two sparse matrices would require two nested calls to {SparseTreeScan} to multiply every item from one matrix with an element from the other, and add it to the appropriate element in the resulting sparse matrix.

When the matrix elements $ A[i][j] $ are defined by a function $ f(i,j) $ (which can be considered a dense representation), and it needs to be multiplied with a sparse vector $ v[j] $, it is better to iterate over the sparse vector $ v[j] $. Representation defines the most efficient algorithm to use in this case.

The API to sparse trees is:

- {CreateSparseTree(coefs,fact)} - Create a sparse tree with one monomial, where 'coefs' is the key, and 'fact' the value. 'coefs' should be a list of integers.
- {SparseTreeMap(op,depth,tree)} - Walk over the sparse tree, one element at a time, and apply the function "op" on the arguments (key,value). The 'value' in the tree is replaced by the value returned by the {op} function.

'depth' signifies the dimension of the tree (number of indices in the key).

- {SparseTreeScan(op,depth,tree)} - Same as SparseTreeMap, but without changing elements.
- {AddSparseTrees(depth,x,y)}, {MultiplyAddSparseTrees(depth,x,y,coefs,fact)} - Add sparse tree 'y' to sparse tree 'x', destructively. in the {MultiplyAdd} case, the monomials are treated as if they were multiplied by a monomial with coefficients with the (key,value) pair (coefs,fact). 'depth' signifies the dimension of the tree (number of indices in the key).
- {SparseTreeGet(key,tree)} - return value stored for key in the tree.
- {SparseTreeSet(key,tree,newvalue)} - change the value stored for the key to newvalue. If the key was not found then {newvalue} is stored as a new item. The variable {Current} is set to the old value (or zero if the key didn't exist in the tree) before evaluating {newvalue}.

5.8.3 Implementation of multivariate polynomials

This section describes the implementation of multivariate polynomials in Yacas.

Concepts and ideas are taken from the books [Davenport et al. 1989] and [von zur Gathen et al. 1999].

Definitions

The following definitions define multivariate polynomials, and the functions defined on them that are of interest for using such multivariates.

A term is an object which can be written as

$$c * x[1]^n[1] * x[2]^n[2] * ... * x[m]^n[m]$$

for $ m $ variables ($ x[1] $, ..., $ x[m] $). The numbers $ n[m] $ are integers. $ c $ is called a coefficient, and $x[1]^n[1] * x[2]^n[2] * ... * x[m]^n[m]$ a monomial.

A multivariate polynomial is taken to be a sum over terms.

We write $ c[a]*x^a $ for a term, where $ a $ is a list of powers for the monomial, and $ c[a] $ the coefficient of the term.

It is useful to define an ordering of monomials, to be able to determine a canonical form of a multivariate.

For the currently implemented code the lexicographic order has been chosen:

- first an ordering of variables is chosen, ($ x[1] $, ..., $ x[m] $)
- for the exponents of a monomial, $ a $ = ($ a[1] $, ..., $ a[m] $) the lexicographic order first looks at the first exponent, $ a[1] $, to determine which of the two monomials comes first in the multivariate. If the two exponents are the same, the next exponent is considered.

This method is called lexicographic because it is similar to the way words are ordered in a usual dictionary.

For all algorithms (including division) there is some freedom in the ordering of monomials. One interesting advantage of the lexicographic order is that it can be implemented with a recursive data structure, where the first variable, $ x[1] $ can be treated as the main variable, thus presenting it as a univariate polynomial in $ x[1] $ with all its terms grouped together.

Other orderings can be used, by re-implementing a part of the code dealing with multivariate polynomials, and then selecting the new code to be used as a driver, as will be described later on.

Given the above ordering, the following definitions can be stated:

For a non-zero multivariate polynomial

$$ f = Sum(a,a[max],a[min],c[a]*x^a) $$

with a monomial order:

- 1. $ c[a]*x^a $ is a term of the multivariate.
- 1. the multidegree of $ f $ is $ mdeg(f) := a[max] $.
- 1. the leading coefficient of $ f $ is $ lc(f):=c[mdeg(f)] $, for the first term with non-zero coefficient.
- 1. the leading monomial of $ f $ is $ lm(f):=x^mdeg(f) $.
- 1. the leading term of $ f $ is $ lt(f):=lc(f)*lm(f) $.

The above define access to the leading monomial, which is used for divisions, gcd calculations and the like. Thus an implementation needs be able to determine { {mdeg(f),lc(f)} } . Note the similarity with the (key,value) pairs described in the sparse tree section. $ mdeg(f) $ can be thought of as a 'key', and $ lc(f) $ as a 'value'.

The multicontent, $ multicont(f) $, is defined to be a term that divides all the terms in $ f $, and is the term described by ($ Min(a) $, $ Gcd(c) $), with $ Gcd(c) $ the GCD of all the coefficients, and $ Min(a) $ the lowest exponents for each variable, occurring in $ f $ for which $ c $ is non-zero.

The multiprimitive part is then defined as $ pp(f):=f$/$multicont(f) $.

For a multivariate polynomial, the obvious addition and (distributive) multiplication rules hold

- `(a+b) + (c+d) := a+b+c+d`
- `a*(b+c) := (a*b)+(a*c)`

These are supported in the Yacas system through a multiply-add operation: $$ muadd(f,t,g) := f+t*g $$. This allows for both adding two polynomials ($ t:=1 $), or multiplication of two polynomials by scanning one polynomial, and multiplying each term of the scanned polynomial with the other polynomial, and adding the result to the polynomial that will be returned. Thus there should be an efficient {muadd} operation in the system.

Representation

For the representation of polynomials, on computers it is natural to do this in an array: ($a[1]$, $a[2]$, ..., $a[n]$) for a univariate polynomial, and the equivalent for multivariates. This is called a dense representation, because all the coefficients are stored, even if they are zero. Computers are efficient at dealing with arrays. However, in the case of multivariate polynomials, arrays can become rather large, requiring a lot of storage and processing power even to add two such polynomials. For instance, $ x^200*y^100*z^300+1 $ could take 6000000 places in an array for the coefficients. Of course variables could be substituted for the single factors, $ p:=x^200 $ etc., but it requires an additional ad hoc step.

An alternative is to store only the terms for which the coefficients are non-zero. This adds a little overhead to polynomials that could efficiently be stored in a dense representation, but it is still little memory, whereas large sparse polynomials are stored in acceptable memory too. It is of importance to still be able to add, multiply divide and get the leading term of a multivariate polynomial, when the polynomial is stored in a sparse representation.

For the representation, the data structure containing the {(exponents,coefficient)} pair can be viewed as a database holding {(key,value)} pairs, where the list of exponents is the key, and the coefficient of the term is the value stored for that key. Thus, for a variable set {{x,y}} the list {{{1,2},3}} represents $ 3*x*y^2 $.

Yacas stores multivariates internally as {MultiNomial (vars, terms)}, where {vars} is the ordered list of variables, and terms some object storing all the {(key, value)} pairs representing the terms. Note we keep the storage vague: the {terms} placeholder is implemented by other code, as a database of terms. The specific representation can be configured at startup (this is described in more detail below).

For the current version, Yacas uses the 'sparse tree' representation, which is a recursive sparse representation. For example, for a variable set {{x,y,z}}, the 'terms' object contains a list of objects of form {{deg,terms}}, one for each

degree {deg} for the variable 'x' occurring in the polynomial. The 'terms' part of this object is then a sub-sparse tree for the variables {{y,z}}.

An explicit example:

```
In> MM(3*x^2+y)
Out> MultiNomial({x,y},{{2,{{0,3}}},{0,{{1,1}, {0,0}}}});
```

The first item in the main list is {{2,{{0,3}}}}, which states that there is a term of the form $ x^2*y^0*3 $. The second item states that there are two terms, $ x^0*y^1*1 $ and $ x^0*y^0*0 = 0 $.

This representation is sparse:

```
In> r:=MM(x^1000+x)
Out> MultiNomial({x},{{1000,1},{1,1}});
```

and allows for easy multiplication:

```
In> r*r
Out> MultiNomial({x},{{2000,1},{1001,2},{2,1},{0,0}});
In> NormalForm(%)
Out> x^2000+2*x^1001+x^2;
```

Internal code organization

The implementation of multivariates can be divided in three levels.

At the top level are the routines callable by the user or the rest of the system: MultiDegree, MultiDivide, MultiGcd, Groebner, etc. In general, this is the level implementing the operations actually desired.

The middle level does the book-keeping of the {MultiNomial(vars,terms)} expressions, using the functionality offered by the lowest level.

For the current system, the middle level is in {multivar.rep/ sparsenomial.ys}, and it uses the sparse tree representation implemented in {sparsetree.ys}.

The middle level is called the 'driver', and can be changed, or re-implemented if necessary. For instance, in case calculations need to be done for which dense representations are actually acceptable, one could write C++ implementing above-mentioned database structure, and then write a middle-level driver using the code. The driver can then be selected at startup. In the file 'yacasinit.ys' the default driver is chosen, but this can be overridden in the {.yacasrc} file or some file that is loaded, or at the command line, as long as it is done before the multivariates module is loaded (which loads the selected driver). Driver selection is as simple as setting a global variable to contain a file name of the file implementing the driver:

```
Set(MultiNomialDriver,
  "multivar.rep/sparsenomial.ys");
```

where "multivar.rep/sparsenomial.ys" is the file implementing the driver (this is also the default driver, so the above command would not change any thing).

The choice was made for static configuration of the driver before the system starts up because it is expected that there will in general be one best way of doing it, given a certain system with a certain set of libraries installed on the operating system, and for a specific version of Yacas. The best version can then be selected at start up, as a configuration step. The advantage of static selection is that no overhead is imposed: there is no performance penalty for the abstraction layers between the three levels.

Driver interface

The driver should implement the following interface:

- {CreateTerm(vars,{exp,coef})} - create a multivariate polynomial with one term, in the variables defined in 'var', with the (key,value) pair (coefs,fact)
- {MultiNomialAdd(multi1, multi2) } - add two multivars, and (possibly) destructively modify multi1 to containthe result: [multi1 := multi1 + multi2; multi1;];
- {MultiNomialMultiplyAdd(multi1, multi2,exp,coef)} - add two multivars, and (possibly) destructively modify multi1 to contain the result. multi2 is considered multiplied by a term represented by the (key,value) pair (exp,coef). [multi1 := multi1 + term * multi2; multi1;];
- {MultiNomialNegate(multi)} - negate a multivar, returning -multi, and destructively changing the original. [multi := - multi; multi1;];
- {MultiNomialMultiply(multi1,multi2)} - Multiply two multivars, and (possibly) destructively modify multi1 to contain the result, returning the result: [multi1 := multi1 * multi2; multi1;];
- {NormalForm(multi)} - convert MultiNomial to normal form (as would be typed in be the user). This is part of the driver because the driver might be able to do this more efficiently than code above it which can use ScanMultiNomial.
- {MultiLeadingTerm(multi)} - return the (key,value) pair (mdeg(f),lc(f)) representing the leading term. This is all the information needed about the leading term, and thus the leading coefficient and multidegree can be extracted from it.
- {MultiDropLeadingZeroes(multi)} - remove leading terms with zero factors.
- {MultiTermLess(x,y)} - for two (key,value) pairs, return `True` if $x<y$, where the operation {<} is the one used for the representation, and `False` otherwise.
- {ScanMultiNomial(op,multi)} - traverse all the terms of the multivariate, applying the function 'op' to each (key,value) pair (exp,coef). The monomials are traversed in the ordering defined by MultiTermLess. 'op' should be a function accepting two arguments.
- {MultiZero(multi)} - return `True` if the multivariate is zero (all coefficients are zero), `False` otherwise.

5.9 Integration

Integration can be performed by the function {Integrate}, which has two calling conventions:

- Integrate(variable) expression
- Integrate(variable, from, to) expression

Integrate can have its own set of rules for specific integrals, which might return a correct answer immediately. Alternatively, it calls the function AntiDeriv, to see if the anti-derivative can be determined for the integral requested. If this is the case, the anti-derivative is used to compose the output.

If the integration algorithm cannot perform the integral, the expression is returned unsimplified.

5.9.1 The integration algorithm

This section describes the steps taken in doing integration.

General structure

The integration starts at the function {Integrate}, but the task is delegated to other functions, one after the other. Each function can deem the integral unsolvable, and thus return the integral unevaluated. These different functions offer hooks for adding new types of integrals to be handled.

Expression clean-up

Integration starts by first cleaning up the expression, by calling {TrigSimpCombine} to simplify expressions containing multiplications of trigonometric functions into additions of trigonometric functions (for which the integration rules are trivial), and then passing the result to {Simplify}.

Generalized integration rules

For the function {AntiDeriv}, which is responsible for finding the anti-derivative of a function, the code splits up expressions according to the additive properties of integration, eg. integration of $ a+b $ is the same as integrating $ a $ + integrating $ b $.

- Polynomials which can be expressed as univariate polynomials in the variable to be integrated over are handled by one integration rule.
- Expressions of the form $ p*f(x) $, where $ p $ represents a univariate polynomial, and $ f(x) $ an integrable function, are handled by a special integration rule. This transformation rule has to be designed carefully not to invoke infinite recursion.
- Rational functions, $ f(x)/g(x) $ with both $ f(x) $ and $ g(x) $ univariate polynomials, is handled separately also, using partial fraction expansion to reduce rational function to a sum of simpler expressions.

Integration tables

For elementary functions, Yacas uses integration tables. For instance, the fact that the anti-derivative of $ Cos(x) $ is $ Sin(x) $ is declared in an integration table.

For the purpose of setting up the integration table, a few declaration functions have been defined, which use some generalized pattern matchers to be more flexible in recognizing expressions that are integrable.

Integrating simple functions of a variable

For functions like $ Sin(x) $ the anti-derivative can be declared with the function {IntFunc}.

The calling sequence for {IntFunc} is:

```
IntFunc(variable,pattern,antiderivative)
```

For instance, for the function $ Cos(x) $ there is a declaration:

```
IntFunc(x,Cos(_x),Sin(x));
```

The fact that the second argument is a pattern means that each occurrence of the variable to be matched should be referred to as {_x}, as in the example above.

IntFunc generalizes the integration implicitly, in that it will set up the system to actually recognize expressions of the form $ Cos(a*x+b) $, and return $ Sin(a*x+b)/a $ automatically. This means that the variables $ a $ and $ b $ are reserved, and can not be used in the pattern. Also, the variable used (in this case, {_x} is actually matched to the expression passed in to the function, and the variable {var} is the real variable being integrated over. To clarify: suppose the user wants to integrate $ Cos(c*y+d) $ over $ y $, then the following variables are set:

- {a} = c
- {b} = d
- {x} = $a*y+b$
- {var} = x

When functions are multiplied by constants, that situation is handled by the integration rule that can deal with univariate polynomials multiplied by functions, as a constant is a polynomial of degree zero.

Integrating functions containing expressions of the form $ a*x^2+b $

There are numerous expressions containing sub-expressions of the form $ a*x^2+b $ which can easily be integrated.

The general form for declaring anti-derivatives for such expressions is:

```
IntPureSquare(variable, pattern, sign2, sign0, antiderivative)
```

Here {IntPureSquare} uses {MatchPureSquared} to match the expression.

The expression is searched for the pattern, where the variable can match to a sub-expression of the form $ a*x^2+b $, and for which both $ a $ and $ b $ are numbers and $ a*sign2>0 $ and $ b*sign0>0 $.

As an example:

```
IntPureSquare(x,num_IsFreeOf(var)/(_x),
  1,1,(num/(a*Sqrt(b/a)))*
  ArcTan(var/Sqrt(b/a)));
```

declares that the anti-derivative of $ c/(a*x^2+b) $ is $$ (c/(a*Sqrt(b/a)))*ArcTan(x/Sqrt(b/a)) $$, if both $ a $ and $ b $ are positive numbers.

5.10 Transforms

Currently the only tranform defined is {LaplaceTransform}, which has the calling convention:

```
LaplaceTransform(var1,var2,func)
```

It has been setup much like the integration algorithm. If the transformation algorithm cannot perform the transform, the expression (in theory) is returned unsimplified. Some cases may still erroneously return {Undefined} or {Infinity}.

5.10.1 The {LaplaceTransform} algorithm

This section describes the steps taken in doing a Laplace transform.

General structure

{LaplaceTransform} is immediately handed off to {LapTran}. This is done because if the last {LapTran} rule is met, the Laplace transform couldn't be found and it can then return {LaplaceTransform} unevaluated.

Operational Properties

The first rules that are matched against utilize the various operational properties of {LaplaceTransform}, such as:

- Linearity Properties
- Shift properties, i.e. multiplying the function by an exponential
- $ y * x^n = (-1)^n * Deriv(x,n) LaplaceTransform(x,x[2],y) $
- $ y/x = Integrate(x[2],x[2],Infinity) LapTran(x,x[2],y) $

The last operational property dealing with integration is not yet fully bug-tested, it sometimes returns {Undefined} or {Infinity} if the integral returns such.

Transform tables

For elementary functions, Yacas uses transform tables. For instance, the fact that the Laplace transform of $ Cos(t) $ is $ s/(s^2+1) $ is declared in a transform table.

For the purpose of setting up the transform table, a few declaration functions have been defined, which use some generalized pattern matchers to be more flexible in recognizing expressions that are transformable.

Transforming simple functions

For functions like $ Sin(t) $ the transform can be declared with the function {LapTranDef}.

The calling sequence for {LapTranDef} is:

```
LapTranDef(in, out)
```

Currently {in} must be a variable of {_t} and {out} must be a function of {s}. For instance, for the function $ Cos(t) $ there is a declaration:

```
LapTranDef(Cos(_t), s/(s^2+1));
```

The fact that the first argument is a pattern means that each occurrence of the variable to be matched should be referred to as {_t}, as in the example above.

{LapTranDef} generalizes the transform implicitly, in that it will set up the system to actually recognize expressions of the form $ Cos(a*t) $ and $Cos(t/a)$, and return the appropriate answer. The way this is done is by three separate rules for case of {t} itself, {a*t} and {t/a}. This is similar to the {MatchLinear} function that {Integrate} uses, except {LaplaceTransforms} must have {b=0}.

Further Directions

Currenlty $Sin(t)*Cos(t)$ cannot be transformed, because it requires a convolution integral. This will be implemented soon. The inverse laplace transform will be implement soon also.

5.11 Finding real roots of polynomials

5.11.1 real roots

This section deals with finding roots of polynomials in the field of real numbers.

Without loss of generality, the coefficients a_i of a polynomial

$$p = a_n x^n + \ldots + a_0$$

can be considered to be rational numbers, as real-valued numbers are truncated in practice, when doing calculations on a computer.

Assuming that the leading coefficient $a_n = 1$, the polynomial p can also be written as

$$p = p_1^{n_1} \ldots p_m^{n_m}$$

where p_i are the m distinct irreducible monic factors of the form $p_i = x - x_i$, and n_i are multiplicities of the factors. Here the roots are x_i and some of them may be complex. However, complex roots of a polynomial with real coefficients always come in conjugate pairs, so the corresponding irreducible factors should be taken as $p_i = x^2 + c_i x + d_i$. In this case, there will be less than m irreducible factors, and all coefficients will be real.

5.11.2 square free decomposition

To find roots, it is useful to first remove the multiplicities, i.e. to convert the polynomial to one with multiplicity 1 for all irreducible factors, i.e. find the polynomial $p_1 \dots p_m$. This is called the "square-free part" of the original polynomial p.

The square-free part of the polynomial p can be easily found using the polynomial GCD algorithm. The derivative of a polynomial p can be written as:

$$p' = Sum(i, 1, m, p[1]^n[1] * ... * n[i] * p[i]^(n[i] - 1) * (D(x)p[i]) * ... * p[m]^n[m])$$

The g.c.d. of p and p' equals

$$Gcd(p, p') = Factorize(i, 1, m, p[i]^(n[i] - 1)).$$

So if we divide p by $Gcd(p, p')$, we get the square-free part of the polynomial:

$$SquareFree(p) := Div(p, Gcd(p, p')) = p[1] * ... * p[m].$$

In what follows we shall assume that all polynomials are square-free with rational coefficients. Given any polynomial, we can apply the functions {SquareFree} and {Rationalize} and reduce it to this form. The function {Rationalize} converts all numbers in an expression to rational numbers. The function {SquareFree} returns the square-free part of a polynomial. For example:

```
In> Expand((x+1.5)^5)
Out> x^5+7.5*x^4+22.5*x^3+33.75*x^2+25.3125*x +7.59375;
In> SquareFree(Rationalize(%))
Out> x/5+3/10;
In> Simplify(%*5)
Out> (2*x+3)/2;
In> Expand(%)
Out> x+3/2;
```

5.11.3 Sturm sequences

For a polynomial $ p(x) $ of degree $ n $, the Sturm sequence $ p[0] $, $ p[1] $, ... $ p[n] $ is defined by the following equations (following the book [Davenport et al. 1989]):

$$ p[0] = p(x) $$, $$ p[1] = p'(x) $$, $$ p[i] = -remainder(p[i-2],p[i-1]) $$, where $ remainder(p,q) $ is the remainder of division of polynomials $ p $, $ q $.

The polynomial $ p $ can be assumed to have no multiple factors, and thus $ p $ and $ p' $ are relatively prime. The sequence of polynomials in the Sturm sequence are (up to a minus sign) the consecutive polynomials generated by Euclid's algorithm for the calculation of a greatest common divisor for $ p $ and $ p' $, so the last polynomial $p[n] $ will be a constant.

In Yacas, the function {SturmSequence(p)} returns the Sturm sequence of $ p $, assuming $ p $ is a univariate polynomial in $ x $, $ p = p(x) $.

variations in Sturm sequences

Given a Sturm sequence $ S = SturmSequence(p) $ of a polynomial $ p $, the <I>variation</I> in the Sturm sequence $ V(S,y) $ is the number of sign changes in the sequence $ p[0] $, $ p[1] $, ... , $ p[n] $, evaluated at point y, and disregarding zeroes in the sequence.

Sturm's theorem states that if $ a $ and $ b $ are two real numbers which are not roots of $ p $, and $ a < b $, then the number of roots between $ a $ and $ b $ is $ V(S,a) - V(S,b) $. A proof can be found in Knuth, <I>The Art of Computer Programming, Volume 2, Seminumerical Algorithms</I>.

For $ a $ and $ b $, the values $ -Infinity $ and $ Infinity $ can also be used. In these cases, $ V(S,Infinity) $ is the number of sign changes between the leading coefficients of the elements of the Sturm sequence, and $ V(S,-Infinity) $ the same, but with a minus sign for the leading coefficients for which the degree is odd.

Number of real roots

Thus, the number of real roots of a polynomial is $ V(S,-Infinity) - V(S,Infinity) $. The function {NumRealRoots(p)} returns the number of real roots of $ p $.

The function {SturmVariations(S,y)} returns the number of sign changes between the elements in the Sturm sequence $ S $, at point $ x = y $:

```
In> p:=x^2-1
Out> x^2-1;
In> S:=SturmSequence(p)
Out> {x^2-1,2*x,1};
In> SturmVariations(S,-Infinity)- \
SturmVariations(S,Infinity)
Out> 2;
In> NumRealRoots(p)
Out> 2;
In> p:=x^2+1
Out> x^2+1;
In> S:=SturmSequence(p)
Out> {x^2+1,2*x,-1};
In> SturmVariations(S,-Infinity)- \
SturmVariations(S,Infinity)
Out> 0;
In> NumRealRoots(p)
Out> 0;
```

Finding bounds on real roots

Armed with the variations in the Sturm sequence given in the previous section, we can now find the number of real roots in a range (a,b), for $ a < b $. We can thus bound all the roots by subdividing ranges until there is only one root in each range. To be able to start this process, we first need some upper bounds of the roots, or an interval that contains all roots. Davenport gives limits on the roots of a polynomial given the coefficients of the polynomial, as

$$ Abs(a) <= Max(Abs(a[n-1]/a[n]), Abs(a[n-2]/a[n])^(1/2), ... , Abs(a[0]/a[n])^(1/n)) $$,

for all real roots $ a $ of $ p $. This gives the upper bound on the absolute value of the roots of the polynomial in question. if $ p(0) != 0 $, the minimum bound can be obtained also by considering the upper bound of $ p(1/x)*x^n $, and taking $ 1/bound $.

We thus know that given $$ a[max] = MaximumBound(p) $$ and $$ a[min] = MinimumBound(p) $$ for all roots $ a $ of polynomial, either $$ -a[max] <= a <= -a[min] $$ or $$ a[min] <= a <= a[max] $$.

Now we can start the search for the bounds on all roots. The search starts with initial upper and lower bounds on ranges, subdividing ranges until a range contains only one root, and adding that range to the resulting list of bounds. If, when dividing a range, the middle of the range lands on a root, care must be taken, because the bounds should not be on a root themselves. This can be solved by observing that if $ c $ is a root, $ p $ contains a factor $ x-c $, and thus taking $ p(x+c) $ results in a polynomial with all the roots shifted by a constant $ -c $, and the root $ c $ moved to zero, e.g. $ p(x+c) $ contains a factor $ x $. Thus a new ranges to the left and right of $ c $ can be determined by first calculating the minimum bound $ M $ of $ p(x+c)/x $. When the original range was (a,b), and $ c = (a+b)/2 $ is a root, the new ranges should become (a,$c-M$) and ($c+M$,b).

In Yacas, {MimimumBound(p)} returns the lower bound described above, and {MaximumBound(p)} returns the upper bound on the roots in $ p $. These bounds are returned as rational numbers. {BoundRealRoots(p)} returns a list with sublists with the bounds on the roots of a polynomial:

```
In> p:=(x+20)*(x+10)
Out> (x+20)*(x+10);
In> MinimumBound(p)
Out> 10/3;
In> MaximumBound(p)
Out> 60;
In> BoundRealRoots(p)
Out> {{-95/3,-35/2},{-35/2,-10/3}};
In> N(%)
Out> {{-31.6666666666,-17.5}, {-17.5,-3.3333333333}};
```

It should be noted that since all calculations are done with rational numbers, the algorithm for bounding the roots is very robust. This is important, as the roots can be very unstable for small variations in the coefficients of the polynomial in question (see Davenport).

Finding real roots given the bounds on the roots

Given the bounds on the real roots as determined in the previous section, two methods for finding roots are available: the secant method or the Newton method, where the function is locally approximated by a line, and extrapolated to find a new estimate for a root. This method converges quickly when "sufficiently" near a root, but can easily fail otherwise. The secant method can easily send the search to infinity.

The bisection method is more robust, but slower. It works by taking the middle of the range, and checking signs of the polynomial to select the half-range where the root is. As there is only one root in the range (a,b), in general it will be true that $ p(a)*p(b) < 0 $, which is assumed by this method.

Yacas finds the roots by first trying the secant method, starting in the middle of the range, $ c = (a+b)/2 $. If this fails the bisection method is tried.

The function call to find the real roots of a polynomial $ p $ in variable $ x $ is {FindRealRoots(p)}, for example:

```
In> p:=Expand((x+3.1)*(x-6.23))
Out> x^2-3.13*x-19.313;
In> FindRealRoots(p)
Out> {-3.1,6.23};
In> p:=Expand((x+3.1)^3*(x-6.23))
Out> x^4+3.07*x^3-29.109*x^2-149.8199\
In> *x-185.59793;
In> p:=SquareFree(Rationalize( \
In> Expand((x+3.1)^3*(x-6.23))))
Out> (-160000*x^2+500800*x+3090080)/2611467;
In> FindRealRoots(p)
Out> {-3.1,6.23};
```

5.12 Number theory algorithms

This chapter describes the algorithms used for computing various number-theoretic functions. We call "number-theoretic" any function that takes integer arguments, produces integer values, and is of interest to number theory.

5.12.1 Euclidean GCD algorithms

The main algorithm for the calculation of the GCD of two integers is the binary Euclidean algorithm. It is based on the following identities: $ Gcd(a,b) = Gcd(b,a) $, $ Gcd(a,b) = Gcd(a-b,b)$, and for odd b, $Gcd(2*a,b) = Gcd(a,b)$. Thus we can produce a sequence of pairs with the same GCD as the original two numbers, and each pair will be at most half the size of the previous pair. The number of steps is logarithmic in the number of digits in a, b. The only operations needed for this algorithm are binary shifts and subtractions (no modular division is necessary). The low-level function for this is {MathGcd}.

To speed up the calculation when one of the numbers is much larger than another, one could use the property $Gcd(a,b)=Gcd(a,Mod(a,b))$. This will introduce an additional modular division into the algorithm; this is a slow operation when the numbers are large.

5.12.2 Prime numbers: the Miller-Rabin test and its improvements

Small prime numbers

- EVAL "$p<=" : (ToString()Write(FastIsPrime(0))) : "$"

are simply stored in a precomputed table as an array of bits; the bits corresponding to prime numbers are set to 1. This makes primality testing on small numbers very quick. This is implemented by the function {FastIsPrime}.

Primality of larger numbers is tested by the function {IsPrime} that uses the Miller-Rabin algorithm. (FOOT Initial implementation and documentation was supplied by Christian Obrecht.) This algorithm is deterministic (guaranteed correct within a certain running time) for "small" numbers $n<3.4*10^13$ and probabilistic (correct with high probability but not guaranteed) for larger numbers. In other words, the Miller-Rabin test could sometimes flag a large number n as prime when in fact n is composite; but the probability for this to happen can be made extremely small. The basic reference is [Rabin 1980]. We also implemented some of the improvements suggested in [Davenport 1992].

The idea of the Miller-Rabin algorithm is to improve the Fermat primality test. If n is prime, then for any x we have $Gcd(n,x)=1$. Then by Fermat's "little theorem", $x^(n-1):=Mod(1,n)$. (This is really a simple statement; if n is prime, then $n-1$ nonzero remainders modulo n: 1, 2, ..., $n-1$ form a cyclic multiplicative group.) Therefore we pick some "base" integer x and compute $Mod(x^(n-1), n)$; this is a quick computation even if n is large. If this value is not equal to 1 for some base x, then n is definitely not prime. However, we cannot test every base $x<n$; instead we test only some x, so it may happen that we miss the right values of x that would expose the non-primality of n. So Fermat's test sometimes fails, i.e. says that n is a prime when n is in fact not a prime. Also there are infinitely many integers called "Carmichael numbers" which are not prime but pass the Fermat test for every base.

The Miller-Rabin algorithm improves on this by using the property that for prime n there are no nontrivial square roots of unity in the ring of integers modulo n (this is Lagrange's theorem). In other words, if $x^2:=Mod(1,n)$ for some x, then x must be equal to 1 or -1 modulo n. (Since $n-1$ is equal to -1 modulo n, we have $n-1$ as a trivial square root of unity modulo n. Note that even if n is prime there may be nontrivial divisors of 1, for example, $2*49:=Mod(1,97)$.)

We can check that n is odd before applying any primality test. (A test $n^2:=Mod(1,24)$ guarantees that n is not divisible by 2 or 3. For large n it is faster to first compute $Mod(n,24)$ rather than n^2, or test n directly.) Then we note that in Fermat's test the number $n-1$ is certainly a composite number because $n-1$ is even. So if we first find the largest power of 2 in $n-1$ and decompose $n-1=2^r*q$ with q odd, then $x^(n-1):=Mod(a^(2^r),n)$

where $a:=Mod(x^q,n)$. (Here $r>=1$ since n is odd.) In other words, the number $Mod(x^(n-1),n)$ is obtained by repeated squaring of the number a. We get a sequence of r repeated squares: a, a^2, . . . , $a^(2^r)$. The last element of this sequence must be 1 if n passes the Fermat test. (If it does not pass, n is definitely a composite number.) If n passes the Fermat test, the last-but-one element $a^(2^(r-1))$ of the sequence of squares is a square root of unity modulo n. We can check whether this square root is non-trivial (i.e. not equal to 1 or -1 modulo n). If it is non-trivial, then n definitely cannot be a prime. If it is trivial and equal to 1, we can check the preceding element, and so on. If an element is equal to -1, we cannot say anything, i.e. the test passes (n is "probably a prime").

This procedure can be summarized like this:

- 1. Find the largest power of 2 in $n-1$ and an odd number q such that $n-1=2^r*q$.
- 2. Select the "base number" $x<n$. Compute the sequence $a:=Mod(x^q,n)$, a^2, a^4, ..., $a^(2^r)$ by repeated squaring modulo n. This sequence contains at least two elements since $r>=1$.
- 3. If $a=1$ or $a=n-1$, the test passes on the base number x. Otherwise, the test passes if at least one of the elements of the sequence is equal to $n-1$ and fails if none of them are equal to $n-1$. This simplified procedure works because the first element that is equal to 1 must be preceded by a -1, or else we would find a nontrivial root of unity.

Here is a more formal definition. An odd integer n is called strongly-probably-prime for base b if $b^q:=Mod(1,n)$ or $b^(q*2^i):=Mod(n-1,n)$ for some i such that $0 <= i < r$, where q and r are such that q is odd and $n-1 = q*2^r$.

A practical application of this procedure needs to select particular base numbers. It is advantageous (according to [Pomerance et al. 1980]) to choose prime numbers b as bases, because for a composite base $b=p*q$, if n is a strong pseudoprime for both p and q, then it is very probable that n is a strong pseudoprime also for b, so composite bases rarely give new information.

An additional check suggested by [Davenport 1992] is activated if $r>2$ (i.e. if $n:=Mod(1,8)$ which is true for only 1/4 of all odd numbers). If $i>=1$ is found such that $b^(q*2^i):=Mod(n-1,n)$, then $b^(q*2^(i-1))$ is a square root of -1 modulo n. If n is prime, there may be only two different square roots of -1. Therefore we should store the set of found values of roots of -1; if there are more than two such roots, then we woill find some roots $s1$, $s2$ of -1 such that $s1+s2!=Mod(0,n)$. But $s1^2-s2^2:=Mod(0,n)$. Therefore n is definitely composite, e.g. $Gcd(s1+s2,n)>1$. This check costs very little computational effort but guards against some strong pseudoprimes.

Yet another small improvement comes from [Damgard et al. 1993]. They found that the strong primality test sometimes (rarely) passes on composite numbers n for more than $1/8$ of all bases $x<n$ if n is such that either $3*n+1$ or $8*n+1$ is a perfect square, or if n is a Carmichael number. Checking Carmichael numbers is slow, but it is easy to show that if n is a large enough prime number, then neither $3*n+1$, nor $8*n+1$, nor any $s*n+1$ with small integer s can be a perfect square. [If $s*n+1=r^2$, then $s*n=(r-1)*(r+1)$.] Testing for a perfect square is quick and does not slow down the algorithm. This is however not implemented in Yacas because it seems that perfect squares are too rare for this improvement to be significant.

If an integer is not "strongly-probably-prime" for a given base b, then it is a composite number. However, the converse statement is false, i.e. "strongly-probably-prime" numbers can actually be composite. Composite strongly-probably-prime numbers for base b are called strong pseudoprimes for base b. There is a theorem that if n is composite, then among all numbers b such that $1 < b < n$, at most one fourth are such that n is a strong pseudoprime for base b. Therefore if n is strongly-probably-prime for many bases, then the probability for n to be composite is very small.

For numbers less than $B=34155071728321$, exhaustive (FOOT And surely exhausting.) computations have shown that there are no strong pseudoprimes simultaneously for bases 2, 3, 5, 7, 11, 13 and 17. This gives a simple and reliable primality test for integers below B. If $n >= B$, the Rabin-Miller method consists in checking if n is strongly-probably-prime for k base numbers b. The base numbers are chosen to be consecutive "weak pseudoprimes" that are easy to generate (see below the function {NextPseudoPrime}).

In the implemented routine {RabinMiller}, the number of bases k is chosen to make the probability of erroneously passing the test $p < 10^(-25)$. (Note that this is not the same as the probability to give an incorrect answer, because all numbers that do not pass the test are definitely composite.) The probability for the test to pass mistakenly on a given number is found as follows. Suppose the number of bases k is fixed. Then the probability for a given composite number to pass the test is less than $p[f]=4^(-k)$. The probability for a given number n to be prime is roughly $p[p]=1/Ln(n)$ and to be composite $p[c]=1-1/Ln(n)$. Prime numbers never fail the test. Therefore, the probability for the test to pass is $p[f]*p[c]+p[p]$ and the probability to pass erroneously is $$ p = (p[f]*p[c])/(p[f]*p[c]+p[p]) < Ln(n)*4^(-k) $$. To make $p<epsilon$, it is enough to select $k=1/Ln(4)*(Ln(n)-Ln(epsilon))$.

Before calling {MillerRabin}, the function {IsPrime} performs two quick checks: first, for $n>=4$ it checks that n is not divisible by 2 or 3 (all primes larger than 4 must satisfy this); second, for $n>257$, it checks that n does not contain small prime factors $p<=257$. This is checked by evaluating the GCD of n with the precomputed product of all primes up to 257. The computation of the GCD is quick and saves time in case a small prime factor is present.

There is also a function {NextPrime(n)} that returns the smallest prime number larger than {n}. This function uses a sequence 5,7,11,13,... generated by the function {NextPseudoPrime}. This sequence contains numbers not divisible by 2 or 3 (but perhaps divisible by 5,7,...). The function {NextPseudoPrime} is very fast because it does not perform a full primality test.

The function {NextPrime} however does check each of these pseudoprimes using {IsPrime} and finds the first prime number.

5.12.3 Factorization of integers

When we find from the primality test that an integer n is composite, we usually do not obtain any factors of n. Factorization is implemented by functions {Factor} and {Factors}. Both functions use the same algorithms to find all prime factors of a given integer n. (Before doing this, the primality checking algorithm is used to detect whether n is a prime number.) Factorization consists of repeatedly finding a factor, i.e. an integer f such that $Mod(n, f)=0$, and dividing n by f. (Of course, each fastor f needs to be factorized too.)

small prime factors

First we determine whether the number n contains "small" prime factors $p<=257$. A quick test is to find the GCD of n and the product of all primes up to 257: if the GCD is greater than 1, then n has at least one small prime factor. (The product of primes is precomputed.) If this is the case, the trial division algorithm is used: n is divided by all prime numbers $p<=257$ until a factor is found. {NextPseudoPrime} is used to generate the sequence of candidate divisors p.

checking for prime powers

After separating small prime factors, we test whether the number n is an integer power of a prime number, i.e. whether $n=p^s$ for some prime number p and an integer $s>=1$. This is tested by the following algorithm. We already know that n is not prime and that n does not contain any small prime factors up to 257. Therefore if $n=p^s$, then $p>257$ and $2<=s<s[0]=Ln(n)/Ln(257)$. In other words, we only need to look for powers not greater than $s[0]$. This number can be approximated by the "integer logarithm" of n in base 257 (routine {IntLog(n, 257)}).

Now we need to check whether n is of the form p^s for $s=2$, 3, . . . , $s[0]$. Note that if for example $n=p^24$ for some p, then the square root of n will already be an integer, $n^(1/2)=p^12$. Therefore it is enough to test whether $n^(1/s)$ is an integer for all prime values of s up to $s[0]$, and then we will definitely discover whether n is a power of some other integer. The testing is performed using the integer n-th root function {IntNthRoot} which quickly computes the integer part of n-th root of an integer number. If we discover that n has an integer root p of order s, we have to check that p itself is a prime power (we use the same algorithm

recursively). The number n is a prime power if and only if p is itself a prime power. If we find no integer roots of orders $s<=s[0]$, then n is not a prime power.

Pollard's "rho" algorithm

If the number n is not a prime power, the Pollard "rho" algorithm is applied [Pollard 1978]. The Pollard "rho" algorithm takes an irreducible polynomial, e.g. $p(x)=x^2+1$ and builds a sequence of integers $x[k+1]:=Mod(p(x[k]),n)$, starting from $x[0]=2$. For each k, the value $x[2*k]-x[k]$ is attempted as possibly containing a common factor with n. The GCD of $x[2*k]-x[k]$ with n is computed, and if $Gcd(x[2*k]-x[k],n)>1$, then that GCD value divides n.

The idea behind the "rho" algorithm is to generate an effectively random sequence of trial numbers $t[k]$ that may have a common factor with n. The efficiency of this algorithm is determined by the size of the smallest factor p of n. Suppose p is the smallest prime factor of n and suppose we generate a random sequence of integers $t[k]$ such that $1<=t[k]<n$. It is clear that, on the average, a fraction $1/p$ of these integers will be divisible by p. Therefore (if $t[k]$ are truly random) we should need on the average p tries until we find $t[k]$ which is accidentally divisible by p. In practice, of course, we do not use a truly random sequence and the number of tries before we find a factor p may be significantly different from p. The quadratic polynomial seems to help reduce the number of tries in most cases.

But the Pollard "rho" algorithm may actually enter an infinite loop when the sequence $x[k]$ repeats itself without giving any factors of n. For example, the unmodified "rho" algorithm starting from $x[0]=2$ loops on the number 703. The loop is detected by comparing $x[2*k]$ and $x[k]$. When these two quantities become equal to each other for the first time, the loop may not yet have occurred so the value of GCD is set to 1 and the sequence is continued. But when the equality of $x[2*k]$ and $x[k]$ occurs many times, it indicates that the algorithm has entered a loop. A solution is to randomly choose a different starting number $x[0]$ when a loop occurs and try factoring again, and keep trying new random starting numbers between 1 and n until a non-looping sequence is found. The current implementation stops after 100 restart attempts and prints an error message, "failed to factorize number".

A better (and faster) integer factoring algorithm needs to be implemented in Yacas.

overview of algorithms

Modern factoring algorithms are all probabilistic (i.e. they do not guarantee a particular finishing time) and fall into three categories:

- 1. Methods that work well (i.e. quickly) if there is a relatively small factor p of n (even if n itself is large). Pollard's "rho" algorithm belongs to this category. The fastest in this category is Lenstra's elliptic curves method (ECM).
- 2. Methods that work equally quickly regardless of the size of factors (but slower with larger n). These are the continued fractions method and the various "sieve" methods. The current best is the "General Number Field Sieve" (GNFS) but it is quite a complicated algorithm requiring operations with high-order algebraic numbers. The next best one is the "Multiple Polynomial Quadratic Sieve" (MPQS).
- 3. Methods that are suitable only for numbers of special "interesting" form, e.g. Fermat numbers $2^(2^k)-1$ or generally numbers of the form r^s+a where s is large but r and a are very small integers. The best method seems to be the "Special Number Field Sieve" which is a faster variant of the GNFS adapted to the problem.

There is ample literature describing these algorithms.

5.12.4 The Jacobi symbol

A number m is a "quadratic residue modulo n" if there exists a number k such that $k^2:=Mod(m,n)$.

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 m/n is an integer.

The Jacobi symbol $[m/n;]$ is defined as the product of the Legendre symbols of the prime factors $f[i]$ of $n=f[1]^p[1]*...*f[s]^p[s]$, $$ [m/n;] := [m/f[1];]^p[1]*...*[m/f[s];]^p[s] $$. (Here we used the same notation $[a/b;]$ 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.

The Jacobi symbol can be efficiently computed without knowing the full factorization of the number n. The currently used method is based on the following four identities for the Jacobi symbol: * 1. $[a/1;] = 1$. * 2. $[2/b;] = (-1)^((b^2-1)/8)$. * 3. $[(a*b)/c;] = [a/c;]*[b/c;]$. * 4. If $a:=Mod(b,c)$, then $[a/c;]=[b/c;]$. * 5. If a, b are both odd, then $[a/b;]=[b/a;] * (-1)^((a-1)*(b-1)/4)$.

Using these identities, we can recursively reduce the computation of the Jacobi symbol $[a/b;]$ to the computation of the Jacobi symbol for numbers that are on the average half as large. This is similar to the fast "binary" Euclidean algorithm for the computation of the GCD. The number of levels of recursion is logarithmic in the arguments a, b.

More formally, Jacobi symbol $[a/b;]$ is computed by the following algorithm. (The number b must be an odd positive integer, otherwise the result is undefined.)

- 1. If $b=1$, return 1 and stop. If $a=0$, return 0 and stop. Otherwise, replace $[a/b;]$ by $[Mod(a,b)/b;]$ (identity 4).
- 2. Find the largest power of 2 that divides a. Say, $a=2^s*c$ where c is odd. Replace $[a/b;]$ by $[c/b;]*(-1)^(s*(b^2-1)/8)$ (identities 2 and 3).
- 3. Now that $c<b$, replace $[c/b;]$ by $[b/c;]*(-1)^((b-1)*(c-1)/4)$ (identity 5).
- 4. Continue to step 1.

Note that the arguments a, b may be very large integers and we should avoid performing multiplications of these numbers. We can compute $(-1)^((b-1)*(c-1)/4)$ without multiplications. This expression is equal to 1 if either b or c is equal to 1 mod 4; it is equal to -1 only if both b and c are equal to 3 mod 4. Also, $(-1)^((b^2-1)/8)$ is equal to 1 if either $b:=1$ or $b:=7$ mod 8, and it is equal to -1 if $b:=3$ or $b:=5$ mod 8. Of course, if s is even, none of this needs to be computed.

5.12.5 Integer partitions

partitions of an integer

A partition of an integer n is a way of writing n as the sum of positive integers, where the order of these integers is unimportant. For example, there are 3 ways to write the number 3 in this way: $3=1+1+1$, $3=1+2$, $3=3$. The function {PartitionsP} counts the number of such partitions.

partitions of an integer!by Rademacher-Hardy-Ramanujan series

Large n

The first algorithm used to compute this function uses the Rademacher-Hardy-Ramanujan (RHR) theorem and is efficient for large n. (See for example [Ahlgren et al. 2001].) The number of partitions $P(n)$ is equal to an infinite sum:

$$ P(n) = 1/(Pi*Sqrt(2))*Sum(k,1,Infinity, Sqrt(k)*A(k,n)*S(k,n)) $$, where the functions A and S are defined as follows:

$$S(k,n) := Deriv(n) Sinh(Pi/k*Sqrt(2/3*(n-1/24)))/Sqrt(n-1/24)$$

$$ A(k,n) := Sum(l,1,k, delta(Gcd(l,k),1)*Exp(-2*Pi*I*(l*n)/k+Pi*I*B(k,l))) $$, where $delta(x,y)$ is the Kronecker delta function (so that the summation goes only over integers l which are mutually prime with k) and B is defined by $$ B(k,l) := Sum(j,1,k-1, j/k*(l*j/k-Floor(l*j/k)-1/2)) $$.

The first term of the series gives, at large n, the Hardy-Ramanujan asymptotic estimate, $$ P(n) <> P_0(n) := 1/(4*n*Sqrt(3))*Exp(Pi*Sqrt((2*n)/3))$$. The absolute value of each term decays quickly, so after $O(Sqrt(n))$ terms the series gives an answer that is very close to the integer result.

There exist estimates of the error of this series, but they are complicated. The series is sufficiently well-behaved and it is easier to determine the truncation point heuristically. Each term of the series is either 0 (when all terms in $A(k,n)$ happen to cancel) or has a magnitude which is not very much larger than the magnitude of the previous nonzero term. (But the series is not actually monotonic.) In the current implementation, the series is truncated when $Abs(A(k,n)*S(n)*Sqrt(k))$ becomes smaller than 0.1 for the first time; in any case, the maximum number of calculated terms is $5+Sqrt(n)/2$. One can show that asymptotically for large n, the required number of terms is less than $mu/Ln(mu)$, where $mu:=Pi*Sqrt((2*n)/3)$.

[Ahlgren et al. 2001] mention that there exist explicit constants $B[1]$ and $B[2]$ such that $$ Abs(P(n)-Sum(k,1,B[1]*Sqrt(n),A(k,n))) < B[2]*n^(-1/4)$$.

The floating-point precision necessary to obtain the integer result must be at least the number of digits in the first term $P_0(n)$, i.e. $$ Prec > (Pi*Sqrt(2/3*n)-Ln(4*n*Sqrt(3)))/Ln(10) $$. However, Yacas currently uses the fixed-point precision model. Therefore, the current implementation divides the series by $P_0(n)$ and computes all terms to $Prec$ digits.

The RHR algorithm requires $O((n/Ln(n))^(3/2))$ operations, of which $O(n/Ln(n))$ are long multiplications at precision $Prec<>O(Sqrt(n))$ digits. The computational cost is therefore $O(n/Ln(n)*M(Sqrt(n)))$.

partitions of an integer!by recurrence relation

Small n

The second, simpler algorithm involves a recurrence relation $$ P[n] = Sum(k,1,n, (-1)^(k+1)*(P[n-k*(3*k-1)/2]+P[n-k*(3*k+1)/2])) $$. The sum can be written out as $$ P(n-1)+P(n-2)-P(n-5)-P(n-7)+... $$, where 1, 2, 5, 7, ... is the "generalized pentagonal sequence" generated by the pairs $k*(3*k-1)/2$, $k*(3*k+1)/2$ for $k=1$, 2, ... The recurrence starts from $P(0)=1$, $P(1)=1$. (This is implemented as {PartitionsP'recur}.)

The sum is actually not over all k up to n but is truncated when the pentagonal sequence grows above n. Therefore, it contains only $O(Sqrt(n))$ terms. However, computing $P(n)$ using the recurrence relation requires computing and storing $P(k)$ for all $1<=k<=n$. No long multiplications are necessary, but the number of long additions of numbers with $Prec<>O(Sqrt(n))$ digits is $O(n^(3/2))$. Therefore the computational cost is $O(n^2)$. This is asymptotically slower than the RHR algorithm even if a slow $O(n^2)$ multiplication is used. With internal Yacas math, the recurrence relation is faster for $n<300$ or so, and for larger n the RHR algorithm is faster.

Miscellaneous functions

divisors

The function {Divisors} currently returns the number of divisors of integer, while {DivisorsSum} returns the sum of these divisors. (The current algorithms need to factor the number.) The following theorem is used:

Let $p[1]^k[1] * ... * p[r]^k[r]$ be the prime factorization of n, where r is the number of prime factors and $k[r]$ is the multiplicity of the r-th factor. Then

..math:: Divisors(n) =(k[1]+1)*...*(k[r]+1)

..math:: DivisorsSum(n) = (((p[1]^(k[1]+1) -1)/(p[1]-1)))*...*(p[r]^(k[r]+1)-1)/(p[r]-1) $$.

divisors!proper

The functions {ProperDivisors} and {ProperDivisorsSum} are functions that do the same as the above functions, except they do not consider the number n as a divisor for itself. These functions are defined by:

$ ProperDivisors(n) = Divisors(n) - 1 $,

$ ProperDivisorsSum(n) = DivisorsSum(n) - n $.

Another number-theoretic function is {Moebius}, defined as follows: $Moebius(n)=(-1)^r$ if no factors of n are repeated, $Moebius(n)=0$ if some factors are repeated, and $Moebius(n)=1$ if $n = 1$. This again requires to factor the number n completely and investigate the properties of its prime factors. From the definition, it can be seen that if n is prime, then $Moebius(n) = -1 $. The predicate {IsSquareFree(n)} then reduces to $Moebius(n)!=0$, which means that no factors of n are repeated.

5.12.6 Gaussian integers

A "Gaussian integer" is a complex number of the form $z = a + b * \imath$, where a and b are ordinary (rational) integers. (FOOT To distinguish ordinary integers from Gaussian integers, the ordinary integers (with no imaginary part) are called "rational integers".) The ring of Gaussian integers is usually denoted by Z[I] in the mathematical literature. It is an example of a ring of algebraic integers.

The function {GaussianNorm} computes the norm $N(z)=a^2+b^2$ of z. The norm plays a fundamental role in the arithmetic of Gaussian integers, since it has the multiplicative property: $$ N(z.w) = N(z).N(w) $$.

A unit of a ring is an element that divides any other element of the ring. There are four units in the Gaussian integers: 1, -1, I, $-I$. They are exactly the Gaussian integers whose norm is 1. The predicate {IsGaussianUnit} tests for a Gaussian unit.

Two Gaussian integers z and w are "associated" is z/w is a unit. For example, $2+I$ and $-1+2*I$ are associated.

A Gaussian integer is called prime if it is only divisible by the units and by its associates. It can be shown that the primes in the ring of Gaussian integers are:

- 1. $1+i$ and its associates.
- 2. The rational (ordinary) primes of the form $4*n+3$.
- 3. The factors $a+b*I$ of rational primes p of the form $p=4*n+1$, whose norm is $p=a^2+b^2$.

For example, 7 is prime as a Gaussian integer, while 5 is not, since $ 5 = (2+I)*(2-I) $. Here $2+I$ is a Gaussian prime.

Factors

The ring of Gaussian integers is an example of an Euclidean ring, i.e. a ring where there is a division algorithm. This makes it possible to compute the greatest common divisor using Euclid's algorithm. This is what the function {GaussianGcd} computes.

As a consequence, one can prove a version of the fundamental theorem of arithmetic for this ring: The expression of a Gaussian integer as a product of primes is unique, apart from the order of primes, the presence of units, and the ambiguities between associated primes.

The function {GaussianFactors} finds this expression of a Gaussian integer z as the product of Gaussian primes, and returns the result as a list of pairs {{p,e}}, where p is a Gaussian prime and e is the corresponding exponent. To do that, an auxiliary function called {GaussianFactorPrime} is used. This function finds a factor of a rational prime of the form $4*n+1$. We compute $a := (2*n)!$ (mod p). By Wilson's theorem a^2 is congruent to -1 (mod p), and it follows that p divides $(a+I)*(a-I)=a^2+1$ in the Gaussian integers. The desired factor is then the {GaussianGcd} of $a+I$ and p. If the result is $a+b*I$, then $p=a^2+b^2$.

If z is a rational (i.e. real) integer, we factor z in the Gaussian integers by first factoring it in the rational integers, and after that by factoring each of the integer prime factors in the Gaussian integers.

If z is not a rational integer, we find its possible Gaussian prime factors by first factoring its norm $N(z)$ and then computing the exponent of each of the factors of $N(z)$ in the decomposition of z.

References for Gaussian integers

- 1. G. H. Hardy and E. M. Wright, An Introduction to the Theory of Numbers. Oxford University Press (1945).
- 2. H. Pollard, The theory of Algebraic Numbers. Wiley, New York (1965).

5.12.7 A simple factorization algorithm for univariate polynomials

This section discusses factoring polynomials using arithmetic modulo prime numbers. Information was used from D. Knuth, <I>The Art of Computer Programming, Volume 2, Seminumerical Algorithms </I> and J.H. Davenport et. al., <I>Computer Algebra, SYSTEMS AND ALGORITHMS FOR ALGEBRAIC COMPUTATION</I>.

A simple factorization algorithm is developed for univariate polynomials. This algorithm is implemented as the function {BinaryFactors}. The algorithm was named the binary factoring algorithm since it determines factors to a polynomial modulo 2^n for successive values of n, effectively adding one binary digit to the solution in each iteration. No reference to this algorithm has been found so far in literature.

Berlekamp showed that polynomials can be efficiently factored when arithmetic is done modulo a prime. The Berlekamp algorithm is only efficient for small primes, but after that Hensel lifting can be used to determine the factors modulo larger numbers.

The algorithm presented here is similar in approach to applying the Berlekamp algorithm to factor modulo a small prime, and then factoring modulo powers of this prime (using the solutions found modulo the small prime by the Berlekamp algorithm) by applying Hensel lifting. However it is simpler in set up. It factors modulo 2, by trying all possible factors modulo 2 (two possibilities, if the polynomial is monic). This performs the same action usually left to the Berlekamp step. After that, given a solution modulo 2^n, it will test for a solution f_i modulo 2^n if f_i or $f_i + 2^n$ are a solution modulo $2^(n+1)$.

This scheme raises the precision of the solution with one digit in binary representation. This is similar to the linear Hensel lifting algorithm, which factors modulo p^n for some prime p, where n increases by one after each iteration. There is also a quadratic version of Hensel lifting which factors modulo p^2^n, in effect doubling the number of digits (in p-adic expansion) of the solution after each iteration. However, according to “Davenport”, the quadratic algorithm is not necessarily faster.

The algorithm here thus should be equivalent in complexity to Hensel lifting linear version. This has not been verified yet.

5.12.8 Modular arithmetic

This section copies some definitions and rules from <I>The Art of Computer Programming, Volume 1, Fundamental Algorithms </I> regarding arithmetic modulo an integer.

Arithmetic modulo an integer p requires performing the arithmetic operation and afterwards determining that integer modulo p. A number x can be written as $$x=q*p+r$$ where q is called the quotient, and r remainder. There is some liberty in the range one chooses r to be in. If r is an integer in the range ${0,1, ... ,(p-1)}$ then it is the modulo, $r = Mod(x,p)$.

When $Mod(x,p) = Mod(y,p)$, the notation $Mod(x=y,p)$ is used. All arithmetic calculations are done modulo an integer p in that case.

For calculations modulo some p the following rules hold:

- If $Mod(a=b,p)$ and $Mod(x=y,p)$, then $Mod(a*x=b*y,p)$, $Mod(a+x=b+y,p)$, and $Mod(a-x=b-y,p)$. This means that for instance also $Mod(x^n,p) = Mod(Mod(x,p)^n,p)$
- Two numbers x and y are relatively prime if they don't share a common factor, that is, if their greatest common denominator is one, $Gcd(x,y)=1$.
- If $Mod(a*x=b*y,p)$ and if $Mod(a=b,p)$, and if a and p are relatively prime, then $Mod(x=y,p)$. This is useful for dividing out common factors.
- $Mod(a=b,p)$ if and only if $Mod(a*n=b*n,n*p)$ when $n != 0$. Also, if r and s are relatively prime, then $Mod(a=b,r*s)$ only if $Mod(a=b,r)$ and $Mod(a=b,s)$. These rules are useful when the modulus is changed.

For polynomials $v_1(x)$ and $v_2(x)$ it further holds that $$ Mod((v_1(x)+v_2(x))^p = v_1(x)^p + v_2(x)^p,p) $$ This follows by writing out the expression, noting that the binomial coefficients that result are multiples of p, and thus their value modulo p is zero (p divides these coefficients), so only the two terms on the right hand side remain.

Some corollaries

One corollary of the rules for calculations modulo an integer is Fermat's theorem, 1640 : if p is a prime number then $$Mod(a^p=a,p)$$ for all integers a (for a proof, see Knuth).

An interesting corollary to this is that, for some prime integer p: $$ Mod(v(x)^p = v(x^p),p) $$. This follows from writing it out and using Fermat's theorem to replace a^p with a where appropriate (the coefficients to the polynomial when written out, on the left hand side).

Factoring using modular arithmetic

The task is to factor a polynomial

$$ p(x) = a_n*x^n + ... + a_0 $$

into a form

$$ p(x) = C*g(x)*f_1(x)^p_1*f_2(x)^p_2 * ... * f_m(x)^p_m $$

Where $f_i(x)$ are irreducible polynomials of the form:

$$ f_i(x) = x+c_i $$

The part that could not be factorized is returned as $g(x)$, with a possible constant factor C.

The factors $f_i(x)$ and $g(x)$ are determined uniquely by requiring them to be monic. The constant C accounts for a common factor.

The c_i constants in the resulting solutions $f_i(x)$ can be rational numbers (or even complex numbers, if Gaussian integers are used).

Preparing the polynomial for factorization

The final factoring algorithm needs the input polynomial to be monic with integer coefficients (a polynomial is monic if its leading coefficient is one). Given a non-monic polynomial with rational coefficients, the following steps are performed:

Convert polynomial with rational coefficients to polynomial with integer coefficients

First the least common multiple lcm of the denominators of the coefficients $p(x)$ has to be found, and the polynomial is multiplied by this number. Afterwards, the C constant in the result should have a factor $1/lcm$.

The polynomial now only has integer coefficients.

Convert polynomial to a monic polynomial

The next step is to convert the polynomial to one where the leading coefficient is one. In order to do so, following "Davenport", the following steps have to be taken:

- 1. Multiply the polynomial by $a_n^(n-1)$
- 2. Perform the substitution $x=(y/a_n)$

The polynomial is now a monic polynomial in y.

After factoring, the irreducible factors of $p(x)$ can be obtained by multiplying C with $1/(a_n^(n-1))$, and replacing y with a_n*x. The irreducible solutions a_n*x+c_i can be replaced by $x+c_i/a_i$ after multiplying C by a_n, converting the factors to monic factors.

After the steps described here the polynomial is now monic with integer coefficients, and the factorization of this polynomial can be used to determine the factors of the original polynomial $p(x)$.

Definition of division of polynomials

To factor a polynomial a division operation for polynomials modulo some integer is needed. This algorithm needs to return a quotient $q(x)$ and remainder $r(x)$ such that:

$$ Mod(p(x) = q(r)*d(x) + r(x),p) $$

for some polymomial $d(x)$ to be divided by, modulo some integer p. $d(x)$ is said to divide $p(x)$ (modulo p) if $r(x)$ is zero. It is then a factor modulo p.

For binary factoring algorithm it is important that if some monic $d(x)$ divides $p(x)$, then it also divides $p(x)$ modulo some integer p.

Define $deg(f(x))$ to be the degree of $f(x)$ and $lc(f(x))$ to be the leading coefficient of $f(x)$. Then, if $deg(p(x)) >= deg(d(x))$, one can compute an integer s such that

$$ Mod(lc(d(x))*s = lc(p(x)),p) $$

If p is prime, then

$$s = Mod(lc(p(x))*lc(d(x))^(p-2),p) $$

Because $Mod(a^(p-1) = 1,p)$ for any a. If p is not prime but $d(x)$ is monic (and thus $lc(d(x)) = 1$),

$$s = lc(p(x)) $$

This identity can also be used when dividing in general (not modulo some integer), since the divisor is monic.

The quotient can then be updated by adding a term:

$term = s*x^(deg(p(x))-deg(d(x)))$

and updating the polynomial to be divided, $p(x)$, by subtracting $d(x)*term$. The resulting polynomial to be divided now has a degree one smaller than the previous.

When the degree of $p(x)$ is less than the degree of $d(x)$ it is returned as the remainder.

A full division algorithm for arbitrary integer $p>1$ with $lc(d(x)) = 1$ would thus look like:

```
divide(p(x),d(x),p)
   q(x) = 0
   r(x) = p(x)
   while (deg(r(x)) >= deg(d(x)))
      s = lc(r(x))
      term = s*x^(deg(r(x))-deg(d(x)))
      q(x) = q(x) + term
```

```
    r(x) = r(x) - term*d(x) mod p
  return {q(x),r(x)}
```

The reason we can get away with factoring modulo 2^n as opposed to factoring modulo some prime p in later sections is that the divisor $d(x)$ is monic. Its leading coefficient is one and thus $q(x)$ and $r(x)$ can be uniquely determined. If p is not prime and $lc(d(x))$ is not equal to one, there might be multiple combinations for which $p(x) = q(x)*d(x)+r(x)$, and we are interested in the combinations where $r(x)$ is zero. This can be costly to determine unless {q(x),r(x)} is unique. This is the case here because we are factoring a monic polynomial, and are thus only interested in cases where $lc(d(x)) = 1$.

Determining possible factors modulo 2

We start with a polynomial $p(x)$ which is monic and has integer coefficients.

It will be factored into a form:

$$ p(x) = g(x)*f_1(x)^p_1*f_2(x)^p_2 * ... * f_m(x)^p_m $$

where all factors $f_i(x)$ are monic also.

The algorithm starts by setting up a test polynomial, $p_test(x)$ which divides $p(x)$, but has the property that

$$ p_test(x) = g(x)*f_1(x)*f_2(x) * ... * f_m(x) $$

Such a polynomial is said to be square-free. It has the same factors as the original polynomial, but the original might have multiple of each factor, where $p_test(x)$ does not.

The square-free part of a polynomial can be obtained as follows:

$$ p_test(x) = p(x)/Gcd(p(x),D(x)p(x)) $$

It can be seen by simply writing this out that $p(x)$ and $D(x)p(x)$ will have factors $f_i(x)^(p_i-1)$ in common. these can thus be divided out.

It is not a requirement of the algorithm that the algorithm being worked with is square-free, but it speeds up computations to work with the square-free part of the polynomial if the only thing sought after is the set of factors. The multiplicity of the factors can be determined using the original $p(x)$.

Binary factoring then proceeds by trying to find potential solutions modulo $p=2$ first. There can only be two such solutions: $x+0$ and $x+1$.

A list of possible solutions L is set up with potential solutions.

Determining factors modulo 2^n given a factorization modulo 2

At this point there is a list L with solutions modulo 2^n for some n. The solutions will be of the form: $x+a$. The first step is to determine if any of the elements in L divides $p(x)$ (not modulo any integer). Since $x+a$ divides $p_test(x)$ modulo 2^n, both $x+a$ and $x+a-2^n$ have to be checked.

If an element in L divides $p_test(x)$, $p_test(x)$ is divided by it, and a loop is entered to test how often it divides $p(x)$ to determine the multiplicity p_i of the factor. The found factor $f_i(x) = x+c_i$ is added as a combination ($x+c_i$, p_i). $p(x)$ is divided by $f_i(x)^p_i$.

At this point there is a list L of factors that divide $p_test(x)$ modulo 2^n. This implies that for each of the elements u in L, either u or $u+2^n$ should divide $p_test(x)$ modulo $2^(n+1)$. The following step is thus to set up a new list with new elements that divide $p_test(x)$ modulo $2^(n+1)$.

The loop is re-entered, this time doing the calculation modulo $2^(n+1)$ instead of modulo 2^n.

The loop is terminated if the number of factors found equals $deg(p_test(x))$, or if 2^n is larger than the smallest non-zero coefficient of $p_test(x)$ as this smallest non-zero coefficient is the product of all the smallest non-zero coefficients of the factors, or if the list of potential factors is zero.

The polynomial $p(x)$ can not be factored any further, and is added as a factor ($p(x)$, 1).

The function {BinaryFactors}, when implemented, yields the following interaction in Yacas:

```
In> BinaryFactors((x+1)^4*(x-3)^2)
Out> {{x-3,2},{x+1,4}}
In> BinaryFactors((x-1/5)*(2*x+1/3))
Out> {{2,1},{x-1/5,1},{x+1/6,1}}
In> BinaryFactors((x-1123125)*(2*x+123233))
Out> {{2,1},{x-1123125,1},{x+123233/2,1}}
```

The binary factoring algorithm starts with a factorization modulo 2, and then each time tries to guess the next bit of the solution, maintaining a list of potential solutions. This list can grow exponentially in certain instances. For instance, factoring $(x-a)*(x-2*a)*(x-3*a)* ... $ implies a that the roots have common factors. There are inputs where the number of potential solutions (almost) doubles with each iteration. For these inputs the algorithm becomes exponential. The worst-case performance is therefore exponential. The list of potential solutions while iterating will contain a lot of false roots in that case.

Efficiently deciding if a polynomial divides another

Given the polynomial $p(x)$, and a potential divisor $$ f_i(x) = x-p $$ modulo some $q=2^n$ an expression for the remainder after division is

$$ rem(p)=Sum(i,0,n,a_i*p^i) $$

For the initial solutions modulo 2, where the possible solutions are x and $x-1$. For $p=0$, $rem(0) = a_0$. For $p=1$, $rem(1) = Sum(i,0,n,a_i)$.

Given a solution $x-p$ modulo $q=2^n$, we consider the possible solutions $Mod(x-p,2^(n+1))$ and $Mod(x-(p+2^n),2^n+1)$.

$x-p$ is a possible solution if $Mod(rem(p),2^(n+1)) = 0$.

$x-(p+q)$ is a possible solution if $Mod(rem(p+q),2^(n+1)) = 0$. Expanding $Mod(rem(p+q),2*q)$ yields:

$$ Mod(rem(p+q),2*q) = Mod(rem(p) + extra(p,q),2*q) $$

When expanding this expression, some terms grouped under $extra(p,q)$ have factors like $2*q$ or q^2. Since $q=2^n$, these terms vanish if the calculation is done modulo $2^(n+1)$.

The expression for $extra(p,q)$ then becomes $$ extra(p,q) = q*Sum(i,1,n/2, (2*i-1)*a(2*i)*p^(2*i-2)) $$

An efficient approach to determining if $x-p$ or $x-(p+q)$ divides $p(x)$ modulo $2^(n+1)$ is then to first calculate $Mod(rem(p),2*q)$. If this is zero, $x-p$ divides $p(x)$. In addition, if $Mod(rem(p)+extra(p,q),2*q)$ is zero, $x-(p+q)$ is a potential candidate.

Other efficiencies are derived from the fact that the operations are done in binary. Eg. if $q=2^n$, then $q_next=2^(n+1) = 2*q = q<<1 $ is used in the next iteration. Also, calculations modulo 2^n are equivalent to performing a bitwise and with 2^n-1. These operations can in general be performed efficiently on todays hardware which is based on binary representations.

Extending the algorithm

Only univariate polynomials with rational coefficients have been considered so far. This could be extended to allow for roots that are complex numbers $a+I*b$ where both a and b are rational numbers.

For this to work the division algorithm would have to be extended to handle complex numbers with integer a and b modulo some integer, and the initial setup of the potential solutions would have to be extended to try $x+1+I$ and $x+I$ also. The step where new potential solutions modulo $2^(n+1)$ are determined should then also test for $x+I*2^n$ and $x+2^n+I*2^n$.

The same extension could be made for multivariate polynomials, although setting up the initial irreducible polynomials that divide $p_test(x)$ modulo 2 might become expensive if done on a polynomial with many variables ($2^(2^m-1)$ trials for m variables).

Lastly, polynomials with real-valued coefficients could be factored, if the coefficients were first converted to rational numbers. However, for real-valued coefficients there exist other methods (Sturm sequences).

Newton iteration

What the {BinaryFactor} algorithm effectively does is finding a set of potential solutions modulo $2^(n+1)$ when given a set of potential solutions modulo 2^n. There is a better algorithm that does something similar: Hensel lifting. Hensel lifting is a generalized form of Newton iteration, where given a factorization modulo p, each iteration returns a factorization modulo p^2.

Newton iteration is based on the following idea: when one takes a Taylor series expansion of a function:

$$f(x[0]+dx) := f(x[0]) + (D(x)f(x[0]))*dx + ... $$

Newton iteration then proceeds by taking only the first two terms in this series, the constant plus the constant times dx. Given some good initial value x_0, the function will is assumed to be close to a root, and the function is assumed to be almost linear, hence this approximation. Under these assumptions, if we want $f(x_0+dx)$ to be zero,

$$f(x[0]+dx) = f(x[0]) + (D(x)f(x[0]))*dx = 0 $$

This yields:

$$ dx := -f(x[0])/(D(x)f(x[0])) = 0 $$

And thus a next, better, approximation for the root is $x[1]:=x_0-f(x[0])/(D(x)f(x[0]))$, or more general:

$$ x[n+1] =x[n]-f(x[n])/(D(x)f(x[n]))$$

If the root has multiplicity one, a Newton iteration can converge quadratically, meaning the number of decimals precision for each iteration doubles.

As an example, we can try to find a root of $Sin(x)$ near 3, which should converge to Pi.

Setting precision to 30 digits,:

```
In> Builtin'Precision'Set(30)
Out> True;
```

We first set up a function $dx(x)$:

```
In> dx(x):=Eval(-Sin(x)/(D(x)Sin(x)))
Out> True;
```

And we start with a good initial approximation to Pi, namely 3. Note we should set {x} after we set dx(x), as the right hand side of the function definition is evaluated. We could also have used a different parameter name for the definition of the function $dx(x)$.:

```
In> x:=3
Out> 3;
```

We can now start the iteration:

```
In> x:=N(x+dx(x))
Out> 3.14254654307427780529563541053 4;
In> x:=N(x+dx(x))
Out> 3.14159265330047681544988577172;
In> x:=N(x+dx(x))
Out> 3.14159265358979323846264338328 7;
In> x:=N(x+dx(x))
Out> 3.14159265358979323846264338328;
In> x:=N(x+dx(x))
Out> 3.14159265358979323846264338328;
```

As shown, in this example the iteration converges quite quickly.

Finding roots of multiple equations in multiple variables using Newton iteration

One generalization, mentioned in W.H. Press et al., NUMERICAL RECIPES in C, The Art of Scientific computing is finding roots for multiple functions in multiple variables.

Given N functions in N variables, we want to solve

$$ f_i(x[1],...,x[N]) = 0 $$

for $i = 1 .. N $. If de denote by X the vector $$ X := {x[1],x[2],...,x[N]} $$

and by dX the delta vector, then one can write

$$ f_i(X+dX) := f_i(X)+Sum(j,1,N,(D(x_j)f_i(X)))*dx[j] $$

Setting $f_i(X+dX)$ to zero, one obtains

$$ Sum(j,1,N,a[i][j]*dx_j)=b[i]$$

where

$$a[i][j] := D(x_j)f_i(X)$$

and

$$b_i := -f_i(X)$$

So the generalization is to first initialize X to a good initial value, calculate the matrix elements $a[i][j]$ and the vector $b[i]$, and then to proceed to calculate dX by solving the matrix equation, and calculating

$$X[i+1] := X[i] + dX[i]$$

In the case of one function with one variable, the summation reduces to one term, so this linear set of equations was a lot simpler in that case. In this case we will have to solve this set of linear equations in each iteration.

As an example, suppose we want to find the zeroes for the following two functions:

$$f_1(a,x) := Sin(a*x)$$

and

$$f_2(a,x) := a-2$$

It is clear that the solution to this is $a=2$ and $x:=N*Pi/2$ for any integer value N.

We will do calculations with precision 30:

```
In> Builtin'Precision'Set(30)
Out> True;
```

And set up a vector of functions ${f_1(X),f_2(X)}$ where $X:={a,x}$:

```
In> f(a,x):={Sin(a*x),a-2}
Out> True;
```

Now we set up a function {matrix(a,x)} which returns the matrix $a[i][j]$:

```
In> matrix(a,x):=Eval({D(a)f(a,x),D(x)f(a,x)})
Out> True;
```

We now set up some initial values:

```
In> {a,x}:={1.5,1.5}
Out> {1.5,1.5};
```

The iteration converges a lot slower for this example, so we will loop 100 times:

```
In> For(ii:=1,ii<100,ii++)[{a,x}:={a,x}+\
      N(SolveMatrix(matrix(a,x),-f(a,x)));]
Out> True;
In> {a,x}
Out> {2.,0.059667311457823162437151576236};
```

The value for a has already been found. Iterating a few more times:

```
In> For(ii:=1,ii<100,ii++)[{a,x}:={a,x}+\
      N(SolveMatrix(matrix(a,x),-f(a,x)));]
Out> True;
In> {a,x}
Out> {2.,-0.042792753588155918852832259721};
In> For(ii:=1,ii<100,ii++)[{a,x}:={a,x}+\
         N(SolveMatrix(matrix(a,x),-f(a,x)));]
Out> True;
In> {a,x}
Out> {2.,0.035119151349413516969586788023};
```

the value for x converges a lot slower this time, and to the uninteresting value of zero (a rather trivial zero of this set of functions). In fact for all integer values N the value $N*Pi/2$ is a solution. Trying various initial values will find them.

Newton iteration on polynomials

von zur Gathen et al., Modern Computer algebra discusses taking the inverse of a polynomial using Newton iteration. The task is, given a polynomial $f(x)$, to find a polynomial $g(x)$ such that $f(x) = 1/g(x)$, modulo some power in x. This implies that we want to find a polynom g for which:

$$h(g) = 1/g-f = 0$$

Applying a Newton iteration step $g[i+1] = g[i] - h(g[i])/(D(g)h(g[i]))$ to this expression yields:

$$g[i+1] = 2*g[i] - f*(g[i])^2$$

von zur Gathen then proves by induction that for $f(x)$ monic, and thus $f(0)=1$, given initial value $g_0(x) = 1$, that

$$Mod(f*g_i=1,x^(2^i))$$

Example:

suppose we want to find the polynomial $g(x)$ up to the 7th degree for which $Mod(f(x)*g(x) = 1,x^8)$, for the function

$$ f(x):=1+x+x^2/2+x^3/6+x^4/24 $$

First we define the function f:

```
In> f:=1+x+x^2/2+x^3/6+x^4/24
Out> x+x^2/2+x^3/6+x^4/24+1;
```

And initialize g and i.:

```
In> g:=1
Out> 1;
In> i:=0
Out> 0;
```

Now we iterate, increasing i, and replacing g with the new value for g:

```
In> [i++;g:=BigOh(2*g-f*g^2,x,2^i);]
Out> 1-x;
In> [i++;g:=BigOh(2*g-f*g^2,x,2^i);]
Out> x^2/2-x^3/6-x+1;
In> [i++;g:=BigOh(2*g-f*g^2,x,2^i);]
Out> x^7/72-x^6/72+x^4/24-x^3/6+x^2/2-x+1;
```

The resulting expression must thus be:

$$g(x):=x^7/72-x^6/72+x^4/24-x^3/6+x^2/2-x+1$$

We can easily verify this:

```
In> Expand(f*g)
Out> x^11/1728+x^10/576+x^9/216+(5*x^8)/576+1;
```

This expression is 1 modulo x^8, as can easily be shown:

```
In> BigOh(%,x,8)
Out> 1;
```

CHAPTER 6

YAGY

CHAPTER 7

Credits *[0]

*

7.1 Original/primary authors

Ayal Pinkus *apinkus "AT" xs4all "DOT" nl* This project was started by Ayal Pinkus who remains the main author and the primary maintainer.

Serge Winitzki *serge "AT" cosmos "DOT" phy "DOT" tufts "DOT" edu* Added factorials over rationals, TeX-Form, did a major overhaul of the introduction manual (actually, he wrote large part of the manual as it is), and initiated numerous improvements and test code for Yacas, and implemented yacas_client. Actually, Serge has been one of the larger contributors, and the main force behind the improved documentation.

Jitse Niesen *jn221 "AT" damtp "DOT" cam "DOT" ac "DOT" uk* Reported some bugs, helped improve various parts of Yacas, and greatly improved the manual for Yacas.

7.2 Maintainer

Grzegorz Mazur *teoretyk "AT" gmail "DOT" com*

7.3 Contributors

Jim Apple *japple "AT" freeshell "DOT" org* Reported bugs and supplied improved code for gcc 3.3.4

Mark Arrasmith *arrasmith "AT" math "DOT" twsu "DOT" edu* Helped greatly in setting up the fltk-based graphicaluser interface, and fixed some bugs relating to limits regarding infinity.

Fred Bacon *bacon "AT" aerodyne "DOT" com* Fixed some compiler errors on the newer gcc compiles. Reported some important bugs.

Jay Belanger *belanger "AT" truman "DOT" edu* Reported some bugs and improved some of the GnuPlot code. He also wrote the yacas.el file, which allows you to run yacas from within emacs.

Roberto Colistete Junior Is maintaining a version of Yacas for SymbianOS.

Sebastian Ferraro *sferraro "AT" criba "DOT" edu "DOT" ar* Reported bugs and supplied improved code (determinants).

[0] All with last-known email addresses mangled in an obvious way

John Fremlin Added some code for fast calculation of roots of a cubic polynomial.

Peter Gilbert *peterdgilbert "AT" gmail "DOT" com* Made many improvements to the C++ code to make it conform more to standard C++ coding conventions (class interfaces looking more like stl), improved the regression test suite.

James Gilbertson *azurite "AT" telusplanet "DOT" net* Win32 port, improved error reporting. Added initial version of Karatsuba multiplication, and added some matrix functions to the math library.

Gabor Grothendieck Gabor is the maintainer of Ryacas, and gave valuable feedback on the new web site.

Rene Grothmann *2004 "AT" rene-grothmann "DOT" de* Married Euler to Yacas.

Franz Hack *franz.hack "AT" web "DOT" de* Supplied a Delphi interface to the Yacas DLL.

Ingrid Halters Helped improve the ease of use of the Yacas web site.

Mark Hatsell *mark "AT" autograph-maths "DOT" com* Made the server code work on Windows.

Joris van der Hoeven *TeXmacs "AT" math "DOT" u-psud "DOT" fr* Helped with texmacs support.

Wolfgang Hšnig *pocket_software "AT" web "DOT" de* Created a port of Yacas that runs on PocketPC, to be found here.

Daniel Richard G. *straker "AT" MIT "DOT" EDU* Added autoconf/automake scripts, made Sun/Sgi compilation possible, created a rpm spec file, many many many changes to clean up the source distribution.

Igor Khavkine Added 'Diverge' and 'Curl', and implemented threading for the derivative operator (the gradient). Fixed GMP code.

John Lapeyre Made some modifications to the make file, and improved some math code.

Jonathan Leto *jonathan "AT" leto "DOT" net* Helped improve the integration algorithm, and helped extend the tests used for Yacas (finding numerous bugs).

Vladimir Livshits *livshits "AT" cs "DOT" stanford "DOT" edu* Set up the initial sourceforge CVS repository, and updated the Windows version source code. He also greatly improved the logic theorem prover code.

Eugenia Loli Helped build the BeOS version of Yacas.

Adolf Mathias *adolf_mathias "AT" web "DOT" de*

Grzegorz Mazur *teoretyk "AT" gmail "DOT" com*

Pablo De Nápoli *pdenapo "AT" yahoo "DOT" com* Fixed the configure script so Yacas compiles under cygwin.

Gopal Narayanan *gopal "AT" debian "DOT" org* Debian package maintainer. Made a man page for Yacas.

Marta Noga *marta.noga "AT" gmail "DOT" com*

Christian Obrecht *christian "DOT" obrecht "AT" wanadoo "DOT" fr* Made a much better Limit, and made Yacas behave better at infinity.

Alberto González Palomo Implemented a console-mode version of Yacas for AgendaVR. Changed the directory structure for the script files, and implemented initial support for OpenMath.

Doreen Pinkus *d "DOT" pinkus "AT" hccnet "DOT" nl* Designed the second version of the Web site for Yacas.

Mike Pinna *mike "AT" autograph-maths "DOT" com* Applied some bug fixes.

Savario Prinz *yacas "AT" mac "DOT" com* Built a fantastic Mac version of Yacas.

Dirk Reusch Added some linear algebra functions, and fixed some predicate functions.

Daniel Rigby Brought a client-server structure to the EPOC32 version of Yacas.

Juan Pablo Romero *jpablo_romero "AT" hotmail "DOT" com* Reported many bugs, made many suggestions for improvements, and supplied improved code (yacas scripts and makefile code).

Robert V Schipper ***rvs "AT" achilles "DOT" nfia "DOT" org*** Ironed out a few bugs in Yacas.

Schneelocke Reported an important bug in numeric calculations.

HenSiong Tan *tan "AT" stat "DOT" psu "DOT" edu*

Yannick Versley ***yannick "AT" versley "DOT" de*** Sent some patches regarding bugs relating integration and differentiation.

Adrian V. *qwert2003 "AT" users "DOT" sourceforge "DOT" net*

Ladislav Zejda Supplied patches to make Yacas work on Dec Alpha's.

Andrei Zorine Started the body of statistics code.

CHAPTER 8

License

Yacas is Free Software – Free as in Freedom – so you can redistribute yacas or modify it under certain conditions. Yacas comes with ABSOLUTELY NO WARRANTY. See the *GNU Lesser General Public License (LGPL) version 2.1* or (at your discretion) any later version for the full conditions. Yacas documentation is distributed under the terms of *GNU Free Documentation License*.

8.1 GNU Lesser General Public License

```
                  GNU LESSER GENERAL PUBLIC LICENSE
                       Version 2.1, February 1999

 Copyright (C) 1991, 1999 Free Software Foundation, Inc.
 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL.  It also counts
 as the successor of the GNU Library Public License, version 2, hence
 the version number 2.1.]

                            Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
Licenses are intended to guarantee your freedom to share and change
free software--to make sure the software is free for all its users.

  This license, the Lesser General Public License, applies to some
specially designated software packages--typically libraries--of the
Free Software Foundation and other authors who decide to use it.  You
can use it too, but we suggest you first think carefully about whether
this license or the ordinary General Public License is the better
strategy to use in any particular case, based on the explanations below.

  When we speak of free software, we are referring to freedom of use,
not price.  Our General Public Licenses are designed to make sure that
you have the freedom to distribute copies of free software (and charge
for this service if you wish); that you receive source code or can get
it if you want it; that you can change the software and use pieces of
it in new free programs; and that you are informed that you can do
these things.
```

```
  To protect your rights, we need to make restrictions that forbid
distributors to deny you these rights or to ask you to surrender these
rights.  These restrictions translate to certain responsibilities for
you if you distribute copies of the library or if you modify it.

  For example, if you distribute copies of the library, whether gratis
or for a fee, you must give the recipients all the rights that we gave
you.  You must make sure that they, too, receive or can get the source
code.  If you link other code with the library, you must provide
complete object files to the recipients, so that they can relink them
with the library after making changes to the library and recompiling
it.  And you must show them these terms so they know their rights.

  We protect your rights with a two-step method: (1) we copyright the
library, and (2) we offer you this license, which gives you legal
permission to copy, distribute and/or modify the library.

  To protect each distributor, we want to make it very clear that
there is no warranty for the free library.  Also, if the library is
modified by someone else and passed on, the recipients should know
that what they have is not the original version, so that the original
author's reputation will not be affected by problems that might be
introduced by others.

  Finally, software patents pose a constant threat to the existence of
any free program.  We wish to make sure that a company cannot
effectively restrict the users of a free program by obtaining a
restrictive license from a patent holder.  Therefore, we insist that
any patent license obtained for a version of the library must be
consistent with the full freedom of use specified in this license.

  Most GNU software, including some libraries, is covered by the
ordinary GNU General Public License.  This license, the GNU Lesser
General Public License, applies to certain designated libraries, and
is quite different from the ordinary General Public License.  We use
this license for certain libraries in order to permit linking those
libraries into non-free programs.

  When a program is linked with a library, whether statically or using
a shared library, the combination of the two is legally speaking a
combined work, a derivative of the original library.  The ordinary
General Public License therefore permits such linking only if the
entire combination fits its criteria of freedom.  The Lesser General
Public License permits more lax criteria for linking other code with
the library.

  We call this license the "Lesser" General Public License because it
does Less to protect the user's freedom than the ordinary General
Public License.  It also provides other free software developers Less
of an advantage over competing non-free programs.  These disadvantages
are the reason we use the ordinary General Public License for many
libraries.  However, the Lesser license provides advantages in certain
special circumstances.

  For example, on rare occasions, there may be a special need to
encourage the widest possible use of a certain library, so that it becomes
a de-facto standard.  To achieve this, non-free programs must be
allowed to use the library.  A more frequent case is that a free
```

```
library does the same job as widely used non-free libraries.  In this
case, there is little to gain by limiting the free library to free
software only, so we use the Lesser General Public License.

  In other cases, permission to use a particular library in non-free
programs enables a greater number of people to use a large body of
free software.  For example, permission to use the GNU C Library in
non-free programs enables many more people to use the whole GNU
operating system, as well as its variant, the GNU/Linux operating
system.

  Although the Lesser General Public License is Less protective of the
users' freedom, it does ensure that the user of a program that is
linked with the Library has the freedom and the wherewithal to run
that program using a modified version of the Library.

  The precise terms and conditions for copying, distribution and
modification follow.  Pay close attention to the difference between a
"work based on the library" and a "work that uses the library".  The
former contains code derived from the library, whereas the latter must
be combined with the library in order to run.

                  GNU LESSER GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. This License Agreement applies to any software library or other
program which contains a notice placed by the copyright holder or
other authorized party saying it may be distributed under the terms of
this Lesser General Public License (also called "this License").
Each licensee is addressed as "you".

  A "library" means a collection of software functions and/or data
prepared so as to be conveniently linked with application programs
(which use some of those functions and data) to form executables.

  The "Library", below, refers to any such software library or work
which has been distributed under these terms.  A "work based on the
Library" means either the Library or any derivative work under
copyright law: that is to say, a work containing the Library or a
portion of it, either verbatim or with modifications and/or translated
straightforwardly into another language.  (Hereinafter, translation is
included without limitation in the term "modification".)

  "Source code" for a work means the preferred form of the work for
making modifications to it.  For a library, complete source code means
all the source code for all modules it contains, plus any associated
interface definition files, plus the scripts used to control compilation
and installation of the library.

  Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope.  The act of
running a program using the Library is not restricted, and output from
such a program is covered only if its contents constitute a work based
on the Library (independent of the use of the Library in a tool for
writing it).  Whether that is true depends on what the Library does
and what the program that uses the Library does.

  1. You may copy and distribute verbatim copies of the Library's
```

```
complete source code as you receive it, in any medium, provided that
you conspicuously and appropriately publish on each copy an
appropriate copyright notice and disclaimer of warranty; keep intact
all the notices that refer to this License and to the absence of any
warranty; and distribute a copy of this License along with the
Library.

  You may charge a fee for the physical act of transferring a copy,
and you may at your option offer warranty protection in exchange for a
fee.

  2. You may modify your copy or copies of the Library or any portion
of it, thus forming a work based on the Library, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:

    a) The modified work must itself be a software library.

    b) You must cause the files modified to carry prominent notices
    stating that you changed the files and the date of any change.

    c) You must cause the whole of the work to be licensed at no
    charge to all third parties under the terms of this License.

    d) If a facility in the modified Library refers to a function or a
    table of data to be supplied by an application program that uses
    the facility, other than as an argument passed when the facility
    is invoked, then you must make a good faith effort to ensure that,
    in the event an application does not supply such function or
    table, the facility still operates, and performs whatever part of
    its purpose remains meaningful.

    (For example, a function in a library to compute square roots has
    a purpose that is entirely well-defined independent of the
    application.  Therefore, Subsection 2d requires that any
    application-supplied function or table used by this function must
    be optional: if the application does not supply it, the square
    root function must still compute square roots.)

These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Library,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Library, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote
it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Library.

In addition, mere aggregation of another work not based on the Library
with the Library (or with a work based on the Library) on a volume of
a storage or distribution medium does not bring the other work under
```

```
the scope of this License.

  3. You may opt to apply the terms of the ordinary GNU General Public
License instead of this License to a given copy of the Library.  To do
this, you must alter all the notices that refer to this License, so
that they refer to the ordinary GNU General Public License, version 2,
instead of to this License.  (If a newer version than version 2 of the
ordinary GNU General Public License has appeared, then you can specify
that version instead if you wish.)  Do not make any other change in
these notices.

  Once this change is made in a given copy, it is irreversible for
that copy, so the ordinary GNU General Public License applies to all
subsequent copies and derivative works made from that copy.

  This option is useful when you wish to copy part of the code of
the Library into a program that is not a library.

  4. You may copy and distribute the Library (or a portion or
derivative of it, under Section 2) in object code or executable form
under the terms of Sections 1 and 2 above provided that you accompany
it with the complete corresponding machine-readable source code, which
must be distributed under the terms of Sections 1 and 2 above on a
medium customarily used for software interchange.

  If distribution of object code is made by offering access to copy
from a designated place, then offering equivalent access to copy the
source code from the same place satisfies the requirement to
distribute the source code, even though third parties are not
compelled to copy the source along with the object code.

  5. A program that contains no derivative of any portion of the
Library, but is designed to work with the Library by being compiled or
linked with it, is called a "work that uses the Library".  Such a
work, in isolation, is not a derivative work of the Library, and
therefore falls outside the scope of this License.

  However, linking a "work that uses the Library" with the Library
creates an executable that is a derivative of the Library (because it
contains portions of the Library), rather than a "work that uses the
library".  The executable is therefore covered by this License.
Section 6 states terms for distribution of such executables.

  When a "work that uses the Library" uses material from a header file
that is part of the Library, the object code for the work may be a
derivative work of the Library even though the source code is not.
Whether this is true is especially significant if the work can be
linked without the Library, or if the work is itself a library.  The
threshold for this to be true is not precisely defined by law.

  If such an object file uses only numerical parameters, data
structure layouts and accessors, and small macros and small inline
functions (ten lines or less in length), then the use of the object
file is unrestricted, regardless of whether it is legally a derivative
work.  (Executables containing this object code plus portions of the
Library will still fall under Section 6.)

  Otherwise, if the work is a derivative of the Library, you may
```

```
distribute the object code for the work under the terms of Section 6.
Any executables containing that work also fall under Section 6,
whether or not they are linked directly with the Library itself.

  6. As an exception to the Sections above, you may also combine or
link a "work that uses the Library" with the Library to produce a
work containing portions of the Library, and distribute that work
under terms of your choice, provided that the terms permit
modification of the work for the customer's own use and reverse
engineering for debugging such modifications.

  You must give prominent notice with each copy of the work that the
Library is used in it and that the Library and its use are covered by
this License.  You must supply a copy of this License.  If the work
during execution displays copyright notices, you must include the
copyright notice for the Library among them, as well as a reference
directing the user to the copy of this License.  Also, you must do one
of these things:

    a) Accompany the work with the complete corresponding
    machine-readable source code for the Library including whatever
    changes were used in the work (which must be distributed under
    Sections 1 and 2 above); and, if the work is an executable linked
    with the Library, with the complete machine-readable "work that
    uses the Library", as object code and/or source code, so that the
    user can modify the Library and then relink to produce a modified
    executable containing the modified Library.  (It is understood
    that the user who changes the contents of definitions files in the
    Library will not necessarily be able to recompile the application
    to use the modified definitions.)

    b) Use a suitable shared library mechanism for linking with the
    Library.  A suitable mechanism is one that (1) uses at run time a
    copy of the library already present on the user's computer system,
    rather than copying library functions into the executable, and (2)
    will operate properly with a modified version of the library, if
    the user installs one, as long as the modified version is
    interface-compatible with the version that the work was made with.

    c) Accompany the work with a written offer, valid for at
    least three years, to give the same user the materials
    specified in Subsection 6a, above, for a charge no more
    than the cost of performing this distribution.

    d) If distribution of the work is made by offering access to copy
    from a designated place, offer equivalent access to copy the above
    specified materials from the same place.

    e) Verify that the user has already received a copy of these
    materials or that you have already sent this user a copy.

  For an executable, the required form of the "work that uses the
Library" must include any data and utility programs needed for
reproducing the executable from it.  However, as a special exception,
the materials to be distributed need not include anything that is
normally distributed (in either source or binary form) with the major
components (compiler, kernel, and so on) of the operating system on
which the executable runs, unless that component itself accompanies
```

```
the executable.

  It may happen that this requirement contradicts the license
restrictions of other proprietary libraries that do not normally
accompany the operating system.  Such a contradiction means you cannot
use both them and the Library together in an executable that you
distribute.

  7. You may place library facilities that are a work based on the
Library side-by-side in a single library together with other library
facilities not covered by this License, and distribute such a combined
library, provided that the separate distribution of the work based on
the Library and of the other library facilities is otherwise
permitted, and provided that you do these two things:

    a) Accompany the combined library with a copy of the same work
    based on the Library, uncombined with any other library
    facilities.  This must be distributed under the terms of the
    Sections above.

    b) Give prominent notice with the combined library of the fact
    that part of it is a work based on the Library, and explaining
    where to find the accompanying uncombined form of the same work.

  8. You may not copy, modify, sublicense, link with, or distribute
the Library except as expressly provided under this License.  Any
attempt otherwise to copy, modify, sublicense, link with, or
distribute the Library 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.

  9. You are not required to accept this License, since you have not
signed it.  However, nothing else grants you permission to modify or
distribute the Library or its derivative works.  These actions are
prohibited by law if you do not accept this License.  Therefore, by
modifying or distributing the Library (or any work based on the
Library), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Library or works based on it.

  10. Each time you redistribute the Library (or any work based on the
Library), the recipient automatically receives a license from the
original licensor to copy, distribute, link with or modify the Library
subject to these terms and conditions.  You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties with
this License.

  11. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Library at all.  For example, if a patent
license would not permit royalty-free redistribution of the Library by
```

```
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any
particular circumstance, the balance of the section is intended to apply,
and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system which is
implemented by public license practices.  Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

  12. If the distribution and/or use of the Library is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Library under this License may add
an explicit geographical distribution limitation excluding those countries,
so that distribution is permitted only in or among countries not thus
excluded.  In such case, this License incorporates the limitation as if
written in the body of this License.

  13. The Free Software Foundation may publish revised and/or new
versions of the Lesser General Public 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.

Each version is given a distinguishing version number.  If the Library
specifies a version number of this License which applies to it and
"any later version", you have the option of following the terms and
conditions either of that version or of any later version published by
the Free Software Foundation.  If the Library does not specify a
license version number, you may choose any version ever published by
the Free Software Foundation.

  14. If you wish to incorporate parts of the Library into other free
programs whose distribution conditions are incompatible with these,
write to the author to ask for permission.  For software which is
copyrighted by the Free Software Foundation, write to the Free
Software Foundation; we sometimes make exceptions for this.  Our
decision will be guided by the two goals of preserving the free status
of all derivatives of our free software and of promoting the sharing
and reuse of software generally.

                            NO WARRANTY

  15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO
WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW.
EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR
OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY
```

KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE
LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME
THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN
WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY
AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU
FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR
CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE
LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING
RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A
FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF
SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGES.

                     END OF TERMS AND CONDITIONS

           How to Apply These Terms to Your New Libraries

  If you develop a new library, and you want it to be of the greatest
possible use to the public, we recommend making it free software that
everyone can redistribute and change. You can do so by permitting
redistribution under these terms (or, alternatively, under the terms of the
ordinary General Public License).

  To apply these terms, attach the following notices to the library. It is
safest to attach them to the start of each source file to most effectively
convey the exclusion of warranty; and each file should have at least the
"copyright" line and a pointer to where the full notice is found.

    <one line to give the library's name and a brief idea of what it does.>
    Copyright (C) <year> <name of author>

    This library is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This library is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this library; if not, write to the Free Software
    Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the library, if
necessary. Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the
  library `Frob' (a library for tweaking knobs) written by James Random Hacker.

```
  <signature of Ty Coon>, 1 April 1990
  Ty Coon, President of Vice

That's all there is to it!'
```

8.2 GNU Free Documentation License

```
                GNU Free Documentation License
                   Version 1.1, March 2000

 Copyright (C) 2000  Free Software Foundation, Inc.
     51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.


0. 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.


1. 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.


2. 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.


3. 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.


4. 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:

A. 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.
B. 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).
C. State on the Title page the name of the publisher of the
   Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications
   adjacent to the other copyright notices.
F. 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.
G. Preserve in that license notice the full lists of Invariant Sections
   and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. 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.
J. 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.
K. 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.
L. 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.
M. Delete any section entitled "Endorsements".  Such a section
   may not be included in the Modified Version.
N. 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.


5. 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."


6. 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.


7. 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.


8. 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.


9. 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.


10. 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.
```

CHAPTER 9

Glossary

arity Arity is the number of arguments of a function. For example, the function {Cos(x)} has one argument and so we say that "{Cos} has arity 1". Arity of a function can be 0, 1, 2, ...

{Yacas} allows to define functions with the same name but different arities, and different rules corresponding to these arities will be used. Also, it is possible to define a function with optional arguments, for example, {Plot2D} is one such function. Such functions can have any arity larger or equal to a certain minimum arity (the number of non-optional arguments).

See also:

Function(), *OpPrecedence()*, *Rule()*

array An array is a container object that can hold a fixed number of other Yacas objects in it. Individual elements of an array can be accessed using the {[]} operation. Most list operations also work on arrays.

Arrays are faster than lists but the array size cannot be changed.

See also:

Array'Create()

atom Atoms are basic Yacas objects that are used to represent symbols, numbers, and function names. An atom has a string representation which is shown when it is displayed. For example, {3.14159}, {x}, {A123}, {+}, {"good morning"} are atoms.

Atoms can be of type string, number, or symbol. For example, {y1} is a symbolic atom, {954} is a number atom, and {""} is an (empty) string atom. Symbolic atoms are normally used in {Yacas} to denote mathematical unknowns and function names. Number and string atoms are used to denote values.

A symbolic atom can be bound to a value (in which case it becomes a variable), or to a rule or several rules (in which case it becomes a function). An atom can also have a property object.

See also:

Atom(), *String()*

bodied function *Bodied functions* have all arguments except the first one inside parentheses and the first argument outside the argument list, for example:

```
Integrate(x) Sin(x);
```

CAS Abbreviation for "computer algebra system". {Yacas} is a CAS.

constant, cached constant Constants such as *Pi* or *GoldenRatio* are symbolic atoms that are specially interpreted by yacas. For example, there are simplification rules that transform expressions such as `Sin(Pi)` into `0`. When requesting a numerical evaluation of a constant, the numerical value is given to the current value as set with *N()*.

Some constants take a long time to compute and therefore they are cached at the highest precision computed so far. These are the *cached constants*.

See also:

N(), *CachedConstant()*, *Pi*, *GoldenRatio*, CatalanConstant, *gamma*

equation To denote symbolic equations, the operator {==} is used. This operator does not assign or compare its sides. For example, the expression {Sin(x)==1} is kept unevaluated and can be passed as argument to functions. For example, In> Solve(Sin(x)==1, x) Out> {x==Pi/2};

The symbolic equation operator {==} is also useful to represent solutions of equations or to specify substitutions, give options, and so on.

See also:

Solve(), *Where()*, *Plot2D()*

function A function is a symbolic atom that is bound to a rule or several rules. A function can have none, one, or more arguments. Functions can also have a variable number of arguments. Arguments of functions are arbitrary Yacas objects.

Functions can be evaluated, that is, the rules bound to them may be executed. For example, `Cos(Pi+0)` is an expression that contains two functions and four atoms. The atom `Pi` is a symbolic atom which is normally not bound to anything. The atom `0` is a numeric atom.

The atoms `Cos` and `+` are symbolic atoms which are bound to appropriate simplification rules. So these two atoms are functions. Note that these functions have different syntax. `Cos` is a normal function which takes its arguments in parentheses. The atom `+` is a function with special syntax because `+` is placed between its arguments and no parentheses are used.

The rules to which `+` is bound are such that the expression `Pi+0` is evaluated to the symbolic atom `Pi`. The rules for `Cos` are such that the expression `Cos(Pi)` is evaluated to the numeric atom `-1`. The example {Yacas} session is:

```
In> Cos(Pi+0)
Out> -1
```

Some functions are built-in and implemented in C++, while others are library functions.

The built-in functions are usually less flexible than the library functions because they cannot be left unevaluated. Given incorrect arguments, a built-in function will generate an error. However, a user-defined function may simply return unevaluated in such cases.

See also:

Function(), *Rule()*, `<--()`

list A list is a basic {Yacas} container object. A list is written as e.g. {{a, b, c}} or {{}} (empty list). Individual elements of a list can be accessed using the {[]} operation. Lists can be concatenated, and individual elements can be removed or inserted.

Lists are ubiquitous in {Yacas}. Most data structures in the standard library is based on lists.

Lists are also used internally to represent {Yacas} expressions. For example, the expression {Cos(x+1)} is represented internally as a nested list: In> FullForm(Cos(x+1)) (Cos (+ x 1)) Out> Cos(x+1);

See also:

List(), *Listify()*, *UnList()*, *Length()*, *FullForm()*

matrix A matrix is represented as a list of lists. Matrices are represented in the "row-major" order: a matrix is a list of rows, and each row is a list of its elements.

Some basic linear algebra operations on matrices are supported.

See also:

Determinant(), *Identity()*, *IsDiagonal()*, *EigenValues()*

operator Operators are functions that have special syntax declared for them. An operator can be "bodied", infix, prefix or postfix. Because of this, operators must have precedence.

Apart from the syntax, operators are exactly the same as any other functions, they can have rules bound to them in the same way.

See also:

Bodied(), *Infix()*, *Prefix()*, *Postfix()*

precedence Precedence is a property of the syntax of an operator that specifies how it is parsed. Only operators, i.e. functions with special syntax, can have precedence. Precedence values are nonnegative integers: 0, 1, ... Lower numbers bind more tightly.

For example, the operator "{+}" binds less tightly (i.e. has a higher precedence value) than the operator "{*}" and so the expression {a+b*c} is parsed as {a+(b*c)}, as one would expect.

Infix operators can have different left-side and right-side precedence. For example, the infix operator "{-}" has left precedence `*EVAL OpLeftPrecedence("-") and right precedence *EVAL OpRightPrecedence("-")` – this allows us to parse expressions such as {a-b+c} correctly, as $(a-b)+c$, and not as $a-(b+c)$.

See also:

Bodied(), *OpPrecedence()*, *OpLeftPrecedence()*, *OpRightPrecedence()*

property Properties are special additional objects (tags) that can be tied to expressions. For example, the expression {1+x} may be tagged by an expression {y} by the command

```
In> a:= ExtraInfo'Set(1+x,y);
Out> 1+x;
```

Now {a} refers to an expression {1+x} which is different from all other copies of {1+x} because it is tagged by {y}.

See also:

ExtraInfo'Get(), *ExtraInfo'Set()*

rule Rules are the principal mechanism of expression evaluation in {Yacas}. A rule specifies that a certain symbolic expression is to be replaced by another expression. If no rule that matches a given symbolic expression can be found, the expression is left unevaluated. This is usually the desired behavior for a CAS. For example, a user can type

```
In> func1(x+0)
Out> func1(x);
```

and use an undefined function {func1}. Since no rules are defined for the function {func1}, it is not evaluated, but its argument has been simplified.

Only expressions containing functions can be evaluated by rules. (Atoms are evaluated only if they are bound to a value.)

Several rules can be defined for a given function. Rules can be erased or added at run time.

See also:

Rule(), *<--()*, *Retract()*

string A string is an atom with character string value, for example, {"abcd"}. Individual characters of the string can be accessed using the {[]} operation. Some string manipulation functions are supported.

See also:

String(), *StringMid'Get()*, *StringMid'Set()*

syntax {Yacas} uses an infix syntax similar to C or Fortran. However, the syntax is entirely user-defined and very flexible. Infix, prefix, postfix operators can be defined, as well as "bodied" functions. This allows to write mathematical expressions more comfortably, for example

```
In> D(x) Sin(x)+1
Out> Cos(x);
```

Functions with special syntax can have different precedence.

See also:

Bodied(), *Infix()*, *Prefix()*, *Postfix()*, *OpPrecedence()*

threaded function Threaded function applied to a list

```
In> Cos({Pi/2, Pi/4})
Out> {0,Sqrt(1/2)}
```

variable Variables are symbolic atoms bound to a "value". Value is any Yacas object, such as an atom or a list. For example, after executing

```
In> a := 1
Out> 1;
```

the symbolic atom {a} becomes a variable bound to a value, the numeric atom {1}.

See also:

Eval(), *:=()*, *Clear()*

warranty {Yacas} is Free Software ("logiciel libre") and comes with {NO WARRANTY}. See *<the appropriate section of the GPL|yacasdoc://refprog/1/2/>* for full information.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z