

Introduction to Yacas: tutorial and examples

by the YACAS team ¹

YACAS version: 1.0.57
generated on November 28, 2006

This document gives a short introduction to Yacas. Included is a brief tutorial on the syntax and some commands to get you started using Yacas. There are also some examples.

¹This text is part of the YACAS software package. Copyright 2000–2002. Principal documentation authors: Ayal Zwi Pinkus, Serge Winitzki, Jitse Niesen. 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 no Invariant Sections, no Front-Cover Texts and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

Contents

1	Getting started with Yacas	2
1.1	Introduction	2
1.2	Installing YACAS	2
1.3	Using the console mode	2
1.4	YACAS as a symbolic calculator	3
1.5	Variables	4
1.6	Functions	4
1.7	Strings and lists	4
1.8	Linear Algebra	5
1.9	Control flow: conditionals, loops, blocks	5
2	Examples	6
2.1	Miscellaneous capabilities	6
2.2	A longer calculation with plotting	6
3	Let's learn some more	8
3.1	Using Yacas from the console	8
3.2	Compound statements	10
3.3	“Threading” of functions	10
3.4	Functions as lists	10
3.5	More on syntax	10
3.6	Writing simplification rules	11
3.7	Local simplification rules	12
4	GNU Free Documentation License	13

Chapter 1

Getting started with Yacas

1.1 Introduction

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

Currently, the YACAS programming language is stable and seems powerful enough for all computer algebra applications. External libraries providing additional functionality may be dynamically loaded into YACAS via the “plugin” mechanism.

1.2 Installing Yacas

Read the file `INSTALL` for instructions on how to compile YACAS. YACAS is portable across most Unix-ish platforms and requires only a standard C++ compiler such as `g++`.

The base YACAS application accepts text as input and returns text as output. This makes it rather platform-independent. Apart from Unix-like systems, YACAS has been compiled on Windows and on EPOC32, aka Psion (which doesn’t come with a standard C++ library!). The source code to compile YACAS for Windows can be found at the Sourceforge repository (Web URL: <http://sourceforge.net/projects/yacas/>).

For Unix, compilation basically amounts to the standard sequence

```
./configure
make
make install
```

This will install the binaries to `/usr/local/bin` and the library files to `/usr/local/share/yacas/`.

The arbitrary precision math in YACAS will be generally faster if you compile YACAS with the `libgmp` library (the option `--with-numlib=gmp` for the `configure` script). Precompiled Red Hat (RPM) and Debian (DEB) packages are also available.

Additionally, L^AT_EX-formatted documentation in PostScript and PDF formats can be produced by the command

```
make texdocs
```

or, alternatively, by passing `--enable-ps-doc` or `--enable-pdf-doc` to `./configure` when building YACAS.

In the latter case, the documentation will be automatically rebuilt every time the documentation changes (which is useful when maintaining the documentation).

1.3 Using the console mode

You can run YACAS in the console mode simply by typing `yacas`. The YACAS command prompt looks like this:

```
In>
```

and YACAS’s answers appear after the prompt

```
Out>
```

A YACAS session may be terminated by typing `Exit()` or `quit`. Pressing `^C` will also quit YACAS; however, pressing `^C` while YACAS is busy with a calculation will stop just that calculation. A session can be restarted (forgetting all previous definitions and results) by typing

```
restart
```

Typically, you would enter one statement per line, for example

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

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).

All documentation is accessible from the YACAS prompt. If you type

```
In> ??
```

you should be able to read all available manuals; YACAS will run `lynx` or another browser to show you the HTML documentation. You can also get help on individual functions: to read about the function `Sum()`, type

```
In> ?Sum
```

Type `Example()`; to get some random examples of YACAS calculations.

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.

Typing the first few characters of a previous expression and then hitting the TAB key makes YACAS recall the last expression in the history list that matches these first characters.

Commands spanning multiple lines can (and actually have to) be entered by using a trailing backslash at end of each continued line. For example:

```
In> a:=2+3+
Error on line 1 in file [CommandLine]
Line error occurred on:
>>>
Error parsing expression

In> a:=2+3+ \
In> 1
Out> 6;
```

The error after our first attempt occurred because YACAS has appended a semicolon at end of the first line and `2+3+;` is not a valid YACAS expression.

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

1.4 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:

```
In> 1/14+5/21*(30-(1+1/2)*5^2);
Out> -12/7;
```

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.

```
In> 0+x;
Out> x;
In> x+1*y;
Out> x+y;
In> Sin(ArcSin(alpha))+Tan(ArcTan(beta));
Out> alpha+beta;
In> (x+y)^3-(x-y)^3
Out> (x+y)^3-(x-y)^3;
In> Simplify(%)
Out> 6*x^2*y+2*y^3;
```

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 can deal with arbitrary precision numbers:

```
In> 20!;
Out> 2432902008176640000;
```

When dealing with floating point numbers, the command `Precision(n);` can be used to specify that all floating point numbers should have a fixed precision of `n` digits:

```
In> Precision(30);
Out> True;
In> N(1/243);
Out> 0.004115226337448559670781893004;
```

Note that we need to enter `N()` to force the approximate calculation, otherwise the fraction would have been left unevaluated. The value `True` is a boolean constant.

The `N` function has an optional second argument, the required precision:

```
In> N(1/234,10)
Out> 0.0042735042;
In> N(1/234,20)
Out> 0.0042735042735042735;
In> N(1/234,30)
Out> 0.004273504273504273504273504273;
```

Analytic derivatives of functions can be evaluated:

```
In> D(x) Sin(x);
Out> Cos(x);
In> D(x) D(x) Sin(x);
Out> -Sin(x);
```

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:

```
In> D(x,1)Sin(x)
Out> Cos(x);
In> D(x,2)Sin(x)
Out> -Sin(x);
```

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 π 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 imaginary unit i is denoted `I` and complex numbers can be entered as either expressions involving `I` or explicitly `Complex(a,b)` for $a + ib$.

Some simple equation solving algorithms are in place:

```
In> Solve(x/(1+x) == a, x);
Out> {x==a/(1-a)};
In> Solve(x^2+x == 0, x);
Out> {x==0,x==(-1)};
```

(Note the use of the `==` operator, which does not evaluate to anything, to denote an “equation” object.) Currently `Solve` is rather limited, but in the future there will be more sophisticated algorithms.

Taylor series are supported, for example:

```
In> Taylor(x,0,3) Exp(x)
Out> 1+x+(1/2)*x^2+(1/6)*x^3;
```

As this form of the answer may be a little bit hard to read, you might then type

```
In> PrettyForm(%);
      / 1 \    2    / 1 \    3
1 + x + | - | * x + | - | * x
      \ 2 /          \ 6 /

Out> True;
```

The function `PrettyForm()` tries to render the formula in a better format for reading, using ASCII text. You can also export an expression to \TeX by typing `TeXForm(...)`.

1.5 Variables

YACAS supports variables:

```
In> Set(a,Cos(0));
Out> True;
In> a:=a+1;
Out> 2;
```

The variable `a` has now been globally set to 2. The function `Set()` and the operator `:=` can both be used to assign values to global variables. (Variables local to procedures can also be defined; see below the chapters on programming.) To clear a variable binding, execute `Clear(a)`; "`a`" will now evaluate to just `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.

Currently there is no difference between assigning variables using `Set()` or using the operator `:=`. The latter can however also assign lists and define functions.

1.6 Functions

The `:=` operator can be used to define functions:

```
f(x):=2*x*x
```

will define a new function, `f`, that accepts one argument and returns twice the square of that argument.

One and the same function name such as `f` may be used by different functions if they take different numbers of arguments (but not if they merely take different *types* of arguments, since YACAS does not have a strict type system):

```
In> f(x):=x^2;
Out> True;
In> f(x,y):=x*y;
Out> True;
In> f(3)+f(3,2);
Out> 15;
```

Functions may return values of any type, or may even return values of different types at different times.

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 standard library:

```
In> IsNumber(2+x);
Out> False;
In> IsInteger(15/5);
Out> True;
```

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()`:

```
f(x):=Eval(x+x)
```

will first evaluate `x+x` to `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.

1.7 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. In fact, any YACAS expression can be converted to a list (see below).

Items in a list can be accessed through the `[]` operator. 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.

Another use of lists is the associative list, sometimes called a hash table, 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. As an example, first enter

```
u:={};
```

and then

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

Now, `u["name"]` would return `"Isaia"`. The list `u` now contains three sublists, as we can see:

```
In> u;
Out> { {"is alive", False}, {"occupation",
      "prophet"}, {"name", "Isaia"} };
```

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

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

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 expected.

Let's try some list operations now:

```
In> m:={a,b,c};
Out> True;
```

```
In> Length(m);
Out> 3;
```

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

```

In> Concat(m,m);
Out> {a,b,c,a,b,c};

In> m[1]:="blah blah";
Out> True;

In> m;
Out> {"blah blah",b,c};

In> Nth(m,2);
Out> b;

```

Many more list operations are described in the reference manual.

1.8 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 vector of vectors.

Vector components can be assigned values just like list items, since they are in fact list items:

```

In> l:=ZeroVector(3);
Out> True;

In> l;
Out> {0,0,0};

In> l[ 2 ]:=2;
Out> True;

In> l;
Out> {0,2,0};

```

YACAS can perform multiplication of matrices, vectors and numbers as usual in linear algebra:

```

In> v:={1,0,0,0}
Out> {1,0,0,0};

In> E4:={ {0,u1,0,0},{d0,0,u2,0},
          {0,d1,0,0},{0,0,d2,0}}
Out> {{0,u1,0,0},{d0,0,u2,0},
       {0,d1,0,0},{0,0,d2,0}};

In> CharacteristicEquation(E4,x)
Out> x^4-x*u2*d1*x-u1*d0*x^2;

In> Expand(%,x)
Out> x^4-(u2*d1+u1*d0)*x^2;

In> v+E4*v+E4*E4*v+E4*E4*E4*v
Out> {1+u1*d0,d0+(d0*u1+u2*d1)*d0,
      d1*d0,d2*d1*d0};

```

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 $Ax = b$ where A is a matrix, and x and b are vectors. There are several more matrix operations supported. See the reference manual for more details.

1.9 Control flow: conditionals, loops, blocks

The YACAS language includes some constructs and functions for control flow. Looping can be done with either a `ForEach()` or a `While()` function call. The function `ForEach(x, list)` `body` executes its `body` for each element of the list and assigns the variable `x` to that element each time. The function call `While(predicate) body` repeats the "body" until the "predicate" returns `False`.

Conditional execution is implemented by the `If(predicate, body1, body2)` function call, which works like the C language

`construct (predicate) ? body1 : body2`. If the condition is true, "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:

```
absx := If( x>=0, x, -x );
```

(The library function `Abs()` does this already.)

If several operations need to be executed in sequence to obtain a result, you can use a `Prog()` function call or equivalently the `[]` construct.

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. (What follows is not necessarily the most economical way to do it in YACAS.)

```

In> L := {};
Out> {};

In> i := 2;
Out> 2;

In> While(i<=20) [ L:= Append(L, i); \
                  i := i+2; ]
Out> True;

In> L;
Out> {2,4,6,8,10,12,14,16,18,20};

In> answer := 1;
Out> 1;

In> ForEach(i, L) If (Mod(i, 3)!=0, \
                    answer := answer * i);
Out> True;

In> answer;
Out> 2867200;

```

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`.

The above example is not the shortest possible way to write out the algorithm. A more 'functional' approach would go like this:

First 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.

```

In> 2*(1 .. 10)
Out> {2,4,6,8,10,12,14,16,18,20};

```

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:

```

In> Select({n},Mod(n,3)!=0,2*(1 .. 10))
Out> {2,4,8,10,14,16,20};

```

The numbers 6, 12 and 18 have been correctly filtered out.

All that remains is to factor the items in this list. For this we can use `Factorize`, which accepts a list as argument and returns the product of all the items in that list:

```

In> Factorize(Select({n},Mod(n,3)!=0,2*(1 .. 10)))
Out> 2867200;

```

A more flexible function one can use is `UnFlatten`:

```

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

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

```

Using `UnFlatten`, the result becomes:

```

In> UnFlatten(Select({n},Mod(n,3)!=0,2*(1 .. 10)),"*",1)
Out> 2867200;

```

Chapter 2

Examples

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.

Additional example calculations including the results can be found here:

- A selection of calculations from the *Wester benchmark*, in *Essays on Yacas, Chapter 2*.
- Some additional example calculations (Web URL: mybench2.html) that YACAS can currently perform.

2.1 Miscellaneous capabilities

```
100!;
```

Compute a large factorial using arbitrary precision integers.

```
ToBase(16,255);
FromBase(16,"2FF");
```

Convert between the decimal notation and another number base. (In Yacas, all numbers are written in decimal notation.)

```
Expand((1+x)^5);
```

Expand the expression into a polynomial.

```
Apply("+",{2,3});
```

Apply an operator to a list of arguments. This example would evaluate to 5.

```
Apply({{x,y},x+y},{2,3});
```

Apply a pure function to a list of arguments. This example would also evaluate to 5.

```
D(x)D(x) Sin(x);
```

Take derivatives of a function.

```
Solve(a+x*y==z,x);
```

Solve an equation for a variable.

```
Taylor(x,0,5) Sin(x);
```

Calculate the Taylor series expansion of a function.

```
Limit(x,0) Sin(x)/x;
```

Calculate the limit of a function as a variable approaches a value.

```
Newton(Sin(x),x,3,0.0001);
```

Use Newton's method for numerically finding a zero of a function.

```
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.

```
Factors(x^2-1);
```

Factorize a polynomial.

```
Apart(1/(x^2-1),x);
```

Create a partial fraction expansion of a polynomial.

2.2 A longer calculation with plotting

Here is an example of a semi-realistic numerical calculation using YACAS. The task was to visualize a particular exact solution of an elliptical differential equation. The solution was found as an infinite series. We need to evaluate this infinite series numerically and plot it for particular values of the parameters.

The function $g(q, \phi, \chi)$ is defined by

$$g(q, \phi, \chi) = \frac{1}{2\pi} \frac{\sin q\phi}{\sin 2q\phi} + \frac{1}{\pi} \sum_{n=0}^{\infty} \cos n\chi \frac{\sin \sqrt{q^2 - n^2}\phi}{\sin 2\sqrt{q^2 - n^2}\phi}.$$

Here q , ϕ and χ are numerical parameters of the problem. We would like to plot this series evaluated at fixed q and ϕ as function of χ between 0 and 2π .

To solve this problem, we prepare a separate file with the following YACAS code:

```
/* Auxiliary function */
g1(n, q, phi, chi) := [
  Local(s);
  s := q^2-n^2;
  N(Cos(n*chi) * If(s=0,
    1/2, /* Special case of s=0:
avoid division by 0 */
    Sin(Sqrt(s)*phi)/Sin(2*Sqrt(s)*phi)
  /* now s != 0 */
  /* note that Sqrt(s) may
be imaginary here */
  )
);
];
/* Main function */
g(q, phi, chi) := [
  Local(M, n);
  M := 16;
  /* Exp(-M) will be the precision */
```

```

        /* Use N() to force numerical
        evaluation */
        N(1/2*Sin(q*phi)/Sin(2*q*phi)) +
        /* Estimate the necessary number
        of terms in the series */
        Sum(n, 1, N(1+Sqrt(q^2+M^2/phi^2)),
        g1(n, q, phi, chi)) ;
];
/* Parameters */
q:=3.5;
phi:=2;
/* Make a function for plotting:
it must have only one argument */
f(x) := g(q, phi, x);
/* Plot from 0 to 2*Pi with 80 points */
Plot2D(f(x), 0: 2*Pi);

```

Name this file “fun1” and execute this script by typing

```
Load("fun1");
```

After this you should see a window with a plot.

Chapter 3

Let's learn some more

3.1 Using Yacas from the console

Command-line options

The default operation of YACAS is to run in the interactive console mode. YACAS accepts several options that modify its operation. Here is a summary of options:

- *filename* ... (read and execute a file or several files)
- `-c` (omit line prompts)
- `-d` (print default directory)
- `-v` (print version information)
- `-f` (execute standard input as one statement)
- `-p` (do not use terminal capabilities)
- `-t` (enable extra history features)
- `--archive filename` (use a given library archive file)
- `--dllmdir directory` (specify default directory for plugins)
- `--init filename` (use a given initial file)
- `--patchload` (use `PatchLoad` to load files)
- `--read-eval-print expression` (call this expression for the read-eval-print loop)
- `--rootdir directory` (specify default directory for scripts)
- `--server port` (start YACAS as a network server on given port)
- `--verbose-debug` (turn on showing some additional debugging information on screen)
- `--disable-compiled-plugins` (disable loading of compiled plugins, loading the script versions instead)
- `--stacksize size` (change size of stack arguments are stored on)
- `--execute expression` (run expression from the command line)

Options can be combined, for example

```
yacas -pc filename
```

will read and execute the file `filename` non-interactively without using terminal capabilities and without printing prompts.

Here is a more detailed description of the command-line options.

```
yacas -c
```

Inhibit printing of prompts `In>` and `Out>`. Useful for non-interactive sessions.

```
yacas -f
```

Reads standard input as one file, but executes only the first statement in it. (You may want to use a statement block to have several statements executed.)

```
yacas -p
```

Does not use terminal capabilities, no fancy editing on the command line and no escape sequences printed. Useful for non-interactive sessions.

```
yacas -t
```

Enable some extra history recall functionality in console mode: after executing a command from the history list, the next unmodified command from the history list will be automatically entered on the command line.

```
yacas [options] {filename}
```

Reads and executes commands in the filename and exits. Equivalent to `Load()`.

```
yacas -v
```

Prints version information and exits. (This is the same information as returned by `Version()`.)

```
yacas -d
```

Prints the path to the YACAS default library directory (this information is compiled into the YACAS executable) and exits.

```
yacas --patchload
```

Will load every file on the command line with the `PatchLoad` command instead of the normal `Load` command. This is useful for generating HTML pages for a web site using the YACAS scripting language, much like you can do with the PHP scripting language.

```
yacas --init [file]
```

Tells the system to load `file` as the initialization file. By default it loads the file `yacasinit.js` from the scripts directory. Thus for customization one has two options: write a `7.yacasrc` file with initialization code (as it is loaded after the initialization script is loaded), or write a custom initialization script that first uses `yacasinit.js` and adds some extra custom code.

```
yacas --read-eval-print [expression]
```

Call `expression` for the read-eval-print loop. The default read-eval-print loop is implemented in the initialization script `yacasinit.js` as the function `REP`. The default behavior is therefore equivalent to `--read-eval-print REP()`.

There is also a fallback read-eval-print loop in the kernel; it can be invoked by passing an empty string to this command line option, as `--read-eval-print ""`.

An alternative way to replace the default read-eval-print loop is to write a custom initialization script that implements the read-eval-print loop function `REP()` instead of `yacasinit.ys`.

Care has to be taken with this option because a Yacas session may become unusable if the read-eval-print expression doesn't function correctly.

```
yacas --server <port>
```

On some platforms server mode can be enabled at build time by passing the flag `--enable-server` to the `./configure` script. YACAS then allows you to pass the flag `--server` with a port number behind it, and the YACAS executable will listen to the socket behind that port instead of waiting for user input on the console.

Commands can be sent to the server by sending a text line as one block of data, and the server will respond back with another text block.

One can test this function by using `telnet`. First, set up the server by calling

```
yacas --server 9734
```

and then invoke `telnet` in another window, for example:

```
telnet 127.0.0.1 9734
```

Then type a line of Yacas input and hit Enter. The result will be one line that you will get back from the Yacas server.

Some security measures and resource management measures have been taken. No more than 10 connections can be alive at any time, a calculation cannot take more than 30 seconds, and YACAS operates in the *secure* mode, much like calling an expression by passing it as an argument to the `Secure` function. This means that no system calls are allowed, and no writing to local files, amongst other things. Something that has not been taken care of yet is memory use. A calculation could take up all memory, but not for longer than 30 seconds.

The server is single-threaded, but has persistent sessions for at most 10 users at a time, from which it can service requests in a sequential order. To make the service multi-threaded, a solution might be to have a proxy in front of the service listening to the port, redirecting it to different processes which get started up for users (this has not been tried yet).

```
yacas --rootdir [directory]
```

Tells the system where to find the library scripts. Here, `directory` is a path that is passed to `DefaultDirectory`. It is also possible to give a list of directories, separated by a colon, e.g. `yacas --rootdir scripts/:morescripts/`. Note that it is not necessary to append a trailing slash to the directory names.

```
yacas --dllldir [directory]
```

Tells the system where to find the plugins. Here `directory` is a path that is passed to `DllDirectory`. It is also possible to give a list of directories separated by a colon. The default value is specified at compile time, usually `/usr/local/lib/yacas`.

```
yacas --archive [file]
```

Use a compressed archive instead of the script library.

YACAS has an experimental system where files can be compressed into one file, and accessed through this command line option. The advantages are:

1. Smaller disk/memory use (useful if YACAS is used on small hand-held computers).
2. No problems with directory path separators: "`path/file`" will always resolve to the right file, no matter what platform (read: Windows) it runs on.

3. The start-up time of the program might improve a little, since a smaller file is loaded from disk (disk access being slow), and then decompressed in memory, which might be a lot faster than loading from disk.

An additional savings is due to the fact that the script files are stripped from white spaces and comments, making them smaller and faster loading.

To prepare the compressed library archive, run `./configure` with the command line option `--enable-archive`.

The result should be the archive file `scripts.dat`. Then launch YACAS with the command line option `--archive scripts.dat`, with the file `scripts.dat` in the current directory.

The reason that the `scripts.dat` file is not built automatically is that it is not tested, at this time, that the build process works on all platforms. (Right now it works on Unix, MacOSX, and Win32.)

Alternatively, configure Yacas with

```
./configure --enable-archive
```

and the archive file `scripts.dat` will be created in the `ramscripts/` subdirectory.

When an archive is present, Yacas will try to load it before it looks for scripts from the library directories. Typing

```
make archivetest -f makefile.compressor
```

in the `ramscripts/` directory runs all the test scripts using the archived files.

The currently supported compression schemes are uncompressed and compressed with `minilzo`. Script file stripping (removing whitespace and comments) may be disabled by editing `compressor.cpp` (variable `strip-script`).

```
yacas --disable-compiled-plugins
```

Disable loading of compiled scripts, in favor of scripts themselves. This is useful when developing the scripts that need to be compiled in the end, or when the scripts have not been compiled yet.

```
yacas --stacksize <size>
```

Yacas maintains an internal stack for arguments. For nested function calls, all arguments currently used are on this stack. The size of this stack is 50000 by default.

For a function that would take 4 arguments and has one return value, there would be 5 places reserved on this stack, and the function could call itself recursively 10000 steps deep.

This differs from the `MaxEvalDepth` mechanism. The `MaxEvalDepth` mechanism allows one to specify the number of separate stack frames (number of calls, nested), instead of the number of arguments pushed on the stack. `MaxEvalDepth` was introduced to protect the normal C++ stack.

```
yacas --execute <expression>
```

This instructs Yacas to run a certain expression, passed in over the command line, before dropping to the read-eval-print loop. This can be used to load a file before dropping to the command line without exiting (if there are files to run specified on the command line, Yacas will exit after running these scripts). Alternatively, the expression can exit the interpreter immediately by calling `Exit()`; . When used in combination with `-pc`, the Yacas interpreter can be used to calculate something and print the result to standard output. Example:

```
user% ./yacas -pc --execute '[Echo("answer ",D(x)Sin(x));E
answer Cos(x)
user%
```

Client/server usage

In addition to the interactive console sessions, a remote persistent session facility is provided through the script `yacas_client`. (This is currently only supported on Unix platforms.) By means of this script, the user can configure third-party applications to pass commands to a constantly running “YACAS server” and get output. The “YACAS server” is automatically started by `yacas_client`. It may run on a remote computer; in that case the user should have a user account on the remote computer and privileges to execute `yacas_client` there, as well as `rsh` or `ssh` access. The purpose of `yacas_client` is to enable users to pass commands to YACAS within a persistent session while running another application such as a text editor.

The script `yacas_client` reads YACAS commands from the standard input and passes them to the running “YACAS server”; it then waits 2 seconds and prints whatever output YACAS produced up to this time. Usage may look like this:

```
8:20pm Unix>echo "x:=3" | yacas_client
Starting server.
[editvi] [gnuplot]
True;
To exit Yacas, enter Exit(); or quit
or Ctrl-c. Type ?? for help.
Or type ?function for help on a function.
Type 'restart' to restart Yacas.
To see example commands, keep typing
Example();
In> x:=3
Out> 3;
In> 8:21pm Unix>echo "x:=3+x" | yacas_client
In> x:=3+x
Out> 6;
In> 8:23pm Unix>yacas_client -stop
In> quit
Quitting...
Server stopped.
8:23pm Unix>
```

Persistence of the session means that YACAS remembered the value of `x` between invocations of `yacas_client`. If there is not enough time for YACAS to produce output within 2 seconds, the output will be displayed the next time you call `yacas_client`.

The “YACAS server” is started automatically when first used and can be stopped either by quitting YACAS or by an explicit option `yacas_client -stop`, in which case `yacas_client` does not read standard input.

The script `yacas_client` reads standard input and writes to standard output, so it can be used via remote shell execution. For example, if an account “user” on a remote computer “remote.host” is accessible through `ssh`, then `yacas_client` can be used remotely, like this:

```
echo "x:=2;" | \
ssh user@remote.host yacas_client
```

On a given host computer running the “YACAS server”, each user currently may have only one persistent YACAS session.

3.2 Compound statements

Multiple statements can be grouped together using the `[and]` brackets. The compound `[a; b; c];` evaluates `a`, then `b`, then `c`, and returns the result of evaluating `c`.

A variable can be declared local to a compound statement block by the function `Local(var1, var2,...)`.

3.3 “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.

3.4 Functions as lists

Internally, YACAS represents all atomic expressions (numbers and variables) as strings and all compound expressions as lists, just like LISP. Try `FullForm(a+b*c)`; and you will see the text `(+ a (* b c))` appear on the screen. Also, any expression can be converted to a list by the function `Listify()` or back to an expression by the function `UnList()`:

```
In> Listify(a+b*(c+d));
Out> {+,a,b*(c+d)};
In> UnList({Atom("+"),x,1});
Out> 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.

Pure functions are the equivalent of “lambda expressions” of LISP; in other words, they are YACAS expressions representing bodies of functions. They are currently implemented using lists and the operator `Apply()`. The following line:

```
Apply( {{x,y},x+y} , {2,3} );
```

would evaluate to 5. Here, `{{x,y},x+y}` is a list that is treated as a pure function by the operator `Apply`, the symbols `x` and `y` become local variables bound to the parameters passed, and `x+y` becomes the body of the function.

3.5 More on syntax

The syntax is handled by an infix grammar parser. Precedence of operations can be specified explicitly by parentheses `()` when the normal precedence is not what you want. Most of the time you will enter expressions of the form `Func(var1,var2)`, or using infix operators, `a*(b+c)`, prefix operators: `-x`, or postfix operators: `x++`. The parser is case-sensitive and overall the syntax conventions resemble the C language. Last but not least there are the so called “bodied” functions, which, unlike normal functions such as `f(x,y)`, keep the last argument outside of the bracketed argument list: `“f(x) y”`. This looks somewhat like a mathematical “operator” $f(x)$ acting on y . A typical example is the function `While` which looks like `“While (predicate) body;”`. The derivative operator `D(x)` is also defined as a “bodied” function. Note that if we defined a “bodied” function `A` with only one argument, we would have to use it like this: `“A() x;”` and it would look a bit odd. In this case we could make `“A”` a prefix operator and then the syntax would become somewhat cleaner: `“A x;”`

However, regardless of presentation, internally all functions and operators are equal and merely take a certain number of arguments. The user may define or redefine any operators with either “normal” names such as `“A”` or names made of one or more of the special symbols `+ - * / = ' ? ! @ # $ % & * _ | < >` and declare them to be infix, postfix, or prefix operators, as

well as normal or bodied functions. (The symbol % is reserved for the result of the previous expression; the decimal point . is special and cannot be part of an operator's name unless the whole name consists entirely of points, e.g. the ". ." operator.) Some of these operators and combinations are already defined in YACAS's script library, for instance the "syntactic sugar" operators such as := or <--, but they can be in principle redefined or erased. These "special" operators are in no way special to the system, except for their syntax.

All infix, prefix, postfix operators and bodied functions are assigned a precedence (an integer number); infix operators in addition have a left and a right precedence. All this will only affect the syntax of input and could be arranged for the user's convenience. For example, the infix operator "*" has precedence smaller than that of the infix operator "+", and therefore an expression such as $a+b*c$ is interpreted in the conventional way, i.e. $a+(b*c)$.

An important caveat is to make sure you always type a space between any symbols that could make up an operator. For instance, after you define a new function `@@(x):=x2;` expressions such as `"a:=@@(b);"` typed without spaces will cause an error unless you also define the operator `":=@@"`. This is because the parser will not stop at `":="` when trying to make sense of that expression. The correct way to deal with this is to insert a space: `"a:= @@(b)"`. Similarly, `a!+3` will cause an error unless you define `"!+"`, so use a space: `a! +3`.

Let's now have a hands-on primer for these syntactic features. Suppose we wanted to define a function $F(x, y) = \frac{x}{y} + \frac{y}{x}$. We could use the standard syntax:

```
In> F(a,b) := a/b + b/a;
Out> True;
```

and then use the function as `F(1,2)`; We might also declare an equivalent infix operation, let's call it `"xx"`, so that we could write simply `1 xx 2`. Infix operators must have a precedence, so let's assign it the precedence of the usual division operator. The declaration goes as follows:

```
In> Infix("xx", OpPrecedence("/"));
Out> True;
In> a xx b := a/b + b/a;
Out> True;
In> 3 xx 2 + 1;
Out> 19/6;
```

Check the math and note how the precedence works!

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, or we'd get syntax errors.

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:

```
In> Bodied("##", OpPrecedence("/"));
Out> True;
In> ##(a) b := a xx b;
Out> True;
In> ##(1) 3 + 2;
Out> 16/3;
```

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 `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".

3.6 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. Examples can be found in the library files "standard", "stdfuncs", "deriv" and "solve" that come with the YACAS distribution.

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'll first define a predicate to check whether our argument is indeed a non-negative integer, and we'll use this predicate and the obvious recursion $f(n) = n f(n-1)$ to evaluate the factorial. All this is accomplished by the following three lines:

```
10 # f(0) <-- 1;
20 # f(n_IsIntegerGreaterThanZero)
    <-- n*f(n-1);
IsIntegerGreaterThanZero(_n) <--
    IsInteger(n) And n>0;
```

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.

The first two lines recursively define a factorial function $f(n) = n(n-1) \dots \cdot 1$. 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 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 operator `<--`.

There is another, slightly longer but equivalent way of writing the second rule:

```
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 post-predicate can also use the variable `n` (without the underscore). Equivalent ways of defining the same predicate in rule patterns are simply for convenience.

Precedence values for rules are given by a number followed by the `#` operator. 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.

Predicates can be combined: for example, `IsIntegerGreaterThanZero()` could also have been defined as:

```
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 `(n>0)` clause 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:

```
precedence # pattern _ postpredicate <-- replacement.
```

The optional *precedence* must be a positive integer.

Some more examples of rules:

```
10 # _x + 0 <-- x;
20 # _x - _x <-- 0;
ArcSin(Sin(_x)) <-- x;
```

The last rule has no explicit precedence specified in it (the precedence 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.

There is a slightly more complex and general way of defining rules using the functions `Rule()`, `RuleBase()` and `RulePattern()`. However, the standard library defines the “...
... <--...” construct which is much more readable and usually sufficiently flexible.

3.7 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:

```
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 `<--`. The latter would be used to define a “global” rule.

The `/:` operator traverses an expression much like `Subst()` does, i.e. 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.

Chapter 4

GNU Free Documentation License

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.

59 Temple Place, Suite 330
Boston, MA, 02111-1307
USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

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 non-commercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, **LaTeX** input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified

Version under the terms of this License, in the form shown in the Addendum below.

7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document’s license notice.
8. Include an unaltered copy of this License.
9. Preserve the section entitled “History”, and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
11. In any section entitled “Acknowledgements” or “Dedications”, preserve the section’s title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
13. Delete any section entitled “Endorsements”. Such a section may not be included in the Modified Version.
14. Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above

for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled “Acknowledgements”, and any sections entitled “Dedications”. You must delete all sections entitled “Endorsements.”

Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C) YEAR   YOUR NAME. Permission is
granted to copy, distribute and/or modify this
document under the terms of the GNU Free
Documentation License, Version 1.1 or any later
version published by the Free Software Foundation;
with the Invariant Sections being LIST THEIR
TITLES, with the Front-Cover Texts being LIST, and
with the Back-Cover Texts being LIST. A copy of
the license is included in the section entitled
‘‘GNU Free Documentation License’’.
```

If you have no Invariant Sections, write “with no Invariant Sections” instead of saying which ones are invariant. If you have no Front-Cover Texts, write “no Front-Cover Texts” instead of “Front-Cover Texts being LIST”; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

!, 6
%, 3
.., 4
:=, 4
<--, 11
==, 3
?, 2
??. 2
^C, 2

Abs, 5
Apply, 6, 10
associative lists, 4

bodied functions, 10

closures (pure functions), 10
command history, 2
command-line options, 8
 summary, 8
compiling with `libgmp`, 2
Complex, 3

D(x), 3
defining functions, 4

Example, 2
executing script files, 8
Exit, 2

Factors, 6
False, 4
ForEach, 5
fractions, 3
FromBase, 6
FullForm, 10

getting printed manuals, 2

hash tables, 4

I, 3
If, 5
Integrate, 6

lambda-expressions, 10
Limit, 6
Listify, 10
lists, 4

matrix operations, 5
multi-line commands, 2

N, 3
Newton, 6

online help, 2

pattern matching, 12
Pi, 3
Plot2D, 6
precedence of operators, 11
Precision, 3
prefix operators, 10
PrettyForm, 3
previous result operator, 3

quit, 2

recursion in rules, 11
restart, 2
rule precedence, 12

Solve, 3
special symbols, 10
strings, 4
 embedded quotes in, 4

TAB completion, 2
Taylor, 3
ToBase, 6
True, 4

UnList, 10
using compressed libraries, 9

While, 5, 10

yacas_client, 10

ZeroVector, 5