Essays on Yacas

by the YACAS team [1]

YACAS version: 1.2.2
generated on March 23, 2009

This is a book of essays on Yacas. It covers various topics, from using the Yacas system for specific calculations to general issues related to the Yacas system development and maintenance.

[1]This 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 Yacas: A do-it-yourself symbolic algebra environment **2**
- 1.1 Introduction . . . 2
- 1.2 Basic design . . . 2
- 1.3 The YACAS kernel functionality . . . 4
- 1.4 The YACAS scripting language . . . 5
- 1.5 Currently supported CAS features . . . 5
- 1.6 Interface . . . 7
- 1.7 Documentation . . . 7
- 1.8 Future plans . . . 7
- 1.9 References . . . 7

2 M. Wester's CAS benchmark and Yacas **8**

3 On Yacas programming **11**
- 3.1 Example: implementing a formal grammar . . . 11
- 3.2 Example: Using rules with special syntax operators creatively . . . 12

4 Why $-x^{-1}$ and $-\frac{1}{x}$ are not the same in Yacas **16**
- 4.1 Canonical and normal representations . . . 16
- 4.2 But how can we then build a powerful CAS? . . . 17
- 4.3 Conclusion . . . 17

5 For Yacas developers **18**
- 5.1 A crash course in Yacas maintenance for developers . . . 18
- 5.2 Preparing and maintaining Yacas documentation . . . 19
- 5.3 The Yacas build system . . . 28

6 Designing modules in the Yacas scripting language **32**
- 6.1 Introduction . . . 32
- 6.2 Demonstration of the problem . . . 32
- 6.3 Declaring resources to be local to the module . . . 32
- 6.4 When to use and when not to use `LocalSymbols` . . . 33

7 The Yacas arithmetic library **34**
- 7.1 Introduction . . . 34
- 7.2 The link between the interpreter and the arithmetic library . . . 34
- 7.3 Interface of the `BigNumber` class . . . 34
- 7.4 Precision of arithmetic operations . . . 37
- 7.5 Implementation notes . . . 44

8 GNU Free Documentation License **47**

Chapter 1

Yacas: A do-it-yourself symbolic algebra environment

A much shorter and more polished version of this paper is published as: Ayal Z. Pinkus and Serge Winitzki, "*YACAS: a do-it-yourself computer algebra system*", Lecture Notes in Artificial Intelligence 2385, pp. 332 - 336 (Springer-Verlag, 2002).

by Ayal Zwi Pinkus and Serge Winitzki

Abstract

We describe the design and implementation of Yacas, a free computer algebra system currently under development. The system consists of a core interpreter and a library of scripts that implement symbolic algebra functionality. The interpreter provides a high-level weakly typed functional language designed for quick prototyping of computer algebra algorithms, but the language is suitable for all kinds of symbolic manipulation. It supports conditional term rewriting of symbolic expression trees, closures (pure functions) and delayed evaluation, dynamic creation of transformation rules, arbitrary-precision numerical calculations, and flexible user-defined syntax using infix notation. The library of scripts currently provides basic numerical and symbolic algebra functionality, such as polynomials and elementary functions, limits, derivatives and (limited) integration, solution of (simple) equations. The main advantages of Yacas are: free (GPL) software; a flexible and easy-to-use programming language with a comfortable and adjustable syntax; cross-platform portability and small resource requirements; and extensibility.

1.1 Introduction

Yacas is a computer algebra system (CAS) which has been in development since the beginning of 1999. The goal was to make a small system that allows to easily prototype and research symbolic mathematics algorithms. A secondary future goal is to evolve Yacas into a full-blown general purpose CAS.

Yacas is primarily intended to be a research tool for easy exploration and prototyping of algorithms of symbolic computation. The main advantage of Yacas is its rich and flexible scripting language. The language is closely related to LISP `WH89` but has a recursive descent infix grammar parser `ASU86` which supports defining infix operators at run time similarly to Prolog `B86`, and includes expression transformation (term rewriting) as a basic feature of the language.

The Yacas language interpreter comes with a library of scripts that implement a set of computer algebra features. The Yacas script library is in active development and at the present stage does not offer the rich functionality of industrial-strength systems such as `Mathematica` or `Maple`. Extensive implementation of algorithms of symbolic computation is one of the future development goals.

Yacas handles input and output in plain ASCII, either interactively or in batch mode. (A graphical interface is under development.) There is also an optional plugin mechanism whereby external libraries can be linked into the system to provide extra functionality. Basic facilities are in place to compile Yacas scripts to C++ so they can be compiled into plugins.

1.2 Basic design

Yacas consists of a "core engine" (kernel), which is an interpreter for the Yacas scripting language, and a library of script code.

The Yacas engine has been implemented in a subset of C++ which is supported by almost all C++ compilers. The design goals for Yacas core engine are: portability, self-containment (no dependence on extra libraries or packages), ease of implementing algorithms, code transparency, and flexibility. The Yacas system as a whole falls into the "prototype/hacker" rather than into the "axiom/algebraic" category, according to the terminology of Fateman `F90`. There are relatively few specific design decisions related to mathematics, but instead the emphasis is made on extensibility.

The kernel offers sufficiently rich but basic functionality through a limited number of core functions. This core functionality includes substitutions and rewriting of symbolic expression trees, an infix syntax parser, and arbitrary precision numerics. The kernel does not contain any definitions of symbolic mathematical operations and tries to be as general and free as possible of predefined notions or policies in the domain of symbolic computation.

The plugin inter-operability mechanism allows extension of the Yacas kernel and the use of external libraries, e.g. GUI toolkits or implementations of special-purpose algorithms. A simple C++ API is provided for writing "stubs" that make external functions appear in Yacas as new core functions. Plugins are on the same footing as the Yacas kernel and can in principle manipulate all Yacas internal structures. Plugins can be compiled either statically or dynamically as shared libraries to be loaded at runtime from Yacas scripts. In addition, Yacas

scripts can be compiled to C++ code for further compilation into a plugin. Systems that don't support plugins can then link these modules in statically. The system can also be run without the plugins, for debugging and development purposes. The scripts will be interpreted in that case.

The script library contains declarations of transformation rules and of function syntax (prefix, infix etc.). The intention is that all symbolic manipulation algorithms, definitions of mathematical functions etc. should be held in the script library and not in the kernel. The only exception so far is for a very small number of mathematical or utility functions that are frequently used; they are compiled into the core for speed.

Portability

YACAS is designed to be as platform-independent as possible. All platform-specific parts have been clearly separated to facilitate porting. Even the standard C++ library is considered to be platform-specific, as there exist platforms without support for the standard C++ library (e.g. the EPOC32 platform).

The primary development platform is GNU/Linux. Currently YACAS runs under various Unix variants, Windows environments, Psion organizers (EPOC32), Ipaq PDAs, BeOS, and Apple iMacs. Creating an executable for another platform (including embedded platforms) should not be difficult.

A self-contained system

YACAS should work as a standalone package, requiring minimum support from other operating system components. YACAS takes input and output in plain ASCII, either interactively or in batch mode. (An optional graphical interface is under development.) The system comes with its own (unoptimized) arbitrary precision arithmetic module but could be compiled to use another arbitrary precision arithmetic library; currently linking to `gmp` is experimentally supported. There is also an optional plugin mechanism whereby external libraries can be linked into the system to provide extra functionality.

Self-containment is a requirement if the program is to be easy to port. A dependency on libraries that might not be available on other platforms would reduce portability. On the other hand, YACAS can be compiled with a complement of external libraries on "production" platforms.

Ease of use

YACAS is used mainly by executing programs written in the YACAS script language. A design goal is to create a high-level language that allows the user to conveniently express symbolic algorithms. A few lines of user code should go a long way.

One major advantage of YACAS is the flexibility of its syntax. Although YACAS works internally as a LISP-style interpreter, all user interaction is through the YACAS script language which has a flexible infix grammar. Infix operators are defined by the user and may contain non-alphabetic characters such as "`=`" or "`#`". This means that the user interacts with YACAS using a comfortable and adjustable infix syntax, rather than a LISP-style syntax. The user can introduce such syntactic conventions as are most convenient for a given problem.

For example, the YACAS script library defines infix operators "`+`", "`*`" and so on with conventional precedence, so that an algebraic expression can be entered in the familiar infix form such as

```
(x+1)^2 - (y-2*z)/(y+3) + Sin(x*Pi/2)
```

Once such infix operators are defined, it is possible to describe new transformation rules directly using the new syntax. This makes it easy to develop simplification or evaluation procedures adapted to a particular problem.

Suppose the user needs to reorder expressions containing non-commutative creation and annihilation operators of quantum field theory. It takes about 20 lines of YACAS script code to define an infix operation "`**`" to express non-commutative multiplication with the appropriate commutation relations and to automatically normal-order all expressions involving these symbols and other (commutative) factors. Once the operator `**` is defined (with precedence 40),

```
Infix("**", 40);
```

the rules that express distributivity of the operation `**` with respect to addition may look like this:

```
15 # (_x + _y) ** _z <-- x ** z + y ** z;
15 # _z ** (_x + _y) <-- z ** x + z ** y;
```

Here, `15 #` is a specification of the rule precedence, `_x` denotes a pattern-matching variable `x` and the expression to the right of `<--` is to be substituted instead of a matched expression on the left hand side. Since all transformation rules are applied recursively, these two lines of code are enough for the YACAS engine to expand all brackets in any expression containing the infix operators `**` and `+`.

Rule-based programming is not the only method that can be used in YACAS scripts; there are alternatives that may be more useful in some situations. For example, the familiar `if` / `else` constructs, `For`, `ForEach` loops are defined in the script library for the convenience of users.

Standard patterns of procedural programming, such as subroutines that return values, with code blocks and temporary local variables, are also available. (A "subroutine" is implemented as a new "ground term" with a single rule defined for it.) Users may freely combine rules with C-like procedures or LISP-like list processing primitives such as `Head()`, `Tail()`.

Code clarity vs. speed

Speed is obviously an important factor. For YACAS, where a choice had to be made between speed and clarity of code, clarity was chosen. YACAS is mainly a prototyping system and its future maintainability is more important.

This means that special-purpose systems designed for specific types of calculations, as well as heavily optimized industrial-strength computer algebra systems, will outperform YACAS. However, special-purpose or optimized external libraries can be dynamically linked into YACAS using the plugin mechanism.

Flexible, "policy-free" engine

The core engine of the YACAS system interprets the Yacas script language. The reason to implement yet another LISP-based custom language interpreter instead of taking an already existing one was to have full control over the design of the system and to make it self-contained. While most of the features of the YACAS script language are "syntactic sugar" on top of a LISP interpreter, some features not commonly found in LISP systems were added.

The script library contains declarations of transformation rules and of function syntax (prefix, infix etc.). The intention is that all symbolic manipulation algorithms, definitions of mathematical functions and so on should be held in the script library and not in the kernel. The only exception so far is for a very

small number of mathematical or utility functions that are frequently used; they are compiled into the core for speed.

For example, the mathematical operator "`+`" is an infix operator defined in the library scripts. To the kernel, this operator is on the same footing as any other function defined by the user and can be redefined. The YACAS kernel itself does not store any properties for this operator. Instead it relies entirely on the script library to provide transformation rules for manipulating expressions involving the operator "`+`". In this way, the kernel does not need to anticipate all possible meanings of the operator "`+`" that users might need in their calculations.

This policy-free scheme means that YACAS is highly configurable through its scripting language. It is possible to create an entirely different symbolic manipulation engine based on the same C++ kernel, with different syntax and different naming conventions, by simply using another script library instead of the current library scripts. An example of the flexibility of the YACAS system is a sample script `wordproblems.ys` that comes with the distribution. It contains a set of rule definitions that make YACAS recognize simple English sentences, such as "Tom has 3 apples" or "Jane gave an apple to Tom", as valid YACAS expressions. YACAS can then "evaluate" these sentences to `True` or `False` according to the semantics of the current situation described in them.

The "policy-free" concept extends to typing: strong typing is not required by the kernel, but can be easily enforced by the scripts if needed for a particular problem. The language offers features, but does not enforce their use. Here is an example of a policy implemented in the script library:

```
61 # x_IsPositiveNumber ^ y_IsPositiveNumber
      <-- MathPower(x,y);
```

By this rule, expressions of the form $\mathtt{x}^{\mathtt{y}}$ (representing powers x^y) are evaluated and replaced by a number if and only if `x` and `y` are positive numerical constants. (The function `MathPower` is defined in the kernel.) If this simplification by default is not desirable, the user could erase this rule from the library and have a CAS without this feature.

1.3 The Yacas kernel functionality

YACAS script is a functional language based on various ideas that seemed useful for an implementation of CAS: list-based data structures, object properties, and functional programming (a la LISP); term rewriting [BN98] with pattern matching somewhat along the lines of Mathematica; user-defined infix operators a la PROLOG; delayed evaluation of expressions; and arbitrary-precision arithmetic. Garbage collection is implemented through reference counting.

The kernel provides three basic data types: numbers, strings, and atoms, and two container types: list and static array (for speed). Atoms are implemented as strings that can be assigned values and evaluated. Boolean values are simply atoms `True` and `False`. Numbers are represented by objects on which arithmetic can be performed immediately. Expression trees, association (hash) tables, stacks, and closures (pure functions) are all implemented using nested lists. In addition, more data types can be provided by plugins. Kernel primitives are available for arbitrary-precision arithmetic, string manipulation, array and list access and manipulation, for basic control flow, for assigning variables (atoms) and for defining rules for functions (atoms with a function syntax).

The interpreter engine recursively evaluates expression trees according to user-defined transformation rules from the script library. Evaluation proceeds bottom-up, that is, for each function term, the arguments are evaluated first and then the function is applied to these values.

A `HoldArg()` primitive is provided to not evaluate certain arguments of certain functions before passing them on as parameters to these functions. The `Hold()` and `Eval()` primitives, similarly to LISP's `QUOTE` and `EVAL`, can be used to stop the recursive application of rules at a certain point and obtain an unevaluated expression, or to initiate evaluation of an expression which was previously held unevaluated.

When an expression can not be transformed any further, that is, when no more rules apply to it, the expression is returned unevaluated. For instance, a variable that is not assigned a value will return unevaluated. This is a desired behavior in a symbolic manipulation system. Evaluation is treated as a form of "simplification", in that evaluating an expression returns a simplified form of the input expression.

Rules are matched by a pattern expression which can contain *pattern variables*, i.e. atoms marked by the "`_`" operator. During matching, each pattern variable atom becomes a local variable and is tentatively assigned the subexpression being matched. For example, the pattern `_x + _y` can match an expression `a*x+b` and then the pattern variable `x` will be assigned the value `a*x` (unevaluated) and the variable `y` will have the value `b`.

This type of semantic matching has been frequently implemented before in various term rewriting systems (see, e.g., [C86]). However, the YACAS language offers its users an ability to create a much more flexible and powerful term rewriting system than one based on a fixed set of rules. Here are some of the features:

First, transformation rules in YACAS have predicates that control whether a rule should be applied to an expression. Predicates can be any YACAS expressions that evaluate to the atoms `True` or `False` and are typically functions of pattern variables. A predicate could check the types or values of certain subexpressions of the matching context (see the `_x ^_y` example in the previous subsection).

Second, rules are assigned a precedence value (a positive integer) that controls the order of rules to be attempted. Thus YACAS provides somewhat better control over the automatic recursion than the pattern-matching system of `Mathematica` which does not allow for rule precedence. The interpreter will first apply the rule that matches the argument pattern, for which the predicate returns `True`, and which has the least precedence.

Third, new rules can be defined dynamically as a side-effect of evaluation. This means that there is no predefined "ranking alphabet" of "ground terms" (in the terminology of [TATA99]), in other words, no fixed set of functions with predefined arities. It is also possible to define a "rule closure" that defines rules depending on its arguments, or to erase rules. Thus, a YACAS script library (although it is read-only) does not represent a fixed tree rewriting automaton. An implementation of machine learning is possible in YACAS (among other things). For example, when the module `wordproblems.ys` (mentioned in the previous subsection) "learns" from the user input that `apple` is a countable object, it defines a new postfix operator `apples` and a rule for its evaluation, so the expression `3 apples` is later parsed as a function `apples(3)` and evaluated according to the rule.

Fourth, YACAS expressions can be "tagged" (assigned a "property object" a la LISP) and tags can be checked by predicates in rules or used in the evaluation.

Fifth, the scope of variables can be controlled. In addition to having its own local variables, a function can be allowed to access local variables of its calling environment (the `UnFence()` primitive). It is also possible to encapsulate a group of variables

and functions into a "module", making some of them inaccessible from the outside (the `LocalSymbols()` primitive). The scoping of variables is a "policy decision", to be enforced by the script which defines the function. This flexibility is by design and allows to easily modify the behavior of the interpreter, effectively changing the language as needed.

1.4 The Yacas scripting language

The YACAS interpreter is sufficiently powerful so that the functions `For`, `ForEach`, `if`, `else` etc., as well as the convenient shorthand "...`<--`..." for defining new rules, can be defined in the script library itself rather than in the kernel. This power is fully given to the user, since the library scripts are on the same footing as any user-defined code. Some library functions are intended mainly as tools available to a YACAS user to make algorithm implementation more comfortable. Below are some examples of the features provided by the YACAS script language.

YACAS supports "function overloading": it allows a user to declare functions `f(x)` and `f(x,y)`, each having their own set of transformation rules. Of course, different rules can be defined for the same function name with the same number of arguments (arity) but with different argument patterns or different predicates.

Simple transformations on expressions can be performed using rules. For instance, if we need to expand the natural logarithm in an expression, we could use the following rules:

```
log(_x * _y) <-- log(x) + log(y);
log(_x ^ _n) <-- n * log(x);
```

These two rules define a new symbolic function `log` which will not be evaluated but only transformed if one of these two rules are applicable. The symbol _, as before, indicates that the following atom is a pattern variable that matches subexpressions.

After entering these two rules, the following interactive session is possible:

```
In> log(a*x^2)

log( a ) + 2 * log( x )
```

Integration of the new function `log` can be defined by adding a rule for the `AntiDeriv` function atom,

```
AntiDeriv(_x,log(_x)) <-- x*log(x)-x;
```

Now YACAS can do integrations involving the newly defined `log` function, for example:

```
In> Integrate(x)log(a*x^n)

log( a ) * x + n * ( x * log( x ) - x ) + C18

In> Integrate(x,B,C)log(a*x^n)

log( a ) * C + n * ( C * log( C ) - C ) -

( log( a ) * B + n * ( B * log( B ) - B ) )
```

Rules are applied when their associated patterns match and when their predicates return `True`. Rules also have precedence, an integer value to indicate which rules need to be applied first. Using these features, a recursive implementation of the integer factorial function may look like this in YACAS script,

```
10 # Factorial(_n) _ (n=0) <-- 1;
20 # Factorial(n_IsInteger) _ (n>0) <--
  n*Factorial(n-1);
```

Here the rules have precedence 10 and 20, so that the first rule will be tried first and the recursion will stop when $n = 0$ is reached.

Rule-based programming can be freely combined with procedural programming when the latter is a more appropriate method. For example, here is a function that computes $(x^n \bmod m)$ efficiently:

```
powermod(x_IsPositiveInteger,
   n_IsPositiveInteger,
   m_IsPositiveInteger) <--
[
  Local(result);
  result:=1;
  x:=Mod(x,m);
  While(n != 0)
  [
    if ((n&1) = 1)
[
      result := Mod(result*x,m);
    ];
    x := Mod(x*x,m);
    n := n>>1;
  ];
  result;
];
```

Interaction with the function `powermod(x,n,m)` would then look like this:

```
In> powermod(2,10,100)
Out> 24;
In> Mod(2^10,100)
Out> 24;
In> powermod(23234234,2342424234,232423424)
Out> 210599936;
```

1.5 Currently supported CAS features

YACAS consists of approximately 22000 lines of C++ code and 13000 lines of scripting code, with 170 functions defined in the C++ kernel and 600 functions defined in the scripting language. These numbers are deceptively small. The program is written in clean and simple style to keep it maintainable. Excessive optimization tends to bloat software and make it less readable.

A base of mathematical capabilities has already been implemented in the script library (the primary sources of inspiration were the books [K98], [GG99] and [B86]). The script library is currently under active development. The following section demonstrates a few facilities already offered in the current system.

Basic operations of elementary calculus have been implemented:

```
In> Limit(n,Infinity)(1+(1/n))^n

Exp( 1 )

In> Limit(h,0) (Sin(x+h)-Sin(x))/h

Cos( x )

In> Taylor(x,0,5)ArcSin(x)

     3        5
```

```
    x    3 * x
x + -- + ------
    6      40

In> InverseTaylor(x,0,5)Sin(x)

     5    3
3 * x    x
------ + -- + x
  40     6

In> Integrate(x,a,b)Ln(x)+x

                  2   /                    2 \
                 b    |                   a  |
b * Ln( b ) - b + -- - | a * Ln( a ) - a + -- |
                 2    \                   2  /

In> Integrate(x)1/(x^2-1)

Ln( 2 * ( x - 1 ) )   Ln( 2 * ( x + 1 ) )
------------------- - ------------------- + C38
         2                     2

In> Integrate(x)Sin(a*x)^2*Cos(b*x)

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

Sin( -2 * x * a - b * x )
------------------------- + C39
   4 * ( -2 * a - b )

In> OdeSolve(y''==4*y)

C193 * Exp( -2 * x ) + C195 * Exp( 2 * x )
```

Solving systems of equations has been implemented using a generalized Gaussian elimination scheme:

```
In> Solve( {x+y+z==6, 2*x+y+2*z==10, \
In> x+3*y+z==10}, \
In>      {x,y,z} ) [1]
Out> {4-z,2,z};
```

(The solution of this underdetermined system is returned as a vector, so $x = 4 - z$, $y = 2$, and z remains arbitrary.)

A small theorem prover [B86] using a resolution principle is offered:

```
In> CanProve(P Or (Not P And Not Q))

Not( Q ) Or P

In> CanProve(a > 3 And a < 2)

False
```

Various exact and arbitrary-precision numerical algorithms have been implemented:

```
In> N(1/7,40);    // evaluate to 40 digits
Out> 0.1428571428571428571428571428571428571428;
In> Decimal(1/7);    // obtain decimal period
Out> {0,{1,4,2,8,5,7}};
In> N(LnGamma(1.234+2.345*I)); // gamma-function
Out> Complex(-2.13255691127918,0.70978922847121);
```

Various domain-specific expression simplifiers are available:

```
In> RadSimp(Sqrt(9+4*Sqrt(2)))

Sqrt( 8 ) + 1

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

1

In> TrigSimpCombine(Cos(x/2)^2-Sin(x/2)^2)

Cos( x )

In> GcdReduce((x^2+2*x+1)/(x^2-1),x)

x + 1
-----
x - 1
```

Univariate polynomials are supported in a dense representation, and multivariate polynomials in a sparse representation:

```
In> Factor(x^6+9*x^5+21*x^4-5*x^3-54*x^2-12*x+40)

         3           2
( x + 2 )  * ( x - 1 )  * ( x + 5 )

In> Apart(1/(x^2-x-2))

      1              1
------------- - -------------
3 * ( x - 2 )   3 * ( x + 1 )

In> Together(%)

         9
-------------------
     2
9 * x  - 9 * x - 18

In> Simplify(%)

     1
----------
 2
x  - x - 2
```

Various "syntactic sugar" functions are defined to more easily enter expressions:

```
In> Ln(x*y) /: { Ln(_a*_b) <- Ln(a) + Ln(b) }

Ln( x ) + Ln( y )

In> Add(x^(1 .. 5))

     2    3    4    5
x + x  + x  + x  + x

In> Select("IsPrime", 1 .. 15)
Out> {2,3,5,7,11,13};
```

Groebner bases [GG99] have been implemented:

```
In> Groebner({x*(y-1),y*(x-1)})

/           \
```

```
| x * y - x |
|           |
| x * y - y |
|           |
| y - x     |
|           |
|  2        |
| y  - y    |
\           /
```

(From this it follows that $x = y$, and $x^2 = x$ so x is 0 or 1.)

Symbolic inverses of matrices:

```
In> Inverse({{a,b},{c,d}})

/                                       \
| /       d       \ /    -( b )     \  |
| | ------------- | | ------------- |  |
| \ a * d - b * c / \ a * d - b * c /  |
|                                       |
| /    -( c )     \ /       a       \  |
| | ------------- | | ------------- |  |
| \ a * d - b * c / \ a * d - b * c /  |
\                                       /
```

This list of features is not exhaustive.

1.6 Interface

Currently, YACAS is primarily a text-oriented application with interactive interface through the text console. Commands are entered and evaluated line by line; files containing longer code may be loaded and evaluated. A "notebook" interface under the GNU Emacs editor is available. There is also an experimental graphical interface (`proteus`) for Unix and Windows environments.

Debugging facilities are implemented, allowing to trace execution of a function, trace application of a given rule pattern, examine the stack when recursion did not terminate, or an online debugger from the command line. An experimental debug version of the YACAS executable that provides more detailed information can be compiled.

1.7 Documentation

The documentation for the YACAS is extensive and is actively updated, following the development of the system. Documentation currently consists of two tutorial guides (user's introduction and programmer's introduction), a collection of essays that describe some advanced features in more detail, and a full reference manual.

YACAS currently comes with its own document formatting module that allows maintenance of documentation in a special plain text format with a minimal markup. This text format is automatically converted to HTML, LaTeX, PostScript and PDF formats. The HTML version of the documentation is hyperlinked and is used as online help available from the YACAS prompt.

1.8 Future plans

The long-term goal for YACAS is to become an industrial-strength CAS and to remain a flexible research tool for easy prototyping of various methods of symbolic calculations. YACAS is meant to be a repository and a testbed for such algorithm prototypes.

The plugin facility will be extended in the future, so that a rich set of extra additional libraries (especially free software libraries), system-specific as well as mathematics-oriented, should be loadable from the YACAS system. The issue of speed is also continuously being addressed.

1.9 References

[ASU86] A. Aho, R. Sethi and J. Ullman, *Compilers (Principles, Techniques and Tools)*, Addison-Wesley, 1986.

[B86] I. Bratko, *Prolog (Programming for Artificial Intelligence)*, Addison-Wesley, 1986.

[BN98] F. Baader and T. Nipkow, *Term rewriting and all that*, Cambridge University Press, 1998.

[C86] G. Cooperman, *A semantic matcher for computer algebra*, in Proceedings of the symposium on symbolic and algebraic computation (1986), Waterloo, Ontario, Canada (ACM Press, NY).

[F90] R. Fateman, *On the design and construction of algebraic manipulation systems*, also published as: ACM Proceedings of the ISSAC-90, Tokyo, Japan.

[GG99] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, Cambridge University Press, 1999.

[K98] D. Knuth, *The Art of Computer Programming (Volume 2, Seminumerical Algorithms)*, Addison-Wesley, 1998.

[TATA99] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi, *Tree Automata Techniques and Applications*, 1999, online book: `http://www.grappa.univ-lille3.fr/tata`

[W96] S. Wolfram, *The Mathematica book*, Wolfram Media, Champain, 1996.

[WH89] P. Winston and B. Horn, *LISP*, Addison-Wesley, 1989.

Chapter 2

M. Wester's CAS benchmark and Yacas

In his 1994 paper *Review of CAS mathematical capabilities*, Michael Wester has put forward 123 problems that a reasonable computer algebra system should be able to solve and tested the then current versions of various commercial CAS on this list.

Below is the list of Wester's problems with the corresponding YACAS code. "OK" means a satisfactory solution, "BUG" means that YACAS gives a wrong solution or breaks down, "NO" means that the relevant functionality is not yet implemented.

YACAS version: 1.2.2

1. (OK) 50!

```
Verify(25!, 15511210043330985984000000);
Verify(50!, (26***50)*25!);
```

2. (OK) Factorize 50!

```
Verify(Factors(50!), {{2,47},{3,22},{5,12},
  {7,8},{11,4},{13,3},{17,2},{19,2},{23,2},
  {29,1},{31,1},{37,1},{41,1},{43,1},{47,1}});
```

3. (OK) $\frac{1}{2} + ... + \frac{1}{10} = \frac{4861}{2520}$

```
Verify(Sum(n,2,10,1/n) , 4861/2520);
```

4. (OK) Evaluate $e^{\pi\sqrt{163}}$ to 50 decimal digits

```
Verify(N(1000000000000*(-262537412640768744 +
  Exp(Pi*Sqrt(163))), 50)> -0.75, True);
```

5. (OK) Evaluate the Bessel function J_2 numerically at $z = 1 + i$.

```
NumericEqual(N(BesselJ(2, 1+I)),
  0.4157988694e-1+I*0.2473976415,Builtin'Precision'Get());
```

6. (OK) Obtain period of decimal fraction 1/7=0.(142857).

```
Verify(Decimal(1/7), {0,{1,4,2,8,5,7}});
```

7. Continued fraction of 3.1415926535.

```
Verify([Local(p,r);p:=Builtin'Precision'Get();Builtin'Precision'Set(12);r:=ContFracList(3.1415926535, 6);Built
  {3,7,15,1,292,1});
```

8. (OK) $\sqrt{2\sqrt{3}+4} = 1 + \sqrt{3}$.

```
Verify(RadSimp(Sqrt(2*Sqrt(3)+4)), 1+Sqrt(3));
```

9. (OK) $\sqrt{14+3\sqrt{3+2\sqrt{5-12\sqrt{3-2\sqrt{2}}}}} = 3 + \sqrt{2}$.

```
Verify(RadSimp(Sqrt(14+3*Sqrt(3+2*Sqrt(5-12
  *Sqrt(3-2*Sqrt(2)))))), 3+Sqrt(2));
```

10. (OK) $2\infty - 3 = \infty$.

```
Verify(2*Infinity-3, Infinity);
```

11. (NO) Standard deviation of the sample (1, 2, 3, 4, 5).

12. (NO) Hypothesis testing with t-distribution.

13. (NO) Hypothesis testing with normal distribution (M. Wester probably meant the χ^2 distribution).

14. (OK) $\frac{x^2-4}{x^2+4x+4} = \frac{x-2}{x+2}$.

```
Verify(GcdReduce((x^2-4)/(x^2+4*x+4),x),
  (x-2)/(x+2));
```

15. (NO) $\frac{\exp(x)-1}{\exp\left(\frac{x}{2}\right)+1} = \exp\left(\frac{x}{2}\right) - 1$.

16. (OK) Expand $(1+x)^{20}$, take derivative and factorize.

```
Factor(D(x) Expand((1+x)^20));
```

17. (BUG/NO) Factorize $x^{100} - 1$.

```
Factor(x^100-1);
```

(returns the same expression unfactorized)

18. (NO) Factorize $x^4 - 3x^2 + 1$ in the field of rational numbers extended by roots of $x^2 - x - 1$.

19. (NO) Factorize $x^4 - 3x^2 + 1$ mod 5.

20. (BUG) Partial fraction decomposition of $\frac{x^2+2x+3}{x^3+4x^2+5x+2}$.

```
Apart((x^2+2*x+3)/(x^3+4*x^2+5*x+2), x);
```

(does not obtain full partial fraction representation for higher-degree polynomials, e.g. $p(x)/(x+a)^n$)

21. (NO) Assuming $x \geq y$, $y \geq z$, $z \geq x$, deduce $x = z$.

22. (NO) Assuming $x > y$, $y > 0$, deduce $2x^2 > 2y^2$.

23. (NO) Solve the inequality $|x - 1| > 2$.

24. (NO) Solve the inequality $(x-1)...(x-5) < 0$.

25. (NO) $\frac{\cos 3x}{\cos x} = (\cos x)^2 - 3(\sin x)^2$ or similar equivalent combination.

26. (NO) $\frac{\cos 3x}{\cos x} = 2\cos 2x - 1$.

27. (OK) Define rewrite rules to match $\frac{\cos 3x}{\cos x} = (\cos x)^2 - 3(\sin x)^2$.

```
Cos(3*_x)/Cos(_x) <-- Cos(x)^2-3*Sin(x)^2;
```

28. (OK) $\sqrt{997} - \sqrt[6]{997^3} = 0$

```
Verify(RadSimp(Sqrt(997)-(997^3)^(1/6)), 0);
```

29. (OK) $\sqrt{99983} - \sqrt[6]{99983^3} = 0$

```
Verify(RadSimp(Sqrt(99983)-(99983^3)^(1/6))
  , 0);
```

30. (OK) $\left(\sqrt[3]{2} + \sqrt[3]{4}\right)^2 - 6\left(\sqrt[3]{2} + \sqrt[3]{4}\right) - 6 = 0$

```
Verify(RadSimp((2^(1/3)+4^(1/3))^3-6*(2^(1/3)+
  4^(1/3))-6), 0);
```

31. (NO) $\ln\tan\left(\frac{x}{2} + \frac{\pi}{4}\right) - \operatorname{arcsinh}\tan x = 0$

32. (NO) Numerically, the expression $\ln\tan\left(\frac{x}{2}+\frac{\pi}{4}\right) - \operatorname{arcsinh}\tan x = 0$ and its derivative at $x=0$ are zero.

```
Ln(Tan(x/2+Pi/4))-ArcSinh(Tan(x));
D(x)(Ln(Tan(x/2+Pi/4))-ArcSinh(Tan(x)));
```

33. (NO) $\ln\frac{2\sqrt{r}+1}{\sqrt{4r+4\sqrt{r}+1}} = 0$.

34. (NO) $\left(4r+4\sqrt{r}+1\right)^{\frac{\sqrt{r}}{2\sqrt{r}+1}}\left(2\sqrt{r}+1\right)^{\left(2\sqrt{r}+1\right)^{-1}} - 2\sqrt{r} - 1 = 0$, assuming $r>0$.

35. (OK) Obtain real and imaginary parts of $\ln(3+4i)$.

```
Verify(
  Hold({ {x}, {Re(x), Im(x)}}) @ Ln(3+4*I)
  , {Ln(5),ArcTan(4/3)});
```

36. (BUG) Obtain real and imaginary parts of $\tan(x+iy)$.

```
Hold({ {x}, {Re(x), Im(x)}}) @ Tan(x+I*y);
```

37. (BUG) Simplify $\ln\exp(z)$ to z for $-\pi < Im(z) \leq \pi$.

```
Verify(Simplify(Ln(Exp(z))), z);
```

(no conditions on z are used)

38. (NO) Assuming $Re(x)>0$, $Re(y)>0$, deduce $\sqrt[n]{x}\sqrt[n]{y} - \sqrt[n]{xy} = 0$.

39. (NO) Transform equations, $\frac{x=2}{2} + (1=1) \Rightarrow \frac{x}{2}+1=2$.

40. (BUG) Solve $\exp(x)=1$ and get all solutions.

```
Verify(Solve(Exp(x)==1,x), {x==0});
```

(get only one solution)

41. (BUG) Solve $\tan x = 1$ and get all solutions.

```
Verify(Solve(Tan(x)==1,x), {x==Pi/4});
```

(get only one solution)

42. (OK) Solve a degenerate 3x3 linear system.

```
Verify(Simplify({x+y+z==6,2*x+y+2*z==10,x+3*y+z==10}), {{x==8-3*z,y==-2,z==z}});
```

(the new routine `Solve` cannot do this yet)

43. (OK) Invert a 2x2 symbolic matrix.

```
Verify(Simplify(Inverse({{a,b},{1,a*b}})),
  {{a/(a^2-1), -1/(a^2-1)},
  {-1/(b*(a^2-1)), a/(b*(a^2-1))}});
```

44. (BUG) Compute the determinant of the 4x4 Vandermonde matrix.

```
Factor(Determinant(VandermondeMatrix
  ({a,b,c,d})));
```

(this does not factor correctly)

45. (OK) Find eigenvalues of a 3x3 integer matrix.

```
Verify(EigenValues({{5,-3,-7},{-2,1,2},
  {2,-3,-4}}) , {1,3,-2});
```

46. (OK) Verify some standard limits found by L'Hopital's rule:

```
Verify(Limit(x,Infinity) (1+1/x)^x, Exp(1));
Verify(Limit(x,0) (1-Cos(x))/x^2, 1/2);
```

47. (OK) $\operatorname{Sign}(x)$

```
Verify(D(x) Abs(x), Sign(x));
```

48. (OK) $\int |x|\,dx = |x|\,\frac{x}{2}$

```
Verify(Simplify(Integrate(x) Abs(x)),
  Abs(x)*x/2);
```

1. (OK) Compute derivative of $|x|$, piecewise defined.

```
Verify(D(x)if(x<0) (-x) else x,
  Simplify(if(x<0) -1 else 1));
```

2. (OK) Integrate $|x|$, piecewise defined.

```
Verify(Simplify(Integrate(x)
  if(x<0) (-x) else x),
  Simplify(if(x<0) (-x^2/2) else x^2/2));
```

3. (OK) Taylor series of $\frac{1}{\sqrt{1-\frac{v^2}{c^2}}}$ at $v=0$.

```
S := Taylor(v,0,4) 1/Sqrt(1-v^2/c^2);
TestYacas(S, 1+v^2/(2*c^2)+3/8*v^4/c^4);
```

Note: The result of `Taylor` is not in convenient form but is equivalent.

4. (OK) Compute the Taylor expansion of the inverse square of the above.

```
TestYacas(Taylor(v,0,4) 1/S^2, 1-v^2/c^2);
```

Note: The result of `Taylor` is not in convenient form but is equivalent.

5. (OK) (Taylor expansion of $\sin x$)/(Taylor expansion of $\cos x$) = (Taylor expansion of $\tan x$).

```
TestYacas(Taylor(x,0,5)(Taylor(x,0,5)Sin(x))/
  (Taylor(x,0,5)Cos(x)), Taylor(x,0,5)Tan(x));
```

6. (BUG) Taylor expansion of $(\ln x)^a \exp(-bx)$ at $x=1$.

```
//Taylor(x,1,3)(Ln(x))^a*Exp(-b*x);
```

(bugs in Deriv manipulation)

7. (BUG) Taylor expansion of $\ln\frac{\sin x}{x}$ at $x=0$.

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

(where's Sqr?)

```
Verify(Solve({x+y+z==6, 2*x+y+2*z==10,x+3*y+z==10}, {x,y,z
```

8. (NO) Compute n-th term of the Taylor series of $\ln\frac{\sin x}{x}$ at $x=0$.

9. (NO) Compute n-th term of the Taylor series of $\exp(-x)\sin x$ at $x=0$.

10. (OK) Solve $x = \sin y + \cos y$ for y as Taylor series in x at $x=1$.

```
TestYacas(InverseTaylor(y,0,4) Sin(y)+Cos(y),
  (y-1)+(y-1)^2/2+2*(y-1)^3/3+(y-1)^4);
```

Note that `InverseTaylor` does not give the series in terms of x but in terms of y which is semantically wrong. But other CAS do the same.

11. (OK) Compute Legendre polynomials directly from Rodrigues's formula, $P_n = \frac{1}{(2n)!!}\left(\frac{\partial^n}{\partial x^n}\left(x^2-1\right)^n\right)$.

```
P(n,x) := Simplify( 1/(2*n)!! *
  Deriv(x,n) (x^2-1)^n );
TestYacas(P(4,x), (35*x^4)/8+(-15*x^2)/4+3/8);
```

12. (OK) Compute Legendre polynomials P_n recursively.

```
Verify(OrthoP(4,x)
  , 3/8+((35*x^2)/8-15/4)*x^2);
```

13. (OK) Compute Legendre polynomial P_4 at $x=1$.

```
Verify(OrthoP(4,1), 1);
```

14. (OK) Define the polynomial $p = \sum_{i=1}^{5} a_i x^i$.

```
p:=Sum(i,1,5,a[i]*x^i);
Verify(p, a[1]*x+a[2]*x^2+a[3]*x^3
  +a[4]*x^4+a[5]*x^5);
```

15. (OK) Convert the above to Horner's form.

```
Verify(Horner(p, x), ((((a[5]*x+a[4])*x
  +a[3])*x+a[2])*x+a[1])*x);
```

16. (NO) Convert the result of problem 127 to Fortran syntax.

```
CForm(Horner(p, x));
```

17. (OK) Verify that True $\wedge$ False = False.

```
Verify(True And False, False);
```

18. (OK) Prove x Or Not x.

```
Verify(CanProve(x Or Not x), True);
```

19. (OK) Prove $x \vee y \vee x \wedge y \Rightarrow x \vee y$.

```
Verify(CanProve(x Or y Or x And y => x Or y)
  , True);
```

Chapter 3

On Yacas programming

3.1 Example: implementing a formal grammar

To illustrate the use of rules, consider a theorem prover in a simple formal grammar. (The example is the "ABIN system" from the book: W. Robinson, *Computers, minds and robots*, Temple University Press, 1992. Warning: the book is about philosophy.)

Well-formed expressions consist of symbols `A`, `B`, `I`, `N` and are either

1. `B` followed by zero or more of `I`'s, e.g. `B` , `BIII` ; or
2. `N` followed by a well-formed expression; or
3. `A` followed by *two* well-formed expressions.

This defines a certain set of well-formed expressions (statements of the ABIN language); for example, `NBII` is a statement of the language but `AB` is not. The truth/falsehood interpretation of the ABIN language is the following. All well-formed expressions starting with `NB` are interpreted as true statements (they are "axioms" of the system). In addition, there is one deduction rule allowing one to prove "theorems":

- If x and y are well-formed, then from `N`x follows `NA`xy.

Thus, `NABIBI` can be proved starting from the axiom `NBI`, but `NANBB` cannot be proved. The task at hand is to decide whether a given sequence of symbols is a provable statement of the ABIN language.

(The idea behind this interpretation is to assume that all `B`, `BI` etc. are some false statements that one could denote "B0", "B1" according to the number of "I" symbols; "N" is the logical `Not` and "A" is the logical `And`. Then the statement `NABIB` would mean "it is false that both B0 and B1 are true" and `NANBB` would mean "it is false that both B0 and negation of B0 are true". The `NANBB` statement is true in this interpretation but the deductive system of ABIN is too weak to obtain its proof.)

Implementation using predicates

The easiest way to model the ABIN language in Yacas is by using predicates. Our goal will be to define a predicate `IsProvable(x)` that will return `True` when `x` is a provable ABIN statement and `False` otherwise. We shall define `IsProvable(x)` recursively through several auxiliary predicates. Naturally, we would like to have a predicate to test well-formedness: `IsExpr(x)`. It is necessary also to have predicates for `B`-expressions, `N`-expressions and `A`-expressions, as well as for axioms and theorems. We might implement expressions by lists of symbols, e.g. `{"B", "I"}` and begin to code by

```
IsExpr(x_IsList) <-- IsBExpr(x) Or
  IsNExpr(x) Or IsAExpr(x);
IsProvable(x_IsList) <-- IsAxiom(x) Or
  IsTheorem(x);
IsAxiom(x_IsList) <-- IsNExpr(x) And
  IsBExpr(Tail(x));
```

The definitions of `IsBExpr(x)` and `IsNExpr(x)` are simple recursion to express the rules 1 and 2 of the ABIN grammar. Note the use of `Take` to create a copy of a list (we'd better not modify the value of `x` in the body of the rule).

```
10 # IsBExpr({}) <-- False;
10 # IsBExpr({"B"}) <-- True;
20 # IsBExpr(x_IsList) <-- x[Length(x)]="I"
  And IsBExpr(Take(x, {1, Length(x)-1}));

10 # IsNExpr({}) <-- False;
20 # IsNExpr(x_IsList) <-- x[1] = "N" And
  IsExpr(Tail(x));
```

The predicate `IsAExpr(x)` is a little bit more complicated because our rule 3 requires to find two well-formed expressions that follow `A`. Also, for proving theorems we need to be able to extract the first of these expressions. With this in mind, we define another auxiliary function, `FindTwoExprs(x)`, that returns the results of search for two well-formed expressions in the list `x`. The return value of this function will be a pair such as `{True, 3}` to indicate that two well-formed expressions were found, the first expression being of length 3. We shall use a `For` loop for this function:

```
FindTwoExprs(x_IsList) <-- [
 Local(iter, result);
 For( [ iter:=1; result:=False; ],
  iter < Length(x) And Not result,
    iter:=iter+1 )
   [
    result := IsExpr(Take(x, iter))
     And IsExpr(Take(x, {iter+1,
   Length(x)}));
   ];
 {result, iter-1};
];
```

Now we can define the remaining predicates:

```
10 # IsAExpr(x_IsList)_(Length(x) <= 1)
  <-- False;
20 # IsAExpr(x_IsList) <-- x[1] = "A" And
  FindTwoExprs(Tail(x))[1];

IsTheorem(x_IsList) <-- If(IsNExpr(x) And
  IsAExpr(Tail(x)) And IsProvable(
```

```
      Concat({"N"}, Take(Tail(Tail(x)),
    FindTwoExprs(Tail(Tail(x)))[2]) ));
```

The ABIN language is now complete. Let us try some simple examples:

```
In> IsExpr({"A","B"});
Out> False;
In> IsExpr({"N","B","I"});
Out> True;
In> IsAxiom({"N","B","I"});
Out> True;
In> IsTheorem({"N","B","I"});
Out> False;
In> IsProvable({"N","B","I"});
Out> True;
In> IsProvable({"N","A","B","I","B"});
Out> True;
```

It is somewhat inconvenient to type long lists of characters. So we can create an interface function to convert atomic arguments to lists of characters, e.g. `AtomToCharList(BII)` will return {"B","I","I"} (provided that the symbol `BII` has not been given a definition). Then we define a function `ABIN(x)` to replace `IsProvable`.

```
AtomToCharList(x_IsAtom) <-- [
  Local(index, result);
  For( [ index:=Length(String(x));
     result:={}; ],
   index > 0, index:=index-1 )
   Push(result, StringMid'Get(index, 1,
     String(x)));
  result;
];
Holdarg(AtomToCharList, 1);
ABIN(x) := IsProvable(AtomToCharList(x));

In> AtomToCharList(NBII);
Out> {"N", "B","I","I"};
In> ABIN(NANBB);
Out> False;
```

It is easy to modify the predicates `IsTheorem()` and `IsAxiom()` so that they print the sequence of intermediate theorems and axioms used for deriving a particular theorem. The final version of the code is in the file `examples/ABIN.ys`. Now we can try to check a "complicated" theorem and see an outline of its proof:

```
In> ABIN(NAAABIIBIBNB);
Axiom {"NBII"}
Theorem NABIIBI derived
Theorem NAABIIBIB derived
Theorem NAAABIIBIBNB derived
Out> True;
```

3.2 Example: Using rules with special syntax operators creatively

Any Yacas function can be declared to have *special syntax*: in other words, it can be made into a prefix, infix, postfix, or bodied operator. In this section we shall see how prefix, infix, and postfix operators understood by Yacas can be adapted to a problem that seems to be far removed from algebra. Nevertheless it is instructive to understand how rewriting rules are used with special syntax operators.

Suppose we want to build a system that understands a simple set of English sentences and will be able to answer questions. For example, we would like to say "Tom had an apple and Jane gave 3 apples to Tom"; the system should understand that Tom has 4 apples now. In the usual LISP-based treatments of artificial intelligence, this problem would be illustrated with a cumbersome list syntax such as `(had (Tom apple 1))` but we would like to use the power of the Yacas syntax and use plain English.

We shall create a set of rules that will "simplify" sentences to atoms such as `True` or `False`. As a side-effect, these "simplifications" will maintain a "knowledgebase" of information about all existing persons and objects.

The talking machine

The full source of this example is in the file `examples/wordproblems.ys`. In the next subsection we shall discuss the basic issues of the implementation. For now, here is an example session that shows what functionality we have in mind:

```
Unix>   yacas
True;
Numeric mode: "Internal"
To exit Yacas, enter  Exit(); or quit or Ctrl-c.
Type 'restart' to restart Yacas.
To see example commands, keep typing Example();
In> Load("wordproblems.ys")
Out> True;
In> Jitse and Ayal are persons;
OK, Jitse is a person.
OK,  Ayal is a person.
Out> {True,True};
In> apple is an object;
OK,  apple is an object.
Out> True;
In> there are many apples and pears;
Note: we already know that  apple  is an object
OK, we assume that the plural of " apple " is
  " apples ".
OK,  pear is an object.
OK, we assume that the plural of " pear " is
  " pears ".
Out> {True,True};
In> Serge had an apple;
OK,  Serge is a person.
OK,  Serge  has  1   apples  now.
Out> True;
In> Jitse had (10!) pears;
OK,  Jitse  has  3628800   pears  now.
Out> True;
In> Ayal had (2+3) apples and Serge had \
  2 pears;
OK,  Ayal  has  5   apples  now.
OK,  Serge  has  2   pears  now.
Out> {True,True};
In> Serge ate the apple;
OK,  Serge  has  no   apples  now.
Out> True;
In> Ayal ate a pear;// this should fail
Error:  Ayal  does not have enough
  pears  at this time.
Out> False;
In> Ayal gave an apple to Serge and \
  Serge gave a pear to Ayal;
OK,  Ayal  has  4   apples  now.
```

```
OK,  Serge  has  1   apples  now.
OK,  Serge  has  1   pears  now.
OK,  Ayal  has  1   pears  now.
Out> {True,True};
In> Ayal ate a pear;
OK,  Ayal  has  no   pears  now.
Out> True;
In> soup is an object and Ayal had \
  some soup;
OK,  soup is an object.
OK,  Ayal  has some  soup  now.
Out> {True,True};
In> Ayal gave soup to Serge and Serge \
  ate the soup;
OK,  Ayal  has  no   soup  now.
OK,  Serge  has some  soup  now.
OK,  Serge  has  no   soup  now.
Out> {True,True};
In> Serge has soup
Out> no;
In> Serge has apples
Out> 1;
In> Ayal has apples
Out> 4;
In> Serge has some soup
Out> False;
In> Serge has some apples
Out> True;
In> Ayal has some pears
Out> False;
In> Knowledge();
OK, this is what we know:
Persons: Jitse, Ayal, Serge
Object names: soup, pear, apple
Countable objects: pears, apples
Jitse has:
 3628800  pears

Ayal has:
 4  apples
 no  pears
 no  soup

Serge has:
 1  apples
 1  pears
 no  soup

Out> True;
In> software is an object
OK,  software is an object.
Out> True;
In> Ayal made some software
OK,  Ayal  has some  software  now.
Out> True;
In> Ayal gave some software to everyone
OK,  everyone is a person.
OK,  Ayal  still has some  software
OK,  everyone  has some  software  now.
Out> True;
In> Ayal gave some software to Serge
OK,  Ayal  still has some  software
OK,  Serge  has some  software  now.
Out> True;
In> Serge ate the software
OK,  Serge  has  no   software  now.
Out> True;
```

The string "OK" is printed when there is no error, "Note" when there is a warning, and "Error" on any inconsistencies in the described events. The special function `Knowledge()` prints everything the system currently knows.

Now we shall see how this system can be implemented in Yacas with very little difficulty.

Parsing sentences

A sentence such as "`Mary had a lamb`" should be parsed as a valid Yacas expression. Since this sentence contains more than one atom, it should be parsed as a function invocation, or else Yacas will simply give a syntax error when we type it in.

It is logical to declare "`had`" as an infix operator and "`a`" as a prefix operator quantifying `lamb`. In other words, "`Mary had a lamb`" should be parsed into `had(Mary, a(lamb))`. This is how we can do it:

```
In> [ Infix("had", 20); Prefix("a", 10); ]
Out> True;
In> FullForm(Mary had a lamb)
(had Mary (a lamb ))
Out> Mary had a lamb;
```

Now this sentence is parsed as a valid Yacas expression (although we have not yet defined any rules for the functions "`a`" and "`had`").

Note that we declared the precedence of the prefix operator "`a`" to be 10. We have in mind declaring another infix operator "`and`" and we would like quantifiers such as "a", "an", "the" to bind more tightly than other words.

Clearly, we need to plan the structure of all admissible sentences and declare all necessary auxiliary words as prefix, infix, or postfix operators. Here are the patterns for our admissible sentences:

"X `is a person`" – this declares a person. Parsed: `is(X, a(person))`

"X `and` Y `are persons`" – shorthand for the above. Parsed: `are(and(X, Y), persons)`. "person" and "persons" are unevaluated atoms.

"A `is an object`" – this tells the system that "A" can be manipulated. Parsed: `is(A, an(object))`

"`there are many` As" – this tells the system that "A" can be counted (by default, objects are not considered countable entities, e.g. "milk" or "soup"). Parsed: `are(there, many(As))`. Here "`As`" is a single atom which will have to be stripped of the ending "s" to obtain its singular form.

"X `ate` N1 As", for example, `Tom ate 3 apples` – parsed as `ate(Tom, apples(3))`. Since we cannot make the number `3` into an infix operator, we have to make `apples` into a postfix operator that will act on `3`.

"X `gave` N As `to` Y" – Here "N" is a number and "A" is the name of an object. Parsed as: `gave(X, to(As(N), Y))`. So `to` and `gave` are infix operators and `to` binds tighter than `gave`.

Sentences can be joined by "`and`", for example: "`Tom gave Jane an apple and Jane ate 3 pears`". This will be parsed as the infix operator "`and`" acting on both sentences which are parsed as above. So we need to make "`and`" of higher precedence than other operators, or else it would bind `(apple and Jane)` together.

"X `made some` A" – note that if "A" is not countable, we cannot put a number so we need to write `some` which is again a prefix operator. `made` is an infix operator.

"X `ate some` A" – the interpretation is that some A is still left after this, as opposed to "X ate the A" or "X ate A".

"X `gave some` A `to` Y" – similarly, X still has some A left after this.

After each sentence, the system should know who has what at that time. Each sentence is parsed separately and should be completely interpreted, or "simplified".

All knowledge is maintained in the variable `Knowledge` which is an associative list with three entries:

```
Knowledge := {
{"objects", {} },
{"countable objects", {} },
{"persons", {} }
};
```

The values under the keys "objects" and "countable objects" are lists of names of declared objects. The values of the "persons" key is a doubly nested associative list that specifies which objects each person has and how many. So, for example, `Knowledge["persons"]["Tom"]["apples"]` should give the number of apples Tom has now, or the atom `Empty` if he has none.

Declaring objects

Declaring persons is easy: we just create a new entry in the "persons" list. This can be done by an auxiliary routine `DeclarePerson()`. Note that after we have declared the words "`is`", "`a`" to be operators, we can just write the rule using them:

```
Infix("is", 20);
Prefix("a", 10);
_x is a person <-- DeclarePerson(x);
```

Here "`person`" will be left as an unevaluated atom and we shall never have any rules to replace it. Some other words such as "object", "objects" or "there" will also remain unevaluated atoms.

The operator "`and`" will group its operands into a list:

```
Infix("and", 30);
10 # x_IsList and _y <-- Concat(x, {y});
15 # _x and _y <-- Concat({x}, {y});
```

So expressions such as "`Lisa and Anna and Maria`" will be automatically transformed into `{Lisa, Anna, Maria}`. We shall adapt our rules to operate on lists of operands as well as on simple operands and that will automatically take care of sentences such as "there are many apples and ideas".

```
10 # there are many xs_IsList <--
  MapSingle("DeclareCountable", xs);
20 # there are many _xs <-- DeclareCountable(xs);
```

However, in the present system we cannot simultaneously parse "there are many apples and ideas" and "Isaac had an apple and an idea" because we have chosen `had` to bind tighter than `and`. We could in principle choose another set of precedences for these operators; this would allow some new sentences but at the same time disallow some sentences that are admissible with the current choice. Our purpose, however, is not to build a comprehensive system for parsing English sentences, but to illustrate the usage of syntax in Yacas.

Declaring objects is a little more tricky (the function `DeclareCountable`). For each countable object (introduced by the phrase "there are many ...s") we need to introduce a new postfix operator with a given name. This postfix operator will have to operate on a preceding number, so that a sentence such as "Mary had 3 lambs" will parse correctly.

If `x` were an unevaluated atom such as "`lambs`" which is passed to a function, how can we declare `lambs` to be a postfix operator within that function? The string representation of the new operator is `String(x)`. But we cannot call `Postfix(String(x))` because `Postfix()` does not evaluate its arguments (as of Yacas 1.0.49). Instead, we use the function `UnList` to build the expression `Postfix(String(x))` with `String(x)` evaluated from a list `{Postfix, String(x)}`, and we use the function `Eval()` to evaluate the resulting expression (which would actually call `Postfix()`):

```
Eval(UnList({Postfix, String(x)} ));
```

We also need to declare a rulebase for the operator named `String(x)`. We use `MacroRuleBase` for this:

```
MacroRuleBase(String(x), {n});
```

Finally, we would need to define a rule for "`had`" which can match expressions such as

```
_Person had n_IsNumber _Objects
```

where `_Objects` would be a pattern matcher for an unknown postfix operator such as `lambs` in our previous example. But we discover that it is impossible to write rules that match an unknown postfix operator. The syntax parser of Yacas cannot do this for us; so we should find a workaround. Let us define a rule for each object operator that will transform an expression such as 5 lambs into a list `{lambs, 5}`. In this list, "lambs" will just remain an unevaluated atom.

Incidentally, the parser of Yacas does not allow to keep unevaluated atoms that are at the same time declared as *prefix* operators but it is okay to have infix or postfix operators.

A rule that we need for an operator named `String(x)` can be defined using `MacroRule`:

```
MacroRule(String(x), 1, 1, True) {x, n};
```

Now, after we declare "`lambs`" as an operator, the routine will define these rules, and *anything* on which "`lambs`" acts will be transformed into a list.

```
In> 5 lambs;
Out> {lambs, 5};
In> grilled lambs
Out> {lambs, grilled};
```

But what about the expression "`many lambs`"? In it, `many` is a prefix operator and `lambs` is a postfix operator. It turns out that for Yacas it is the *prefix* operator that is parsed first (and remember, we cannot have unevaluated atoms with the same name as a prefix operator!) so "`many lambs`" will be transformed into `many(lambs)` and not into an illegal expression `{lambs, many}`.

Implementing semantics

After implementing all the syntax, the semantics of these sentences is very easy to transform into rules. All sentences are either about how something exists, or about someone "having", "making", "eating", or "giving" certain objects. With the rules described so far, a complicated sentence such as

```
Ayal gave soup to Serge and Serge ate the soup
```

will be already parsed into function calls

```
{gave(Ayal, to(soup, Serge)), ate(Serge,
  {soup, 1}}
```

So now we only need to make sure that all this information is correctly entered into the knowledgebase and any inconsistencies (e.g. eating something you do not have) are flagged.

Here is the simplest rule: "giving" is implemented as a sequence of "eating" and "making".

```
10 # _x gave _obj to _y <-- [
x ate obj;
y made obj;
];
```

One more subtlety connected with the notion of "countable" vs. "uncountable" objects is that there are two different actions one can perform on an "uncountable" object such as "soup": one can eat (or give away) *all* of it or only *some* of it. This is implemented using the keyword "`some`" which is a prefix operator that turns its argument into a list,

```
some _obj <-- {obj, True};
```

This list looks like the result of another quantifier, e.g.

```
the _x  <-- {x, 1};
```

but in fact the special value `True` in it is used in the definition of "`ate`" so that when you "eat" "some" of the object, you still have "some" of it left.

To implement this, we have made a special rule for the pattern

```
_x had {obj, True} <-- ...
```

separately from the general rule

```
_x had {obj, n_IsNumber} <-- ...
```

and its shorthand

```
(_x had _obj )_(Not IsList(obj)) <--
  x had {obj, 1};
```

Admittedly, the module `wordproblems.ys` has not very much practical use but it is fun to play with and it illustrates the power of syntactic constructions from an unexpected angle.

Chapter 4

Why $-x^{-1}$ and $-\frac{1}{x}$ are not the same in Yacas

Wouldn't it be wonderful if we had a program that could do all the mathematical problems for us we could ever need? Need to solve a set of equations? Just call `Solve` with the appropriate arguments. Want to simplify an expression? Just call `Simplify` and you will always get the form you would like to see. A program, simply, that could replace any mathematician. An expert system, the domain of expertise being mathematics. Wouldn't that be great?

The answer to the above is, at least according to the author, a resounding no. It is doubtful such a program will ever exist, but it is not even sure that such a program would be desirable.

Humans have a long history of making tools to make their lives easier. One important property of a tool is that it is clear conceptually to the user of that tool what that tool does. A tool should not be clever. The user of the tool can be clever about using the tool, or combining it with other tools. A 'clever' tool often results in a tool that is not useful. It is hard to understand what a clever tool does, or why it does what it does. In short: its behavior will be unpredictable.

This is a passionate plea against generic commands like `Simplify` and `Solve`.

Consider this bit of interaction in YACAS:

```
In> a:= -x^(-1)
Out> -x^(-1);
In> b:= -1/x
Out> (-1)/x;
In> a = b
Out> False;
```

Now, that can not be right, can it? Clearly, these are the same? No, they are not. They have a slightly different form, and are thus represented differently internally. the `=` sign compares the internal representation of two expressions , so `a = b` returns `False` because the internal representations of the expressions `a` and `b` are bound to are different. Note that this is behaviour that is simple to explain. The `=` operator is a 'tool', it is simple, and does one thing but does it well. It is easy to use, an important property of a tool.

To drive home this point further, suppose we did modify the `=` operator to detect that `a` and `b` are indeed equal. Great! Wonderful! `a=b` now returns `True`. But consider

```
In> c:=1+r+r^2
Out> r+r^2+1;
In> d:=(1-r^3)/(1-r)
Out> (1-r^3)/(1-r);
In> c=d
Out> False;
```

`c` and `d` are equal for all values of `r` where `r != 1`, but there a limit can be taken:

```
In> Limit(r,1)d
Out> 3;
In> Limit(r,1)c
Out> 3;
```

Now, we have to modify the `=` tool to also detect that these are the same. Actually, this will have to be done for all known identities! Or we shall have to explain for which expressions it can determine equality. This will be a complex story, it will be hard to explain. It will be a complex tool to use. And, more practically, it will be a slow tool.

So, how do we go about verifying that `a` and `b` are the same? Or that `c` and `d` are the same?

The solution lies in devising new tools.

4.1 Canonical and normal representations

A *canonical* representation for a group of expressions is a representation for each object in that group such that if two elements of the group are the same, they also have the same (internal) representation. Thus, when expressions are brought to their canonical representations, the `=` tool can be used to verify that they are the same.

A representation is called a *normal* representation if zero only has one representation. Thus `nf(a-b)=0` should be something that should return `True` if `a` and `b` are the same mathematically.

Consider a normal form defined on rational functions:

```
In> MM(a)
Out> MultiNomial({x},{{-1,-1}});
In> MM(b)
Out> MultiNomial({x},{{0,-1}})/
MultiNomial({x},{{1,1}});
```

However:

```
In> MM(a-b)
Out> MultiNomial({x},{{0,0}})/
MultiNomial({x},{{1,1}});
In> NormalForm(%)
Out> 0;
```

So here we have found a *combination* of tools that together allow us to decide that the `a` and `b` defined in the beginning of this section are the same: convert `a-b` to a normal form of `a-b`, and verify with the `=` tool that they are the same:

```
In> NormalForm(MM(a-b)) = 0
Out> True;
```

Now consider the `c` and `d` defined above. `c` and `d` are both functions of `r` only, $c = c(r)$ and $d = d(r)$. Now, let us define a function $f(r) = c(r) - d(r)$:

```
In> f(r):=Eval(c-d)
Out> True;
In> f(r)
Out> r+r^2+1-(1-r^3)/(1-r);
```

It is not quite clear yet that this is zero. But we can decide that this is zero (and thus `c(r)=d(r)`) by first noting that $f(r)$ is zero for some r, and then that the first derivative of $f(r)$ with respect to r is zero, independent of r:

```
In> f(0)
Out> 0;
In> D(r)f(r)
Out> 2*r+1-(-((1-r)*3*r^2+r^3-1))/(1-r)^2;
In> NormalForm(MM(%))
Out> 0;
```

So here we have avoided bringing `c` and `d` to canonical forms, by for example first discovering that `c` is a geometric series, and gone straight to detecting that `c-d` is in fact zero, and thus $c(r) = d(r)$.

Here again we have combined tools that are simple, do one thing but do it well, and for which it is easy to understand for human beings what they do.

4.2 But how can we then build a powerful CAS?

A new problem is introduced when algorithms are written down that require more powerful comparison tools, tools that are more sophisticated than the `=` tool for detecting that two expressions are indeed the same. The solution to this is to write the algorithm, but leave the actual comparison tool to be used by the algorithm configurable. This makes algorithms more flexible: the comparison operator can be passed in as an argument, or the algorithm can perhaps detect to which group its arguments belong, and use the appropriate tool to detect equality between two expressions.

4.3 Conclusion

A CAS (or any other system built to be used by humans for that matter) should be built up from small, well understood building blocks. Yacas contains hundreds of functions that can be combined into more powerful algorithms. These tools are documented in the documentation that comes with Yacas. Yacas solves the problem in that way. Let the user be smart, and choose the tools he needs based on understanding what the tools do. Large, complicated, cumbersome calculations can be done that way by just using well understood tools and combining them appropriately.

Chapter 5

For Yacas developers

5.1 A crash course in Yacas maintenance for developers

This document intends to give a concise description of the way Yacas is maintained. There are a few parts to maintenance to take into account:

- The `autoconf/automake` part – makefile maintenance over various systems.
- The `cvs` system – enabling developers to work together.
- The back up repository – storage of tarballs with versions of Yacas. This can be found at the following address:

`http://www.xs4all.nl/~apinkus/backups/`

The `autoconf/automake` system

The short story is as follows. You probably do not need to bother about this unless you introduce a new file. However, if you add a new file, it probably should be added to the `Makefile.am` file in the same directory. In many cases, it should be clear from the `Makefile.am` file where your new file should be added. For instance, new Yacas script files go into the huge list in `scripts/Makefile.am` that is assigned to the `SCRIPTFILES` variable. Similarly, test scripts should go in the list in `tests/Makefile.am` that is assigned to the `TESTFILES` variable. Note that you should probably also run the `cvs add` command, as explained in the section on CVS below. If you remove a file, then you should go through the inverse procedure.

The addition of new files to the `Makefile.am` ensures that it will be added to the tarball `yacas-*.tar.gz` which is uploaded to the backup repository. This has the nice side effect that you can have local files which don't automatically get added to the distribution (by not adding them to the `Makefile.am` file). Additionally, files which are not listed in `Makefile.am` may not be built and/or installed automatically. To make sure that the `tar.gz` distribution is complete, you can run the command

```
make distcheck
```

This may take a little while, as it needs to rebuild and test the whole system from the `tar.gz` tarball.

If you want to do more complicated things (like adding files which are not Yacas script or test files, or files which should be compiled or installed only conditionally), or if you are simply curious, you can read more in the chapter entitled "The Yacas build system".

Maintaining Yacas through a `cvs` repository

`CVS` provides an efficient way for developers to work together, automatically merging changes various developers make, and at the same tile is a back up system (uploading your changes to another computer from which you can easily obtain it at a later time). After a little effort setting it up it becomes very easy to use. This section describes the few commands needed for keeping your version and the version in the Yacas repository up to date.

How does `cvs` work?

CVS has a copy of the files in the repository somewhere in a directory on some system, possibly your computer. Then there is such a thing as a cvs server which you can talk to to synchronize your version of the source code with the version on the server.

CVS uses a diff-like scheme for merging differences: it looks at two text files, determines the different lines, and merges accordingly. It discovers the changes you made by looking at the version you checked out last and the version you have now, to discover which lines changed (it maintains an automatic version number for each file).

If the version of a file on your system and the version in the cvs repository has a line that has been changed by both you and some one else, the cvs repository will obviously not know what to do with that, and it will signal a 'collision' which you will have to solve by hand (don't worry, this rarely happens). More on that later.

In commands to be described in this document are in short:

- `cvs checkout` will get you an initial version. You only need to call this once.
- `cvs update` will merge the two versions and put it on your computer, so you have the latest version.
- `cvs commit` will merge the two versions and put it in the cvs repository.
- `cvs add` to add a file or directory.
- `cvs remove` to remove a file.

Checking out an initial version of Yacas

There are two ways to check out a version of Yacas: as anonymous user and as maintainer. Anonymous users don't need to log in, but also have no right to commit changes. Maintainers first need to get an account (at sourceforge), and their account needs to be enabled so they are allowed by the maintainer to make changes. A maintainer needs to log in with every command. To be able to log in, you need ssh1 installed (ssh2 will not work). You can find this at http://www.ssh.org/download.html.

To check out Yacas as anonymous user, type:

```
cvs -d:pserver:anonymous@yacas.cvs.
  sourceforge.net:/cvsroot/yacas login
cvs -z3 -d:pserver:anonymous@yacas.cvs.
  sourceforge.net:/cvsroot/yacas co yacas
```

To check out as a maintainer, type:

```
export CVS_RSH=ssh1
```

This will tell CVS to use ssh1 for communication. Then, in order to download the yacas source tree, type

```
cvs -d:ext:loginname@yacas.cvs.sourceforge.
  net:/cvsroot/yacas co yacas
```

where `loginname` is your name on the sourceforge system. This creates a directory `yacas/` with the full most recent distribution. You need to enter your password there, but other than that, that's it!

Those lines typed above are long and obscure, but it is also the last time you need to type them. From now on, if you want to do anything with cvs, just go into the `yacas/` directory you just checked out, and type the cvs command without the `-d:...` flag. This flag just tells cvs where to find the repository. But future cvs commands will know where to find them, which is why you don't need that flag.

Use case scenario 1 : getting the latest version of Yacas

You haven't looked at Yacas for a while (shame on you!) and want to check out the latest version. Just type

```
cvs update -d
```

on the command line in the `yacas` directory, and that should essentially download the latest version for you in that directory (just the changes). The `-d` option here states that you are also interested in new directories that were added to the repository. Oddly enough, cvs will only get you changed and added files, not added directories, by default.

A command

```
cvs -q update -d
```

will print messages only about changed files.

Use case scenario 2 : you made changes to Yacas

You got the latest version, but saw this huge, glaring omission in Yacas, and start hacking away to add it yourself. After a while, after playing with the code you wrote, and if you think you are finished with it, you decide you like to add it to the cvs repository.

First, you should test the new Yacas system:

```
make test
```

If there are any failed tests, you need to fix them.

Now you can start entering your changes to the CVS. If you created some new files, you need to tell CVS to add them to the source tree:

`cvs add` [list of file names of ascii text files]

This adds ascii text files. If you added binary files (GIF images in the documentation directory, or something like that), you can add it to the CVS with

`cvs add -kb` [list of file names of binary files]

Note that, when adding files to the CVS, you should normally also add them to the Yacas `tar.gz` distribution. This is done by adding the file name to the `EXTRA_DIST` variable in the file `Makefile.am` in the directory where you were adding the file.

In case files need to be removed, there are two options:

- The file is still on your drive: call `cvs remove -f [filename]` to remove the file from both your computer and the cvs repository.
- You already removed the file from your system: call `cvs remove [filename]`

There seems to be no easy way to rename or move files; you would have to remove them at their old location and add them at a new location.

Now, when finished with that, you might want to 'commit' all changes with

```
cvs commit
```

If the commit succeeds, an email is sent out to the maintainers, who can then scan the diff files for changes, to see if they agree with the changes, and perhaps fix mistakes made (if any).

If there is a collision, the commit fails (it will tell you so). This might happen because someone else also edited the same place in a file and their changes cannot be automatically merged with yours. In case of a collision, you need to invoke `cvs update` twice. The `cvs update` outputs a list of file names with a character in front of them. The important ones are the files with a 'C' before them. They have a collision. You can go into the file, and see the collision, which the cvs system conveniently marks as:

```
<<<<<<
old version
==========
new version
>>>>>>
```

You can edit the file by merging the two versions by hand. This happens very rarely, but it can happen. Use `cvs commit` afterwards to commit.

The `commit` and `update` commands can be performed in specific directories, and on specific files, if necessary, by stating them on the command line. Or you can go into a sub directory and do a `cvs commit` or `cvs update` there, if you are confident that is the only place that changed or whose changes you are interested in.

That is basically it, a quick crash course cvs. It is actually very convenient in that usually all that is needed is a `cvs commit` to fix small bugs. You type that in, and your version gets merged with the changes others made, and they get your changes, and you backed up your changes at the same time (all with that little command!).

You can find more information about cvs at `http://cvsbook.red-bean.com/`.

5.2 Preparing and maintaining Yacas documentation

Introduction

Yacas documentation in HTML and PS/PDF formats is generated by Yacas scripts from Yacas source files. Prior to version 1.0.48, all documentation had to be written directly in the Yacas language. However, it was very cumbersome to write those source files in the Yacas language. The scripts `txt2yacasdoc.pl`, `book2TeX.sh`, `book2ys.sh`, `ytxt2tex` were created to help maintain the documentation in an easy-to-read form.

The "source" form of all documentation is maintained in a special plain text format. The format is such that it is clearly

readable without any processing and is easy to edit. To compile the documents, the system processes the plain text docs with a script to prepare Yacas-language files and then runs other scripts to produce the final documentation in HTML and other formats.

The source text must be formatted in a certain fashion to delimit sections, code examples, and so on, but the format is easy enough to enter in a plain text editor. Text is marked up mostly by TAB characters, spaces, and asterisks "*" at the beginning of a line. The format is easy enough so that, for example, the text of the GNU GPL (the file `COPYING`) can be used as a documentation source file without changes. The script `txt2yacasdoc.pl` converts this markup into Yacas code for further processing.

Organization of the Yacas documentation

All documentation source files are kept in the subdirectory `manmake`. During compilation, Yacas language files as well as HTML, LaTeX and PS/PDF files are automatically generated in the `manmake` subdirectory. Contributors should only need to edit `*.txt` files in `manmake`.

Currently, the Yacas documentation consists of four "core" books (the introductory tutorial, the programming tutorial, the user's reference manual, and the programmer's reference manual), and three "extra" books (algorithms, Lisp programming, essays). [1]

All Yacas documentation books are distributed under the GNU Free Documentation License (FDL). If you add a new documentation book to Yacas, please include the file `FDL.chapt`.

The documentation books are meant to be stand-alone texts, except the two "reference manual" books which are meant to be used together because they share a common hyperlinked table of contents.

Stand-alone books are free-form, but reference books must be written with a certain template that allows online hyperlinking. The file `manmake/dummies` is an example template for a reference manual section.

Books are divided into "chapters", "sections" and "subsections". The reference manuals contain descriptions of Yacas commands and functions, and each function is given in a separate (unnumbered) section which is marked by a special `*CMD` label (see below).

At the beginning of each book there *must* be a book title and a short book description (labeled `*BLURB`). The short description appears in the printed version as the subtitle on the title page. It also goes into the HTML top-level book index as the book description.

At the beginning of each chapter there may be a "chapter introduction" labeled `*INTRO` which is also a short description of the contents of that chapter. It may be one paragraph only. The "chapter intro" feature is only used in the HTML reference manual because it is the text that appears at the very top of a reference manual section. As you can see in the HTML version of reference docs, each chapter contains a list of all functions described in it. This list goes right after the first paragraph of "chapter intro". In the printed (PS/PDF) documentation, the `*INTRO` label is ignored and the "chapter intro" paragraph is shown exactly like any other text paragraph.

Each printed book contains a table of contents at the beginning and an index at the end. The index should help a reader to quickly find a particular concept. For example, all documented Yacas commands are automatically entered into the index in the reference manual. Additional index entries should be inserted by hand using the `*A` label (see below).

[1] There is also a documentation book describing the Emacs interface to Yacas (the "Yacas Notebook" mode). This book is not available as online help but is installed separately in the TeXinfo or PostScript formats.

Translating the documentation

The Yacas system is under active development and its documentation is constantly updated. An effort to translate the Yacas documentation from English into other languages has been started.

Translators should try to prepare translated documentation in the same plain text format as the original documentation. Any problems with conversion to HTML and PS/PDF formats should be easy to solve (at least for European languages).

The most important documentation books to translate are the reference manual and the tutorials. There is substantial documentation aimed at developers, for instance the algorithms book or the essays book. It is probably not as important to translate such books, since Yacas developers have to speak English to communicate.

Formatting of source text files

Formatting of source text files uses TAB symbols; if your editor does not support them and converts them to spaces, you should convert the results back to contain real TAB symbols using the standard Unix `unexpand` utility or a custom perl script.

You may want to examine the source of this file (`manmake/YacasDocs.chapt.txt`) to see how various features of the markup are used. Currently the following markup is implemented:

- Paragraphs are separated by blank lines (lines consisting of space characters or empty). Several blank lines next to each other are equivalent to one. However, TAB-indented blank lines inside a code example (see below) do not create another paragraph.
- Book heading is quadruple TAB indented. Chapter heading is triple TAB indented. Section heading is double TAB indented. Subsection heading is indented by a TAB and 4 spaces. Headings must be within one line (but that line can be as long as needed).
- Sample code is single TAB indented, for example:

  ```
  In> 1+2;

  Out> 3;
  ```

  Note that empty lines in a single sample code block must be also TAB indented or else the sample code block will be split into several sample code paragraphs. A sample code block may or may not be separated from the text that follows it by an empty line. If the code block is not separated by an empty line, then the text following it will not be made into a separate paragraph (this currently affects paragraph indentation only in PS/PDF docs).
- If a section or chapter heading immediately follows a sample code block (i.e. when it is the last code sample in the previous section), they *must* be separated from the headings by an empty (unindented) line. The reason for this is that the script will assume that everything which is at least single-TAB indented (up to a separation between paragraphs) belongs to one sample code block. This makes it easier to enter multiply indented sample code: a double-TAB indentation inside a sample code block will not start a new section. For example:

```
While(x<0) [
x:=x+1;
Write(x);
];
```

- Ordinary text must not be indented at all. Line length is arbitrary and linebreaks inside a paragraph are of no significance to the resulting documentation. The first symbol on a line should not be an asterisk (*) because it is reserved for markup purposes. In most cases it is okay to have an asterisk in the first position, though, as long as it does not conflict with any markup labels (see below). All markup labels start with an asterisk "*" in the first position on a line, followed by an uppercase keyword, e.g. `*CMD` or `*INCLUDE`. Some markup labels have arguments that follow them, and some take an entire following paragraph as an argument and must be terminated by a blank line.

- Itemized text is marked by "*" in the first position, followed by TAB. For example:

```
* Item
* Another item
```

This will produce:

- Item
- Another item

- Enumerated text is marked by "*" followed by TAB, number and period. The number used in the source text is irrelevant because the enumerated environment of the final document will introduce its own counter. For example:

```
* 0. First item
* 0. Second item
```

This will produce:

1. First item
2. Second item

Note that the text of an item continues until the next itemized line is given or until end of paragraph (does not have to be all in one line).

Nesting of enumerated or itemized environments is not supported, except for fringe cases of nesting just one itemized list at the very end of an enumerated list or vice versa.

The enumerated environment is currently only implemented in LaTeX docs; HTML docs render them as itemized.

- Emphasized text (*italics*) should be surrounded by `<i>` `</i>`. Note that the whole emphasized fragment of text must be located within a single line, or else it will not be emphasized.

- `Typewriter font` text is surrounded by braces {}. The typewriter font fragment must be within a single line and may contain no more than four nested sets of {} inside. This is hopefully enough for our documentation. A limitation of this markup is that there is no way to put a single brace in text alone without a matching brace. This would be okay for HTML docs but it breaks LaTeX docs because braces are special in TeX and because Serge was too lazy to implement a real parser.

- Web hyperlinks are surrounded by `<* *>`. Text of the link must begin with `http://`, `https://` or `ftp://`. Alternatively, a hyperlink with anchored text is made by the markup `<*` *anchored text* `|` *Web URL* `*>`. For example:

```
<*http://host.net/file.html#anchor*>
```

or

```
<*click here|somewebpage.html*>
```

(the latter example will refer to a local HTML file, i.e. a file in the documentation directory).

Note: Currently, only the HTML documentation is hyperlinked, while the printed PS/PDF documentation contains only text.

- It is also possible to create hyperlinks pointing to other parts of the documentation. Such "documentation hyperlinks" are similar to the Web hyperlinks, except they use a special "protocol" named `yacasdoc` and chapters and sections can be referred to as "subdirectories". For example, this section can be referred to by the following code:

```
<*yacasdoc://essays/5/2/*>
```

A hyperlink with anchored text is made by the markup `<*` *anchored text* `|`yacasdoc://...`*>`. For example:

```
<*this section|yacasdoc://essays/5/2/*>
```

will generate a reference to *this section*, in *Essays on Yacas, Chapter 5, Section 2* in the documentation. There are currently the following ways to create documentation hyperlinks:

1. Hyperlink into an anchor of the same file:

```
<*yacasdoc://#documentation!hyperlinks*>
```

The anchor "`documentation!hyperlinks`" should have been created using the `*A` label.

2. Hyperlink into an anchor in a different documentation book:

```
<*yacasdoc://Algo/3/#adaptive plotting*>
```

Note that one must specify the book name "`Algo`" and the chapter number (`3`) for this to work. Currently supported book names are `Algo`, `coding`, `essays`, `intro`, `Lisp`, `ref`, and `refprog`. A warning message is printed if the book name is not given correctly.

3. Hyperlink into a given chapter or a given section of a documentation book:

```
<*yacasdoc://Algo/3/1/*>
```

- Mathematical expressions should be typed in Yacas syntax (*not* in TeX notation) and surrounded by dollar signs. Both delimiting dollar signs *must* be within one line of text. For example: `$ x^2 + y^2 != z^2 $` produces $x^2 + y^2 \neq z^2$. Double dollar signs will denote a displayed equation, like in TeX; both pairs of dollar signs should still be within one line.

There is a special feature for displayed equations: Any punctuation immediately following the second pair of dollar signs will be displayed on the same line. (This is used to get around the limitation of mathematical expressions that cannot end with a comma or a period or with another punctuation mark.) For example, the formula "`$$x^2/2$$,`" will produce

$$\frac{x^2}{2},$$

with the comma on the same line. A formula such as "`$x+1;$`" will generate an error; the semicolon should be moved out of the dollar signs.

As special exceptions, one can enter the symbols "TEX" and "LATEX" as if they are Yacas expressions, i.e. "`$TeX$`" produces "TEX". One can also create a capitalized form of the name YACAS as "`{Yacas}`".

Please note: Mathematical expressions *must* be *valid Yacas expressions*, with no unbalanced parentheses, no undefined infix operators, no hanging periods and so on, or else the Yacas script that formats the docs will fail! (This limits the scope of mathematical formulae but is hopefully not critical.)

Also, please avoid putting equations into documentation as plain text. Expressions such as $a > 0$ are not correctly typeset by TEX if included into the plain text without the dollar signs: "a¿0". (The HTML documentation is currently not affected.)

Currently, when creating online HTML documentation, all mathematics is kept in Yacas notation and set in boldface font. (This may change in the future.) Of course, LATEX typesets the documentation with correct mathematical symbols.

Another feature of the LATEX exporter is that it will first try to represent all functions and infix operators according to their mathematical meaning, and if no such meaning is defined in Yacas, then it will show them exactly as they are written in Yacas. For infix operators to work, they have to be declared in the standard library, or else an error will occur when processing the manual.

For example, the Yacas operators `=` and `==` are represented in LATEX by an equals sign (=), the operator `:=` becomes "identically equal" ($a \equiv b$), and the cosmetic operators `<>` and `<=>` become $a \sim b$ and $a \approx b$. But you cannot use an infix operator such as "`:=*`" because it is not defined in the standard library. A Yacas function which is not defined in the standard library, for example "`func(x)`", will appear just like that: $\mathrm{func}\,(x)$.

- Documentation may be split between several files for convenience. To insert another file, use the `*INCLUDE` label, e.g.

```
*INCLUDE ../essays/howto.chapt
```

Note that the included document is the file `howto.chapt`, not `howto.chapt.txt`, because it must be a Yacas-language file, not a `.txt` file. (The `IncludeFile()` call, an alias to `Load()`, will be used to execute the specified file.)

- Comments may be introduced by the label `*REM`. The line and the paragraph of text following `*REM` will be omitted from the documentation. An empty line should separate the `*REM` block from other text. (A mark-up label at the beginning of line will also terminate a `*REM` block.) For example,

```
*REM this is
a comment

(documentation text continues)
```

- Footnotes may be entered as a line containing the label `*FOOT`. Footnote text must be within one line (because a footnote does not necessarily break a paragraph).

For example, the text

```
*FOOT This is an example footnote
```

generates the footnote [2]

- Yacas expressions may be evaluated inline by using the directive `*EVAL`. Anything that follows `*EVAL` until the end of the line will be evaluated as a Yacas expression. If this expression prints something (e.g. via `Write`), the output will be inserted into the text (as is). The resulting value of the expression will also be inserted, unless the expression evaluates to `True`. The Yacas expression *must* be on one line.

For example,

```
*EVAL "Yacas version: " : Version()
```

will insert the string ' Yacas version: 1.2.2 ' into the manual. Note the spaces around the version string—these additional spaces are currently unavoidable. [3]

Formatting conventions for the reference manual

The formatting explained in the previous section is enough to create most of the user guide and tutorial documentation. The script `txt2yacasdoc.pl` also implements some additional markup features to help create the reference manual.

A typical reference manual subsection documenting a certain function may look like this in plain text:

```
*CMD PrintList --- print list with padding
*STD
*CALL
   {PrintList}(list)
   {PrintList}(list, padding);

*PARMS

{list} -- a list to be printed

{padding} -- (optional) a string

*DESC
Prints {list} and inserts the {padding} ...

*E.G.

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

*SEE Write, WriteString
```

Compare this with the reference manual section on the function `PrintList` to see how this plain text markup is rendered in the finished documentation.

Notes:

- Some labels have parameters while other labels do not; labels that do not have parameters must be put alone on a line.
- The `*STD` label is for functions defined in the standard library and the `*CORE` label is for built-in functions defined in the Yacas core engine. In addition, the labels `*UNIX`, `*MSWIN` and `*MAC` can be used to denote Un*x, MS Wind*ws and Macint*sh-specific add-on functions.
- There must be *some* whitespace separating a markup label such as `*SEE` and the following text. Either TAB characters or spaces work equally well.
- The comma-space combination "`, `" is mandatory when a label accepts a list of arguments. (One can put several spaces after a comma.) Lists of commands are used by the `*CMD` and `*SEE` labels.

[2] This is an example footnote.

[3] If the absence of spaces is critically important, you can create the required text in a Yacas expression.

- Characters `<` and `>` are of special significance to both HTML and TEX and should be always escaped—either by braces {} or by dollar signs, as appropriate.
- In the "examples" section, there may be just one example, in which case the alternative label "`*EG`" could be used instead of "`*E.G.`". This will currently generate the word "Example:" instead of "Examples:" in the documentation. (This is a cosmetic feature.)
- Each example is a candidate for inclusion into the Yacas test suite. The Yacas code after the `In>` prompts and the resulting expressions that follow the `Out>` prompt are extracted into special files by the script `txt2example.pl`. (This provides an automatic check that the manual still agrees with the actual behavior of Yacas.) The script will ignore any text that is not preceded by `In>` or `Out>`. However, some examples are not appropriate for automatic testing and must be explicitly excluded. An example section is excluded for testing if the `*E.G.` or `*EG` label is followed by the word "notest", for example:

  ```
  *EG notest
  // some advanced tests
  ```

  Tests may be unsuitable for automatic processing for several reasons, including: system-dependent results (e.g. need to have particular files on the system); calculations that take a long time; calculations that output something to the screen and not just return an expression.
- In a subsection there may be either one function documented or several at once: for example, it may make sense to document `Sin`, `Cos` and `Tan` together. In this case, all function names should be simply listed in the `*CMD` header, for example:

  ```
  *CMD Sin, Cos, Tan --- Trigonometric ...
  ```

In addition to the above labels, there are the following tags:

- `*INTRO` to denote a "reference chapter introduction" corresponding to the `ChapterIntro()` function
- `*BLURB` for the short book summary (it enters the HTML book index and the front page of the LATEX docs)
- `*A` to manually create an HTML anchor in a reference manual section and an index entry in the printed docs (see below for more details on indexing)
- `*HEAD` to create a small heading. The `*HEAD` tag might be useful for lowest-level headings. Currently, the special markup for the reference manual implements the same tag for its topical sections ("Parameters", "See also" etc.) `*HEAD`. For instance,

  ```
  *PARMS
  ```

  results in the same text as

  ```
  *HEAD Parameters:
  ```

Usage of the `*A` label currently does not directly affect the appearance of the docs. In the HTML docs, it inserts the insivible anchor tags `<a></a>`. In the printed LATEX docs, the `*A` label adds an index entry. The `*CMD` tag generates all necessary HTML anchors and index entries for commands in the reference manual. So only non-command index entries need to be manually entered using `*A`.

The `*INTRO` and `*BLURB` tags only work for one paragraph. There must be no empty line between `*INTRO`/`*BLURB` and that paragraph. Also, there must be no empty line between the "blurb" and the book title (for technical reasons). There must be one and only one "blurb" paragraph in a "book" and no more than one "chapter intro" paragraph per chapter.

This markup should be sufficient for creating reference documentation in plain text.

Indexing the documentation books

It is not difficult to automatically generate an alphabetically sorted index for the books. An "index entry" is a piece of text that does not appear in the book where it is entered, but instead is printed in the alphabetical list at the end of the text with the relevant page number.

Currently the following facilities are provided for indexing:

1. The label `*CMD` automatically adds index entries for all commands it describes. In this way, the printed reference manual automatically has every documented command listed in the index.
2. The label `*A` can be used to add an index entry by hand. An index entry can be any text that is admissible in the documentation (on one line). Preferably it should be something concise and something that users will want to look up and can't easily locate in the table of contents.

After LATEX generates a "raw" index file `*.idx`, the `makeindex` utility is used to post-process and sort the index into the `.ind` file. If you do not have `makeindex` on your system, the book indices will not be generated.

Note that `makeindex` is not always friendly to special (non-alphanumeric) characters. For example, it uses the symbol `!` to separate index topics, which may conflict with Yacas commands. In other words, document index must be tested and sometimes debugged.

In the HTML docs, the index is currently not generated on a separate page, although HTML anchors are inserted in the text.

An index entry may be a "topic" with "subtopics", which usually appears in book indices like this:

```
gnus, 51
   tame, 51
   wild, 52-341
```

This effect can be achieved with the `!` topic separator:

```
*A gnus
*A gnus!tame
*A gnus!wild
```

This is a special feature of `makeindex`.

Currently, it is possible to include a command or an equation into an index entry, for example,

```
*A {InterestingCommand}
*A calculation of $Sqrt(x)$
```

But this may sometimes conflict with the topic separator.

Summary of mark-up labels

Mark-up labels *must* appear as first characters on a line. The following mark-up labels are currently defined:

Labels with an argument on the same line (affect only the current line):

- `*A` *anchor* – insert anchor and index entry
- `*BOOK` *title* – start a book, give title
- `*EVAL` *statement* – evaluate inline as a YACAS statement and insert results
- `*HEAD` *heading* – lowest-level heading (lower than subsection)

- `*FOOT` *text* – insert a footnote
- `*INCLUDE` *filename* – include another documentation file
- `*YSFILE` *filename* – give an alternative file name for YACAS code extraction (see the section on `book2ys`)

Labels that affect the rest of the line and the subsequent paragraph:

- `*BLURB` – short summary of the book (must immediately precede the `*BOOK` label without any empty lines)
- `*INTRO` – chapter introduction (significant for HTML only) (must be separated by an empty line from what follows)
- `*REM` – documentation comment (must be separated by an empty line from what follows)

Special labels for the reference manual that accept several arguments on the same line:

- `*CMD` or `*FUNC` – command name and one-line description
- `*SEE` – "See also"

Special labels without arguments that generate headings for the reference manual:

- `*STD` – "Standard library"
- `*UNIX` – "Unix-specific"
- `*MSWIN` – "MS Windows-specific"
- `*MAC` – "Macintosh-specific"
- `*CORE` – "Core function"
- `*CALL` – "Calling format"
- `*PARMS` – "Parameters"
- `*DESC` – "Description"
- `*E.G.` – "Examples"
- `*EG` – "Example"

Other special markup:

- `*BREAK` – insert an explicit line break at this point (no new paragraph)
- `*NEWPAGE` – start a new page at this point (LaTeX documentation only)
- `<*` ... `*>` – insert a hyperlink

Summary of special markup syntax

Special syntax entities include:

- TAB indented: book title (4 TABs), chapter (3 TABs), section (2 TABs), subsection (1 TAB and 4 spaces) titles, sample code (1 TAB)
- asterisk-TAB: enumerated environment
- asterisk-TAB-number-period: itemized environment
- curly braces "{}": inline code samples, names of functions or variables (monospaced font)
- special delimiters "`<i>`" and "`</i>`": italics (for emphasis, not for mathematics). Both delimiters must be on the same line.
- dollar signs "`$`", "`$$`": inline and displayed mathematical equations
- special delimiters "`<*`" and "`*>`": Web hyperlinks and documentation hyperlinks. Both delimiters must be on the same line.
- the exclamation mark "`!`": nested index entries (use inside the `*A` label)

Debugging the manual

Sometimes the manual compilation `make` or `make texdocs` will break after you edit the plaintext manual sources. This can happen for one of these reasons:

1. A math syntax error. You have used a mathematical formula that does not evaluate to a Yacas expression. Unbalanced parentheses, invalid infix operators such as `>-`, `=-`, or forgotten punctuation inside the formula such as `x+1:` are the most frequent culprits. This will break both HTML and TeX manual formats.
2. The HTML format compiles but the TeX does not (the `latex` commands never finishes, that is, latex prints an error message which you do not see because it is redirected to `/dev/null`, and waits for your input; you have to kill the process). This means that somewhere the generated TeX code is incorrect. You probably forgot to balance braces {} or something more subtle happened.
3. Some mark-up which should not be split between lines was unintentionally split by reformatting paragraphs in a text editor. This will sometimes not break the compilation but will always give undesired results.

In case of a math syntax error, the documentation exporter cannot print the paragraph where the error occurred, but it usually prints the preceding paragraph. Currently, the easiest way to locate the error is to generate the `.tex` output and look at it, e.g.:

```
make ref.book.tex; less ref.book.tex
```

The last line in the `.tex` file must be `end{document}`. If it is not, then the last portion of the text you see in the `.tex` file is the text directly before the paragraph where the error occurred. Most probably, there is a malformatted math formula in the next paragraph of your plaintext source.

If the last line is `end{document}` but `latex` does not finish, you will have to run `latex` by hand, e.g.

```
latex ref.book.tex
```

and look at the error message(s) it prints.

Using the script `txt2yacasdoc.pl`

The script `txt2yacasdoc.pl` is used to transform plain text markup into the Yacas language. The script acts as a stream filter:

```
perl txt2yacasdoc.pl < file.txt > file.chapt
```

In this example, `file.txt` contains some formatted plain text (source text) and the resulting file `file.chapt` will be produced in Yacas-language documentation format.

There is a single option for `txt2yacasdoc`:

```
perl txt2yacasdoc.pl -debug < file.txt \
  > file.chapt
```

This option is to be used for debugging, i.e. when the resulting file does not compile in Yacas. The effect of this option is to introduce more breaks between text strings in the generated file, so that the `Text()` function is called more often. It is then easier to locate the source of the problem in the Yacas-language file (Yacas will tell you the last line in the Yacas-language file at which a syntax error occurred). This option is largely obsolete because the `Text()` function is called frequently enough by default. See below for hints about finding syntax errors in documentation when the manual does not compile.

book2TeX: preparing typeset documentation

The script `book2TeX.sh` prepares a TEX file out of a Yacas-language documentation book. Usage is similar to `book2txt.sh`, except that only one file is processed at a time and the file must be a "book", not just a "chapter". For example:

```
book2TeX.sh intro.book intro.book.tex
```

will create a LATEX-formatted version of the introductory tutorial. The LATEX file can be processed with standard tools, for example

```
latex intro.book.tex
dvips -o intro.book.ps intro.book dvi
```

will prepare a Postscript version, while

```
pdflatex intro.book.tex
```

will prepare a PDF version.

To generate printed docs, it is necessary to run `latex` (at least) three times in a row. This is because at first `latex` does not know how much space will be taken by the table of contents and the index, so the page numbers are all off by a few pages. Only on the second run `latex` generates correct page numbers for the TOC (the `.aux` file) and for the index (the `.idx` file). After this the index file has to be processed by the `makeindex` routine to sort it, and the third `latex` run is needed to actually insert the correct TOC and the processed index into the final document.

The shell commands in `book2txt.sh` execute the following Yacas commands:

```
Use("book2TeX.ys");
ToFile("file.chapt.tex")
  Load("file.chapt");
```

This requires that the Yacas script `book2TeX.ys` be available in the current directory. The shell script `book2TeX.sh` assumes that `book2TeX.ys` is stored in the same directory as `book2TeX.sh` and that the Yacas executable is available in the directory `../src/`. Alternatively, the command line of the Yacas executable can be specified by the `-run` option. For example, the `Makefile` runs `book2TeX.sh` like this:

```
book2TeX.sh -run "yacas-dir/src/yacas --rootdir
  yacas-dir/scripts" file.book file.book.tex
```

Note that the entire Yacas command line is given in quotes.

Some concerns with the printed documentation

Not all features of the Yacas documentation-generating scripts are compatible with TEX typesetting. To prevent errors, documentation source should avoid certain things. In general, it is a good idea to check the typeset appearance of documentation, since it helps detect errors.

For example, the symbols `%`, { }, `< >`, `#`, , _ and `&` are special to TEX. They should not normally be used in plain text; it is okay to use them in "typewriter" text (within braces {}) or code samples – but *not* in section or chapter heads, because it makes it difficult to export to TEX correctly. TEX commands may be entered but will not be correctly rendered in HTML online documentation.

Sometimes fixed-font text will hang over the right edge of the printed page. A workaround is to break the fixed-font text into shorter fragments or to rephrase the text.

Another concern is that code examples (TAB-indented blocks) are typeset in a fixed-width font and may not fit into the width of the page. To avoid this, the lines in the code examples should not be longer than about 50 characters.

The current implementation uses a "report" style which allows chapters, sections, subsections and includes an automatically generated table of contents and an index. The standard 10 point font and two-column format are used to save space (and trees).

The script `txt2yacasdoc.pl` attempts to convert double quotes `""` into proper English quotation marks "". However, this automatic conversion sometimes fails and produces wrongly-directed quotes. One such case is if the quotes are on the same line as a TAB character (e.g. in an itemized environment). This problem can be circumvented by putting the quoted words on a different line.

Some complicated mathematical expressions may not correctly render in TEX. This is because Yacas uses its library function `TeXForm()` to transform Yacas expressions to TEX. Mathematical expressions are entered in the plain text documentation source using Yacas syntax, then transformed to a special non-evaluating call `TeXMath()` in the Yacas-language documentation, which formats into HTML using a `Write()` call or into TEX using a `TeXForm()` call, as necessary. Testing should be performed on documentation before releasing it. The most stringent limitation is that the expression between dollar signs should evaluate in Yacas (preferably to itself) and not cause syntax errors. In case of doubt, check that the expression evaluates without errors and then try to use `TeXForm` on that expression and see if that evaluates without errors as well. For example, expressions such as `x=+1` will cause a syntax error and this will break the compilation of the manual (both HTML and TEX).

book2ys: extracting Yacas code from books ("literate programming")

`book2ys.sh` is a shell script that extracts Yacas code examples from a documentation chapter into a separate file. All other text is omitted.

Usage is similar to `book2TeX.sh`. For example, the benchmarking test code `wester.yts` can be automatically extracted from the corresponding essay chapter by the command

```
sh ../manmake/book2ys.sh wester-1994.chapt \
  wester.yts
```

After this command, the file `wester.yts` is created. Note that `wester-1994.chapt` is in Yacas language and is itself a generated file.

To prepare a documentation chapter in such a way that code extraction is possible, one needs to make sure that *all* code examples in the chapter taken together will become a correct sequence of Yacas expressions when cut out and written sequentially into a file. So, for instance, semicolons at the end of each statement are required. The script `book2ys` will *not* export example Yacas session code with "`In>`" and "`Out>`" prompts but it will export *all other* example code.

The example code with "`In>`" and "`Out>`" prompts will become comments in the exported Yacas file. It is possible to suppress this comment generation by the `-strip` option to the `book2ys.sh` script.

If the output file name is not given, the name will be the same as the Yacas book source name but with the `.ys` extension.

See the source file `wester-1994.chapt.txt` to get a feeling of how the source documentation is formatted to allow completely automatic code extraction. Note that the printed documentation may be in twocolumn format, and therefore it is necessary to split lines that are too long.

Using the script `book2ys.sh`, one can write documentation and code together, a la "literate programming". The main idea

of literate programming is that the program code and the documentation should be written as one large book explaining to humans how the program is organized and how it works. From this book, a printable copy is generated and all code is automatically extracted for compilation.

Literate programming may require that code be split between many source files, and yet it may be convenient to keep all descriptions in one book. The special document formatting label `*YSFILE` can be used to redirect output to different Yacas source files.

By default, the output goes to the file specified on the `book2ys.sh` command line. This default can be restored by the directive "`*YSFILE -`" at any time. Otherwise, all code output will be printed to the file specified after the `*YSFILE` label.

Note that the multiple file support is somewhat restrictive:

- Once the output file name has been changed, the old file is closed and cannot be appended to. (This is a limitation of the `ToFile()` function in Yacas.)
- If the same file name is chosen again, the file will be overwritten.
- If the output file has been changed at least once, then at the end of the document the file name must be reset to the default "`-`". Otherwise the last file will not be properly closed.

Here is an example of using multiple files. Note how documentation text is interspersed with TAB-indented code.

```
Text that does not appear in the code.
    // code into default file
    // more code
*YSFILE script1.ys
This will not appear in the file.
    x:=1;
    // some code for the file script1.ys
*YSFILE script2.ys
    // some code for the file script2.ys
End of the example, need to reset *YSFILE.
*YSFILE -
```

After processing this file with `book2ys.sh`, one should get three files with Yacas code.

`ys2book`: extracting documentation from Yacas code ("literate programming")

The standard view of literate programming is that one prepares a book readable by humans, and all code is automatically extracted from the book. The focus in this approach is on writing explanations on how the code works.

The converse approach is to write primarily code and embed some documentation as comments in the code files. This approach is implemented by the script `ys2book.pl`.

This script takes a Yacas script file and extracts Yacas comments from it into another file. Usage may look like this:

```
perl ys2book.pl < file.ys > file.chapt.txt
perl ys2book.pl -strip < file.ys > file.chapt.txt
```

Here `file.ys` is the source file and `file.chapt.txt` is the output file.

Not all comments may be desirable as documentation. Formatting of comments is implemented as follows:

- Line comments starting at the beginning of line with three or more slashes ("`///`", "`////`" etc.) are exported as documentation. Line comments starting with "`//`" are not exported. Line comments that are found after some Yacas statements are also not exported. For example, if the source file contains the following lines,

  ```
  a:=1; // initialize a.
     ///// This function needs
    ///   an initializer.
  b:=1; /// also initialize b.
  ```

  then only the text

  ```
  This function needs
  an initializer.
  ```

  will appear in the output file. Note that the initial spaces are stripped from the line comment strings.

- Block comments starting with "`/**`" become documentation. The block comment must be the only content of the line. For example,

  ```
  /** documentation text */
  /**   continues
      here*/
  ```

  will export the following:

  ```
  documentation text
  continues
      here
  ```

  Note that some initial spaces have been stripped. See below for more detail about the stripping of spaces.

- Multiline comments marked with an initial asterisk "`*`" become documentation with the initial asterisk removed. For example, in the block comment

  ```
  /** This starts
   * a multiline
   * comment.
   */
  ```

  the initial `*` and any spaces before it will be removed from each line. This feature is designed to make the format easier to comprehend visually, if the documentation needs to be specially formatted.

- All other comments remain unexported code comments.
- All Yacas code, including unexported comments, is TAB-indented and printed to the output file, unless the "`-strip`" option is given. (TAB-indentation is the Yacas documentation markup for code examples.) With the "`-strip`" option, the output file will contain only exportable documentation comments and no code samples.

All exported text is printed to the output file as is, without any additional reformatting. The only change to the text is stripping of initial spaces.

Any leading spaces after the beginning of the comment sign are removed. For example,

```
/*      text */
```

will be exported as just "text" without the leading spaces. In a multiline comment, such as

```
/*    start
      of comment */
```

the leading spaces in the first line will be stripped. However, the leading spaces (and TABs) in other lines of the multiline comment block will be preserved.

Empty lines can be introduced into the documentation either as part of a multiline comment block, or as a standalone empty comments such as

```
///
////////
```

With these features it is easy to prepare the embedded documentation in the Yacas plaintext documentation format. This format requires space- and TAB-based formatting, which is mostly preserved by the script `ys2book.pl`.

`ytxt2tex`: Conversion of plain text documentation to LaTeX

An auxiliary script `ytxt2tex` converts plain text documentation to LaTeX. The script `ytxt2tex` can be used outside of the Yacas source tree to convert individual documents to LaTeX. This is useful if you would like to produce TeX documents and if you find the plain text format of the Yacas documentation more comfortable. Therefore, `ytxt2tex` is a kind of a special-purpose TeX preprocessor designed for producing Yacas documentation.

This is a standalone script; it is installed by default into `/usr/local/bin` and requires the Yacas executable also on the path, as well as the script files `book2TeX.*` and `txt2yacasdoc.pl` in the `/manmake` subdirectory of the Yacas installation tree `/usr/local/share/yacas/`. The script also requires `perl` and the Unix shell `sh`.

Limitations of this script are:

- it is impossible to include raw TeX code;
- mathematical expressions are limited to those representable using Yacas functions and operators from the standard library;
- the document comes out to be a twocolumn "report" with a certain fixed title page format, a table of contents, and an index;
- front matter is fixed ("*This is Yacas documentation... by the Yacas team... GNU Free Documentation License...*" etc.) and will most probably have to be edited by hand if you need to prepare anything other than Yacas documentation;
- all current limitations of plaintext documentation format, for example, no nested itemized/enumerated environments;
- some advanced features of index generation (e.g. the `!` separators combined with other markup) are not yet supported.

These limitations may be easily overcome by editing the resulting TeX file (but you need to know at least some TeX to do that).

The general usage pattern is

```
ytxt2tex [-o outputfile] file1.txt [file2.txt] ...
```

All source files must have extension ".txt". The command-line option `-o` specifies the name of the output TeX file. If the `-o` option is not given, the output file will be `file1.tex` (i.e. the name of the first `.txt` file with the `.tex` extension). If several `.txt` files are given, the first one must `*INCLUDE` all others.

To illustrate the usage of the script `ytxt2tex`, consider two examples.

The first example is just one plaintext file `example1.txt`. This file will have to be a "book" in itself, i.e. it will have to include a book title indented by four TAB symbols. For example:

```
*REM file: example1.txt
                Example Document Title
*REM this is a section title:
Numbers and letters
*REM here are some index entries:
*A numbers
*REM simple index entries like this are OK
*A letters

Numbers and letters are very important, etc.
```

This file `example1.txt` can be converted to a LaTeX file `example1.tex` by the following simple command:

```
ytxt2tex example1.txt
```

If the resulting file should be named something other than `example1.tex`, say `output1.tex`, then the command is

```
ytxt2tex -o output1.tex example1.txt
```

The second example is a longer "book" consisting of several plaintext files. One of these files is a "master file" and it should include all other files using the `*INCLUDE` label. The `*INCLUDE` label should contain file names without the `.txt` extension.

Suppose we have prepared the files `book1.txt`, `chapter1.txt`, and `chapter2.txt` containing the preamble text and two chapters. For example:

```
*REM this is the file "book1.txt"
*BLURB or, The Multitudinous Attempts
to Avoid Counterproductivity
                Relationships of Entities
*INCLUDE chapter1
*INCLUDE chapter2
```

The chapter files might be:

```
*REM this is the file "chapter1.txt"
        Entities and Epiphenomena
The history of the ambiguous question of
epiphenomenological discourse can be
traced to the pre-postmodern period...

*REM this is the file "chapter2.txt"
        Substrates and Superficiality
In the preceding chapter, we have thoroughly
investigated the metaphilosophical aspects of
the trans-homocentric considerations...
```

The command to create the final LaTeX file `book1.tex` is

```
ytxt2tex book1.txt chapter1.txt chapter2.txt
```

The "master file" `book1.txt` that includes all other text files must be given first. The `-o` option can be used if the final LaTeX file should be named something else than `book1.tex`. For example,

```
ytxt2tex -o MyBook.tex book1.txt chapter*.txt
```

By default, both table of contents and the index are generated. The commands to create a PostScript file out of the LaTeX file might be:

```
latex MyBook.tex
latex MyBook.tex
makeindex MyBook.idx -o MyBook.ind
latex MyBook.tex
```

Note that the resulting LaTeX file needs to be processed three times if the table of contents or index are to be used. Without a table of contents and index, it is enough to process the file with LaTeX twice.

book2txt: Conversion of existing documentation to plain text

(Note: as of version 1.0.49, all Yacas documentation is converted to plaintext format. This section is left for reference only.)

Currently, most but not all of the Yacas documentation markup functionality is implemented in the simple plaintext filter; also, documentation includes some extra HTML files. However, almost all of the reasonable markup needed to write documentation is present. Therefore it is possible to maintain most of the documentation in the plain text format described above. To convert existing Yacas documentation back to the plain text format, a script `book2txt.ys`/`book2txt.sh` can be used.

By using a command such as

```
book2txt.sh file.chapt
```

one can create a source text file `file.chapt.txt` corresponding to the Yacas documentation file `file.chapt`. For example:

```
12:51pm scriabin> book2txt.sh intro.book
True;
Out> True;
Quitting...
File 'intro.book.txt' was created.
12:51pm scriabin>
```

In the above example, the shell commands in `book2txt.sh` executed the following Yacas commands,

```
Use("book2txt.ys");
ToFile("file.chapt.txt")
  Load("file.chapt");
```

This requires that the Yacas script `book2txt.ys` be available in the current directory. The shell script `book2txt.sh` assumes that `book2txt.ys` is stored in the same directory as `book2txt.sh`.

Of course, it is possible that some features of Yacas documentation were not implemented in the script and in that case the resulting file must be edited by hand. But the purpose of the `book2txt` script is exactly this: to make a plain text source file to be edited and maintained.

Several files can be converted at once, for example:

```
book2txt.sh f1.chapt f2.chapt file3.book
```

Each file is processed by an independent Yacas session. Any errors of processing are printed on the screen.

5.3 The Yacas build system

Introduction

This chapter describes the build system of Yacas. So here you will find what happens when you give the `configure` or the `make` command, and how to change this. It will concentrate on Unix systems; other architectures are briefly covered in the final section.

As the Yacas build system is built on the GNU autotools suite, which contains both `automake` and `autoconf`, we will start with a short description of this package. Then we will turn to the various components of Yacas: the program itself, the script files, the documentation, the test suite, and so on.

As explained in the `INSTALL` file, building Yacas requires the following steps.

- (This step is only necessary if building from CVS.) Start by running the `makemake` script. This executes the `automake` and `autoconf` programs.
- Then the `configure` script should be run.
- Finally, Yacas can be built by giving the `make` command.

Both `configure` and `make` accept many options. Some of them are explained below.

The GNU autotools suite

The GNU `autotools` suite is a collection of applications to streamline the build system of other programs, like Yacas. Its two main goals are to present a consistent build procedure to users, and to assist developers in tackling portability problems.

The autotools suite consists of a number of utilities. These are developed separately, but are designed to be used together. They are

- `automake`. Its main goal is to produce a `Makefile.in` file from a high-level description in the `Makefile.am` file. The `configure` script generated by `autoconf` will later transform it in a `Makefile`. The generated Makefiles are portable and contain all the targets specified in the GNU Coding Standards document, like `all`, `install` and `clean`.
- `autoconf`. Its main goal is to produce the `configure` script. When Yacas is built, this script gathers information from the user's system, like the operating system and the location of certain programs used by the Yacas build system. This information is used in turn by the Makefiles.
- `libtool`. This utility provides a portable interface for building and using libraries. Static libraries, shared libraries and dynamically loadable modules are all supported.

The users do not need to run `automake` and `autoconf` (here, "users" refers to the people who do not want to make any changes to Yacas and includes those who just want to compile Yacas from a tar.gz source archive). They do need the `libtool` script. But the `libtool` package is included in the Yacas distribution, so the users do not need to install `libtool` themselves.

Developers do need to install `autoconf` and `automake` on their systems. But they usually do not need to run these tools directly, as the Makefiles contain the necessary commands. When the Makefiles are not present, which occurs for instance when installing afresh from the CVS repository, the `makemake` script in the root of the Yacas source tree can (and probably should) be used to invoke `automake` and `autoconf` in the right order and with the correct flags.

In the following three sections, these utilities are briefly explained. In all cases, the reader is referred to the documentation included in the `autotools` suite for more information. Another useful source of information is *GNU Autoconf, Automake, and Libtool* by Gary V. Vaughan, Ben Elliston, Tom Tromey and Ian Lance, which is published by New Riders. An online version is available from `http://sources.redhat.com/autobook`.

The automake tool

Automake is a tool to generate standard-compliant Makefiles. More precisely, `automake` uses the information in `Makefile.am` to produce a `Makefile.in` file, which will be turned into a `Makefile` by the `configure` script generated by the `autoconf` utility.

The `Makefile.am` file contains the definition of certain macros that are used by `automake`. The rest of the `Makefile.am` file is copied verbatim to the generated `Makefile.in` file.

The most important macros which are used by `automake` are the so-called *primaries*. These list the files that make up the Yacas package. For instance, in the `src` directory, the file `Makefile.am` contains the following line

```
bin_PROGRAMS = yacas
```

This is an example of the PROGRAMS primary, and says that the directory contains a program called yacas. Hence it will be built if the make all command is executed, it will be installed at make install, etc. Other useful primaries are SCRIPTS for executable scripts, HEADERS for header files, and DATA for all files which are just copied verbatim at installation time (this includes Yacas scripts).

The bin prefix in the example above says that yacas should be installed in the binary directory, as determined by the configure script. By default, this is the directory /usr/local/bin. There are also prefixes for the other directories, as well as some prefixes with different meanings: the noinst prefix says that the specified file need not be installed, and the check prefix says that the file is only needed when testing.

There are also so-called *associated variables*. The same Makefile.am contains the following variables associated to the Yacas executable:

```
yacas_SOURCES = yacasmain.cpp commandline.cpp \
      unixcommandline.cpp stdcommandline.cpp
yacas_LDADD = libyacas.a libyacasplatform.a
```

These lines tell that the executable is built from four source files (yacasmain.cpp, commandline.cpp, unixcommandline.cpp and stdcommandline.cpp) and two static libraries (libyacas.a and libyacasplatform.a).

From the information contained in these lines, automake can construct the necessary commands to go in the final Makefile. This Makefile does not only support building, testing, and installing the package, but also rolling the tar-ball for release (use make dist for this, as explained in the section *"Targets for make"*, in *this book* below). In the above example, automake can figure out that yacasmain.cpp should be included in the distribution.

Unfortunately not everything is supported that well. For instance, Yacas comes with its own documentation system, which is of course not supported by automake. So we need to tell automake how to handle these files. To specify which files should be included in the distribution, the EXTRA_DIST variable can be used. The developer should list all files to be included in the distribution that automake does not know about here. If we want to run some commands at build or installation time, we can specify them by Makefile rules in the traditional ways. Just write the rules in Makefile.am and they will be copied verbatim in the generated Makefile. Please keep in mind that the rules should work on a wide variety of platforms in order to retain portability. In particular,

- remember that the source files may be in a different directory, if the user decides to use separate source and build trees (see *"The configure script"*, in *this book*);
- only use the automatic variable $< in suffix rules, as Solaris make does not define $< correctly in other rules;
- do not use pattern rules (like %.o: %.c) as they are not supported on all platforms, use suffix rules (like .c.o:) instead.

We currently assume automake version 1.4 or later. Note that version 1.5 breaks backward compatibility and should be avoided. Version 1.6 contains some useful additions, like the nobase prefix and the possibility to define new prefixes, so at a certain point we may require version 1.6.

For more information about automake, the reader is referred to the documentation that comes with the package.

The autoconf tool

Autoconf is a tool to generate portable shell scripts that users can run to configure the package (in our case Yacas) for their system. It reads the file configure.in and produces the configure script. The latter script can be run by the user to prepare for building Yacas.

The configure.in file consists of standard shell code, interspersed with special macros defined by the autoconf package. These can be recognized by the AC_ prefix.

As the configure.in file only rarely needs to be changed, we will only describe the autoconf tool by one example.

This ends the brief description of autoconf. For more information, the reader is referred to the documentation that comes with the package.

We currently assume autoconf version 2.13 or later.

The configure script

The configure script is run by the user to prepare the Yacas package for the build and installation process. It examines the user's system and the options passed to the script by the user, and generates suitable Makefiles. Furthermore, it generates yacas.spec which is used to build a package in Red Hat's .rpm format, and the C header file config.h.

A nice feature is the possibility to build in a different directory than the source directory by simply running configure from that directory. This not only prevents the source directory from being cluttered up by object files and so on, but it also enables the user to build for different architectures in different directories or to have the source in a read-only directory. For example, suppose that the Yacas source is installed under /mnt/cdrom/src/yacas and that you want to build Yacas under /tmp/build-yacas. This is achieved by the following commands

```
mkdir /tmp/build-yacas
cd /tmp/build-yacas
/mnt/cdrom/src/yacas/configure
make
```

A list of options accepted by the configure script can be retrieved by invoking it with the help option

```
./configure --help
```

The most important is the prefix option, which influences where everything will be installed. The default is /usr/local, meaning that for instance the yacas executable is installed as /usr/local/bin/yacas. To change this in /usr/bin/yacas, invoke the script as follows

```
./configure --prefix=/usr
```

Other options to configure enable the user to fine-tune the location where the various files should be installed.

We will not describe the common configure options which are shared by all packages, but will restrict ourselves to the options exclusively used by Yacas.

- with-numlib=LIB. This option specifies the arbitrary-precision library to use. The supported values for LIB so far are native (the default).
- enable-debug. Build a version of Yacas suitable for debugging.
- enable-archive. Build and install the library archive. The library archive resides in the file scripts.dat and contains all the Yacas scripts in compressed form.
- enable-server. Build a version of the Yacas executable which accepts the --server flag, which puts Yacas in server mode (see *"Command-line options"*, in *Introduction to Yacas, Chapter 3, Section 1*)

The following three options pertain to the extensive documentation that comes with Yacas. By default, only HTML documentation is generated.

- `disable-html-doc`. Do not generate documentation in HTML format.
- `enable-ps-doc`. Generate documentation in PostScript format. This requires the latex suite.
- `enable-pdf-doc`. Generate documentation in PDF format. This also requires the latex suite.

Then there are three options describing where to install various parts of the Yacas package.

- `with-script-dir=DIR`. Install the Yacas script files, which have extension `.ys`, in the specified directory. By default, the script files are installed in `DATADIR/yacas`, where `DATADIR` defaults to `PREFIX/share`. In turn, `PREFIX` refers to the value of the `prefix` option described above; the default value is `/usr/local`. The conclusion is that the script files by default end up in `/usr/local/share/yacas`.
- `with-html-dir=DIR`. This specifies where to install the HTML documentation for the Yacas package. The default is a subdirectory named `documentation` of the directory in which the Yacas script files are installed.
- `with-ps-dir=DIR`. Where to install the PostScript and PDF documentation. This defaults to the same directory as for the documentation in HTML format.

Furthermore, the opposites of the above options (eg. `disable-server`) are also recognized.

Targets for make

One of the advantages of using the autotool suite to generate Makefiles, is that the resulting Makefiles support a variety of targets. Here is a partial list of them.

- `all`. This is the default target, so instead of `make all` one can just type `make`. It builds the executables, which includes `yacas`, and the documentation. By default, only documentation in HTML format will be built, but this can be changed by the `configure` script.
- `install`. Compile the executables, and install them together with the libraries, Yacas scripts, and documentation so that the Yacas program can be run.
- `install-strip`. Same as `install`, but also strip the debug information from the installed executables.
- `uninstall`. Remove the installed files.
- `clean`. Delete files that are created during the build process, for instance `.o` files.
- `distclean`. Delete files that are created during either the configuration or the build process. This leaves only the files in that were included in the distribution.
- `dist`. Create a tar-ball (a gzipped tar archive) ready for distribution.
- `check` (or its synonym `test`). Test whether Yacas works correctly.
- `installcheck`. Test whether Yacas is installed correctly. The difference with the `check` target is that that one tests the *built* version of Yacas, while the `installcheck` target tests the *installed* version.
- `distcheck`. Check whether Yacas is ready for distribution. This creates a tar-ball, unpacks it in a different directory, compiles Yacas with default options, installs it in a temporary directory, and finally checks that the installed version works correctly. Extra flags to be passed to the `configure` script (eg. `--enable-archive`) can be put in the `DISTCHECK_CONFIGURE_FLAGS` environment variable.

The Yacas executable

The main executable is called `yacas` and resides in the `src` directory.

Conceptually, the build process consists of the following parts:

- Generating the source files. Almost all source files were written by the Yacas development team, but the following are generated automatically: `version.h` (generated from `configure.in`), `fastprimes.c` (generated by the `mkfastprimes` program), and `core_yacasmain.h` (generated from `yacasmain.cpp`).
- Building the executables. Besides the Yacas executables, we also build `mkfastprimes` and `gencorefunctions` (which are needed to generate some source files), and the `codereviewer` executable (which does a static analysis of code and can in some circumstances also fix the code where the coding style is not adhered to).

The Yacas script files

The Yacas script files with extension `ys` can be found in the `scripts/` directory and its "repository" subdirectories.

All script files and `.def` files are listed in `Makefile.am`. This definition is used to generate the `packages.ys` file, which contains a list of all `.def` files. The `corefunctions.ys` file is also automatically generated. This is done by running the `gencorefunctions` program, which resides in the `src/` directory.

At installation time, all the script files and `.def` files that are listed in `Makefile.am` are copied. During this installation, the directory structure should be preserved.

The `check` target checks that all the script files and `.def` files listed in the `Makefile.am` are indeed present, and vice versa, that all files present in the `scripts/` hierarchy are indeed listed.

The documentation

The documentation system for Yacas is explained in *Essays on Yacas, Chapter 5, Section 2*.

The documentation is generated from plain text source files with the `txt` extension in the `manmake` directory. The source files are converted to Yacas code by the `txt2yacasdoc.pl` script. The result is a set of Yacas source files containing the text of the documentation. These Yacas source files are not well suited for human reading and must be further processed to obtain the books in HTML and PS/PDF formats.

To generate the books in HTML format, the Yacas source files are processed by the Yacas interpreter using the scripts `manualmaker` and `indexmaker`. The script `indexmaker` creates the file `books.html` with the top-level index to all HTML documentation books. The script `manualmaker` creates the HTML files with the book text (both the framed and the non-framed versions are written out). It is also possible to generate more than one HTML book at once. The two reference manuals are generated together to produce the full hyperlinked index to all commands.

To generate the books in PostScript and PDF format, the `book2TeX.sh` script is used. This script converts the Yacas source files to TEX files. They can then be processed by the standard TEX tools.

By default, only the HTML-formatted documents are generated. To change this, use the options `disable-html-doc`, `enable-ps-doc` and `enable-pdf-doc` in the `configure` script. The `with-html-dir` and `with-ps-dir` options can be used to tell where the documentation should be installed.

At the moment, the `Makefile.am` for the documentation is rather messy. For this reason, we do not describe it in detail. Instead, we just point out some special features:

- The Yacas interpreter and `perl` are needed to generate the documentation.
- The section *"Full listing of core functions"*, in *The Yacas Programmer's Function Reference, Chapter 5, Section 1* is generated from the Yacas source by the `gencorefunctions` program, which resides in the `src` directory.
- The Yacas code from the section *M. Wester's CAS benchmark and Yacas*, built from `wester-1994.chapt.txt`, is extracted with the `book2ys.sh` script. This code is used for testing (see below).
- All the scripts described can be found in the `manmake` directory.

The test suite

The Yacas distribution contains a (hopefully comprehensive) test suite, which can be used to check whether Yacas is (still) functioning correctly. The command `make check` tests the latest compiled version of Yacas. To test the executable that is installed on your system, use the command `make installcheck`.

The `yts` files in the `tests` directory contain the tests for different subsystems of Yacas. For instance, the file `arithmetic.yts` contains tests for the basic arithmetic functions; the first test is to check that `3+2` evaluates to `5`. All the test files are listed in the `TESTFILES` variable. This variable can be overridden. For example, the command

```
make TESTFILES='comments.yts complex.yts' clean
```

only runs the test scripts `comments.yts` and `complex.yts`.

The shell script `test-yacas` does the actual testing. It runs the scripts listed in the `TESTFILES` variable through the Yacas executable. The tests that take a long time are at the end of the list, so that these tests are performed last. The output of the tests is collected in the file `testresult.txt`. The test is considered to be failed, if the exit code is nonzero (which happens for instance if Yacas crashes) or if either of the strings `******` (six stars) or `Error` appear in the output.

A special case is the last test, which goes through the problems put forward by Michael Wester (see the section *M. Wester's CAS benchmark and Yacas*, in *Essays on Yacas, Chapter 2*). The commands for this tests are not in a `yts` file in the `tests` directory, but are extracted from the documentation (see *the immediately preceding section*, in *this book*).

The library archive

The library archive is a file, commonly called `scripts.dat`, which contains all the scripts in compressed form. If Yacas is started with the `--archive` flag, it uses the contents of this file. This is useful for binary releases, as one needs only two files: the Yacas executable and the library archive.

Support for the library archive resides in the `src/` directory. It contains the code for the `compressor` program. This program generates the library archive.

The archive is only built if the `enable-archive` option is passed to the `configure` script. In that case, the `compressor` program is built and run. At installation time, the generated archive is installed in the library directory (`/usr/local/lib` by default). There is also a test to check that the generated archive in fact works.

Chapter 6

Designing modules in the Yacas scripting language

6.1 Introduction

For any software project where the source code grows to a substantial amount of different modules, there needs to be a way to define interfaces between the modules, and a way to make sure the modules don't interact with the environment in an unintended way.

One hallmark of a mature programming language is that it supports modules, and a way to define its interface while hiding the internals of the module. This section describes the mechanisms for doing so in the Yacas scripting language.

6.2 Demonstration of the problem

Unintentional interactions between two modules typically happen when the two modules accidentally share a common "global" resource, and there should be a mechanism to guarantee that this will not happen.

The following piece of code is a little example that demonstrates the problem:

```
SetExpand(fn_IsString) <-- [expand:=fn;];
ram(x_IsList)_(expand != "") <-- ramlocal(x);
expand:="";
ramlocal(x) := Map(expand,{x});
```

This little bit of code defines a function `ram` that calls the function `Map`, passing the argument passed if it is a string, and if the function to be mapped was set with the `SetExpand` function. It contains the following flaws:

1. `expand` is a global variable with a rather generic name, one that another module might decide to use.
2. `ramlocal` was intended to be used from within this module only, and doesn't check for correctness of arguments (a small speed up optimization that can be used for routines that get called often). As it is, it can be called from other parts, or even the command line.
3. the function `ramlocal` has one parameter, named `x`, which is also generic (and might be used in the expression passed in to the function), and `ramlocal` calls `Map`, which calls `Eval` on the arguments.

The above code can be entered into a file and loaded from the command line at leisure. Now, consider the following command line interaction after loading the file with the above code in it:

```
In> ramlocal(a)
In function "Length" :
bad argument number 1 (counting from 1)
Argument matrix[1] evaluated to a
In function call  Length(a)
CommandLine(1) : Argument is not a list
```

We called `ramlocal` here, which should not have been allowed.

```
In> ram(a)
Out> ram(a);
```

The function `ram` checks that the correct arguments are passed in and that `SetExpand` was called, so it will not evaluate if these requirements are not met.

Here are some lines showing the functionality of this code as it was intended to be used:

```
In> SetExpand("Sin")
Out> "Sin";
In> ram({1,2,3})
Out> {Sin(1),Sin(2),Sin(3)};
```

The following piece of code forces the functionality to break by passing in an expression containing the variable `x`, which is also used as a parameter name to `ramlocal`.

```
In> ram({a,b,c})
Out> {Sin(a),Sin(b),Sin(c)};
In> ram({x,y,z})
Out> {{Sin(x),Sin(y),Sin(z)},Sin(y),Sin(z)};
```

This result is obviously wrong, comparing it to the call above. The following shows that the global variable `expand` is exposed to its environment:

```
In> expand
Out> "Sin";
```

6.3 Declaring resources to be local to the module

The solution to the problem is `LocalSymbols`, which changes every symbol with a specified name to a unique name that could never be entered by the user on the command line and guarantees that it can never interact with the rest of the system. The following code snippet is the same as the above, with the correct use of `LocalSymbols`:

```
LocalSymbols(x,expand,ramlocal) [
  SetExpand(fn_IsString) <-- [expand:=fn;];
  ram(x_IsList)_(expand != "") <-- ramlocal(x);
  expand:="";
  ramlocal(x) := Map(expand,{x});
];
```

This version of the same code declares the symbols `x`, `expand` and `ramlocal` to be local to this module.

With this the interaction becomes a little bit more predictable:

```
In> ramlocal(a)
Out> ramlocal(a);
In> ram(a)
Out> ram(a);
In> SetExpand("Sin")
Out> "Sin";
In> ram({1,2,3})
Out> {Sin(1),Sin(2),Sin(3)};
In> ram({a,b,c})
Out> {Sin(a),Sin(b),Sin(c)};
In> ram({x,y,z})
Out> {Sin(x),Sin(y),Sin(z)};
In> expand
Out> expand;
```

6.4 When to use and when not to use `LocalSymbols`

The `LocalSymbols` should ideally be used for every global variable, for functions that can only be useful within the module and thus should not be used by other parts of the system, and for local variables that run the risk of being passed into functions like `Eval`, `Apply`, `Map`, etc. (functions that re-evaluate expressions).

A rigorous solution to this is to make all parameters to functions and global variables local symbols by default, but this might cause problems when this is not required, or even wanted, behaviour.

The system will never be able to second-guess which function calls can be exposed to the outside world, and which ones should stay local to the system. It also goes against a design rule of Yacas: everything is possible, but not obligatory. This is important at moments when functionality is not wanted, as it can be hard to disable functionality when the system does it automatically.

There are more caveats: if a local variable is made unique with `LocalSymbols`, other routines can not reach it by using the `UnFence` construct. This means that `LocalSymbols` is not always wanted.

Also, the entire expression on which the `LocalSymbols` command works is copied and modified before being evaluated, making loading time a little slower. This is not a big problem, because the speed hit is usually during calculations, not during loading, but it is best to keep this in mind and keep the code passed to `LocalSymbols` concise.

Chapter 7

The Yacas arithmetic library

7.1 Introduction

YACAS comes with its own arbitrary-precision arithmetic library, to reduce dependencies on other software.

This part describes how the arithmetic library is embedded into YACAS.

7.2 The link between the interpreter and the arithmetic library

The YACAS interpreter has the concept of an *atom*, an object which has a string representation. Numbers are also atoms and are initially entered into Yacas as decimal strings. [1] As soon as a calculation needs to be performed, the string representation is used to construct an object representing the number, in an internal representation that the arithmetic library can work with.

The basic layout is as follows: there is one class `BigNumber` that offers basic numerical functions, arithmetic operations such as addition and multiplication, through a set of class methods. Integers and floating-point numbers are handled by the same class.

The `BigNumber` class delegates the actual arithmetic operations to the auxiliary classes `BigInt` and `BigFloat`. These two classes are direct wrappers of an underlying arithmetic library. The library implements a particular internal representation of numbers.

The responsibility of the class `BigNumber` is to perform precision tracking, floating-point formatting, error reporting, type checking and so on, while `BigInt` and `BigFloat` only concern themselves with low-level arithmetic operations on integer and floating-point numbers respectively. In this way Yacas isolates higher-level features like precision tracking from the lower-level arithmetic operations. The number objects in a library should only be able to convert themselves to/from a string and perform basic arithmetic. It should be easy to wrap a generic arithmetic library into a `BigNumber` implementation.

It is impossible to have several alternative number libraries operating at the same time. [In principle, one might write the classes `BigInt` and `BigFloat` as wrappers of two different arithmetic libraries, one for integers and the other for floats, but at any rate one cannot have two different libraries for integers at the same time.] Having several libraries in the same Yacas session does not seem to be very useful; it would also incur a lot of overhead because one would have to convert the numbers from one internal library representation to another. For performance benchmarking or for testing purposes one can compile separate versions of YACAS configured with different arithmetic libraries.

To embed an arbitrary-precision arithmetic library into Yacas, one needs to write two wrapper classes, `BigInt` and `BigFloat`. (Alternatively, one could write a full `BigNumber` wrapper class but that would result in code duplication unless the library happens to implement a large portion of the `BigNumber` API. There is already a reference implementation of `BigNumber` through `BigInt` and `BigFloat` in the file `numbers.cpp`.) The required API for the `BigNumber` class is described below.

7.3 Interface of the `BigNumber` class

The following C++ code demonstrates how to use the objects of the `BigNumber` class.

```
// Calculate z=x+y where x=10 and y=15
BigNumber x("10",100,10);
BigNumber y("15",100,10);
BigNumber z;
z.Add(x,y,10));
// cast the result to a string
LispString  str;
z.ToString(str,10);
```

The behaviour is such that in the above example `z` will contain the result of adding `x` and `y`, without modifying `x` or `y`. This is equivalent to `z:=x+y` in Yacas.

A calculation might modify one of its arguments. This might happen when one argument passed in is actually the object performing the calculation itself. For example, if a calculation

```
x.Add(x,y);
```

were issued, the result would be assigned to `x`, and the old value of `x` is deleted. This is equivalent to the Yacas code `x:=x+y`. In this case a specific implementation might opt to perform the operation destructively ("in-place"). Some operations can be performed much more efficiently in-place, without copying the arguments. Among them are for example `Negate`, `Add`, `ShiftLeft`, `ShiftRight`.

Therefore, all class methods of `BigNumber` that allow a `BigNumber` object as an argument should behave correctly when called destructively on the same `BigNumber` object. The result must be exactly the same as if all arguments were copied to temporary locations before performing tasks on them, with no other side-effects. For instance, if the specific object representing the number inside the numeric class is shared with other objects, it should not allow the destructive operation, as then other objects might start behaving differently.

[1] There are functions to work with numbers in non-decimal bases, but direct input/output of numbers is supported only in decimal notation.

The basic arithmetic class `BigNumber` defines some simple arithmetic operations, through which other more elaborate functions can be built. Particular implementations of the multiple-precision library are wrapped by the `BigNumber` class, and the rest of the Yacas core should only use the `BigNumber` API.

This API will not be completely exposed to Yacas scripts, because some of these functions are too low-level. Among the low-level functions, only those that are very useful for optimization will be available to the Yacas scripts. (For the functions that seem to be useful for Yacas, suggested Yacas bindings are given below.) But the full API will be available to C++ plugins, so that multiple-precision algorithms could be efficiently implemented when performance is critical. Intermediate-level arithmetic functions such as `MathAdd`, `MathDiv`, `MathMod` and so on could be implemented either in the Yacas core or in plugins, through this low-level API. The library scripts will be able to transform numerical expressions such as `x:=y+z` into calls of these intermediate-level functions.

Here we list the basic arithmetic operations that need to be implemented by a multiple-precision class `BigNumber`. The operations are divided into several categories for convenience. Equivalent Yacas script code is given, as well as examples of C++ usage.

1. Input/output operations.

- `BigNumber::SetTo` – Construct a number from a string in given base. The format is the standard integer, fixed-point and floating-point representations of numbers. When the string does not contain the period character "`.`" or the exponent character "`e`" (the exponent character "`@`" should be used for base > 10), the result is an integer number and the precision argument is ignored. Otherwise, the result is a floating-point number rounded to a given number of *base digits*. C++:

  ```
  x.SetTo("2.e-19", 100, 10);
  ```

  Here we encounter a problem of ambiguous hexadecimal exponent:

  ```
  x.SetTo("2a8c.e2", 100, 16);
  ```

  It is not clear whether the above number is in exponential notation or not. But this is hopefully not a frequently encountered situation. We may assume that the exponent character for base > 10 is "`@`" and not "`e`".

- The same function is overloaded to construct a number from a platform number (a 32-bit integer or a double precision value). C++:

  ```
  x.SetTo(12345); y.SetTo(-0.001);
  ```

- `BigNumber::ToString` – Print a number to a string in a given precision and in a given base. The precision is given as the number of digits in the given base. The value should be rounded to that number of significant base digits. (Integers are printed exactly, regardless of the given precision.) C++:

  ```
  x.ToString(buffer, 200, 16); // hexadecimal
  x.ToString(buffer, 40, 10); // decimal
  ```

- `BigNumber::Double` – Obtain an approximate representation of `x` as double-precision value. (The conversion may cause overflow or underflow, in which case the result is undefined.) C++:

  ```
  double a=x.Double();
  ```

2. Basic object manipulation.

These operations, as a rule, do not need to change the numerical value of the object.

- `BigNumber::SetTo` – Copy a number, `x := y`. This operation should copy the numerical value exactly, without change. C++:

  ```
  x.SetTo(y);
  ```

- `BigNumber::Equals` – Compare two numbers for equality, `x = y`. C++:

  ```
  x.Equals(y)==true;
  ```

  Yacas:

  ```
  MathEquals(x,y)
  ```

  The values are compared arithmetically, their internal precision may differ, and integers may be compared to floats. Two floats are considered "equal" when their values coincide within their precision. It is only guaranteed that `Equals` returns true for equal integers, for an integer and a floating-point number with the same integer value, and for two exactly bit-by-bit equal floating-point numbers. Floating-point comparison may be unreliable due to roundoff error and particular internal representations. So it may happen that after `y:=x+1; y:=y-1;` the comparison

  ```
  y.Equals(x)
  ```

  will return `false`, although such cases should be rare.

- `BigNumber::IsInt` – Check whether the number `x` is of integer or floating type. (Both types are represented by the same class `BigNumber`, and we need to be able to distinguish them.) C++:

  ```
  x.IsInt()==true;
  ```

  Yacas: part of the implementation of `IsInteger(x)`.

- `BigNumber::IsIntValue` – Check whether the number `x` has an integer value. (Not the same as the previous function, because a floating-point type can also have an integer value.) Always returns `true` on objects of integer type. C++:

  ```
  x.IsIntValue()==true;
  ```

  Yacas:

  ```
  FloatIsInt(x)
  ```

- `BigNumber::BecomeInt`, `BigNumber::BecomeFloat` – Change the type of a number from integer to float without changing the numerical value. The precision is either set automatically (to enough digits to hold the integer), or explicitly to a given number of bits. (Roundoff might occur.) Change the type from float to integer, rounding off if necessary. C++:

  ```
  x.BecomeInt(); x.BecomeFloat();
  x.BecomeFloat(100);
  ```

3. Basic arithmetic operations.

Note that here "precision" always means the number of significant *bits*, i.e. digits in the base 2, *not decimal digits*.

- `BigNumber::LessThan` – Compare two objects, `x<y`. Returns `true` if the numerical comparison holds, regardless of the value types (integer or float). If the numbers are equal up to their precision, the comparison returns `false`. C++:

```
x.LessThan(y)==true;
```

Yacas:

```
LessThan(x,y)
```

- `BigNumber::Floor` – Compute the integer part of a number, `x := Floor(y)`. This function should round toward algebraically smaller integers, as usual. C++:

```
x.Floor(y);
```

Yacas:

```
MathFloor(x)
```

If there are enough digits in `x` to compute its integer part, then the result is an exact integer. Otherwise the floating-point value `x` is returned unchanged and an error message may be printed.

- `BigNumber::GetExactBits` – Report the current precision of a number `x` in bits. C++:

```
prec=x.GetExactBits();
```

Yacas:

```
GetExactBits(x)
```

Every floating-point number contains information about how many significant bits of mantissa it currently has. A particular implementation may hold more bits for convenience but the additional bits may be incorrect. [Integer numbers are always exact and do not have a concept of precision. The function `GetExactBits` should not be used on integers; it will return a meaningless result.] The precision of a number object is changed automatically by arithmetic operations, by conversions from strings (to the given precision), or manually by the function `SetExactBits`. It is not strictly guaranteed that `GetExactBits` returns the number of correct bits. Rather, this number of bits is intended as rough lower bound of the real achieved precision. (It is difficult to accurately track the round-off errors accumulated after many operations, without a time-consuming interval arithmetic or another similar technique.) Note: the number of bits is a platform signed integer (C++ type `long`).

- `BigNumber::SetExactBits` – Set the precision of a number `x` *and truncate* (or expand) it to a given floating-point precision of `n` bits. This function has an effect of converting the number to the floating-point type with `n` significant bits of mantissa. The `BigNumber` object is changed. [No effect on integers.] Note that the `Floor` function is not similar to `SetExactBits` because 1) `Floor` always converts to an integer value while `SetExactBits` converts to a floating-point value, 2) `Floor` always decreases the number while `SetExactBits` tries to find the closest approximation. For example, if $x = -1123.38$ then `x.SetExactBits(1)` should return "`-1024.`" which is the best one-bit floating-point approximation. However, `Floor(-1123.38)` returns `-1124` (the largest integer not greater than `-1123.38`). C++:

```
x.SetExactBits(300);
```

Yacas:

```
SetExactBits(x, 300)
```

Note: the number of bits is a platform signed integer (C++ type `long`).

- `BigNumber::Add` – Add two numbers, `x := y+z`, at given precision. C++:

```
x.Add(y,z, 300);
```

Yacas:

```
MathAdd(x,y)
```

When subtracting almost equal numbers, a loss of precision will occur. The precision of the result will be adjusted accordingly.

- `BigNumber::Negate` – Negate a number, `x := -y`. C++:

```
x.Negate(y);
```

Yacas:

```
MathNegate(x)
```

- `BigNumber::Multiply` – Multiply two numbers, `x := y*z`, at given precision. C++:

```
x.Multiply(y,z, 300);
```

Yacas:

```
MathMultiply(x,y)
```

- `BigNumber::Divide` – Divide two numbers, `x := y/z`, at given precision. (Integers are divided exactly as integers and the "precision" argument is ignored.) C++:

```
x.Divide(y,z, 300);
```

Yacas:

```
MathDivide(x,y)
```

4. Auxiliary operations.

Some additional operations useful for optimization purposes. These operations can be efficiently implemented with a binary-based internal representation of big numbers.

- `BigNumber::IsSmall` – Check whether the number `x` fits into a platform type `long` or `double`. (Optimization of comparison.) This test should helps avoid unnecessary calculations with big numbers. Note that the semantics of this operation is different for integers and for floats. An integer is "small" only when it fits into a platform `long` integer. A float is "small" when it can be approximated by a platform `double` (that is, when its decimal exponent is smaller than 1021). For example, a `BigNumber` representing π to 1000 digits is "small" because it can be approximated by a platform float. C++:

```
x.IsSmall()==true;
```

Yacas:

```
MathIsSmall(x)
```

- `BigNumber::MultiplyAdd` – Multiply two numbers and add to the third, `x := x+y*z`, at given precision. (Optimization of a frequently used operation.) C++:

```
x.MultiplyAdd(y,z, 300);
```

Yacas:

```
MathMultiplyAdd(x,y,z)
```

- `BigNumber::Mod` – Obtain the remainder modulo an integer, `x:=Mod(y,n)`. C++:

```
x.Mod(y,n);
```

Yacas:

```
MathMod(x,n)
```

(Optimization of integer division, important for number theory applications.) The integer modulus `n` is a big number. The function is undefined for floating-point numbers.

- `BigNumber::Sign` – Obtain the sign of the number `x` (result is `-1`, `0` or `1`). (Optimization of comparison with 0.) C++:

```
int sign_of_x = x.Sign();
```

Yacas:

```
MathSign(x)
```

- `BigNumber::BitCount` – Obtain the integer part of the binary logarithm of the absolute value of `x`. For integers, this function counts the significant bits, i.e. the number of bits needed to represent the integer. This function is not to be confused with the number of bits that are set to 1, sometimes called the "population count" of an integer number. The population count of 4 (binary "100") is 1, and the bit count of 4 is 3.

For floating-point numbers, `BitCount` should return the binary exponent of the number (with sign), like the integer output of the standard C function `frexp`. More formally: if $n = \mathrm{BitCount}\,(x)$, and $x \neq 0$, then $\frac{1}{2} \leq |x| \cdot 2^{-n} < 1$. The bit count of an integer or a floating 0 is arbitrarily defined to be 1. (Optimization of the binary logarithm.) C++:

```
x.BitCount();
```

Yacas:

```
MathBitCount(x)
```

Note: the return type of the bit count is a platform signed integer (C++ type `long`).

- `BigNumber::ShiftLeft`, `BigNumber::ShiftRight` – Bitshift the number (multiply or divide by the n-th power of 2), `x := y >> n`, `x := y << n`. For integers, this operation can be efficiently implemented because it has hardware support. For floats, this operation is usually also much more efficient than multiplication or division by 2 (cf. the standard C function `ldexp`). (Optimization of multiplication and division by a power of 2.) Note that the shift amount is a platform signed integer (C++ type `long`). C++:

```
x.ShiftLeft(y, n); x.ShiftRight(y, n);
```

Yacas:

```
ShiftLeft(x,n); ShiftRight(x,n);
```

- `BigNumber::BitAnd`, `BigNumber::BitOr`, `BigNumber::BitXor`, `BigNumber::BitNot` – Perform bitwise arithmetic, like in C: `x = y&z`, `x = y|z`, `x = y^z`, `x = ~y`. This should be implemented only for integers. Integer values are interpreted as bit sequences starting from the least significant bit. (Optimization of operations on bit streams and some arithmetic involving powers of 2.) C++:

```
x.BitAnd(y,z); x.BitOr(y,z);
x.BitXor(y,z); x.BitNot(y);
```

Yacas:

```
BitAnd(x,y); BitOr(y,z);
BitXor(y,z); BitNot(y);
```

The API includes only the most basic operations. All other mathematical functions such as GCD, power, logarithm, cosine and so on, can be efficiently implemented using this basic interface.

Note that generally the arithmetic functions will set the type of the resulting object to the type of the result of the operation. For example, operations that only apply to integers (`Mod`, `BitAnd` etc.) will set the type of the resulting object to integer if it is a float. The results of these operations on non-integer arguments are undefined.

7.4 Precision of arithmetic operations

All operations on integers are exact. Integers must grow or shrink when necessary, limited only by system memory. But floating-point numbers need some precision management.

In some arithmetic operations (add, multiply, divide) the working precision is given explicitly. For example,

```
x.Add(y,z,100)
```

will add `y` to `z` and put the result into `x`, truncating it to at most 100 bits of mantissa, if necessary. (The precision is given in bits, not in decimal digits, because when dealing with low-level operations it is much more natural to think in terms of bits.) If the numbers `y`, `z` have fewer than 100 bits of mantissa each, then their sum will not be precise to all 100 digits. That is fine; but it is important that the sum should not contain *more* than 100 digits. Floating-point values, unlike integers, only grow up to the given number of significant bits and then a round-off *must* occur. Otherwise we will be wasting a lot of time on computations with many meaningless digits.

Automatic precision tracking

The precision of arithmetic operations on floating-point numbers can be maintained automatically. A rigorous way to do it would be to represent each imprecise real number x by an interval with rational bounds within which x is guaranteed to be. This is usually called "interval arithmetic." A result of an interval-arithmetic calculation is "exact" in the sense that the actual (unknown) number x is *always* within the resulting interval. However, interval arithmetic is computationally expensive and at any rate the width of the resulting interval is not guaranteed to be small enough for a particular application.

For the Yacas arithmetic library, a "poor man's interval arithmetic" is proposed where the precision is represented by the "number of correct bits". The precision is not tracked exactly but almost always adequately. The purpose of this kind of rough precision tracking is to catch a critical roundoff error or to indicate an unexpected loss of precision in numerical calculations.

Suppose we have two floating-point numbers x and y and we know that they have certain numbers of correct mantissa bits, say m and n. In other words, x is an approximation to an unknown real number $x' = x\,(1+\delta)$ and we know that $|\delta| < 2^{-m}$; and similarly $y' = y\,(1+\epsilon)$ with $|\epsilon| < 2^{-n}$. Here δ and ϵ are the relative errors for x and y. Typically δ and ϵ are much smaller than 1.

Suppose that every floating-point number knows the number of its correct digits. We can symbolically represent such numbers as pairs `{x,m}` or `{y,n}`. When we perform an arithmetic operation on numbers, we need to update the precision component as well.

Now we shall consider the basic arithmetic operations to see how the precision is updated.

Multiplication

If we need to multiply x and y, the correct answer is $x'y'$ but we only know an approximation to it, xy. We can estimate the precision by $x'y' = xy(1+\delta)(1+\epsilon)$ and it follows that the relative precision is at most $\delta + \epsilon$. But we only represent the relative errors by the number of bits. The whole idea of the simplified precision tracking is to avoid costly operations connected with precision. So instead of tracking the number $\delta + \epsilon$ exactly, we represent it roughly: either set the error of xy to the larger of the errors of x and y, or double the error.

More formally, we have the estimates $|\delta| < 2^{-m}$, $|\epsilon| < 2^{-n}$ and we need a similar estimate $|r| < 2^{-p}$ for $r = \delta + \epsilon$.

If the two numbers x and y have the same number of correct bits, we should double the error (i.e. decrease the number of significant bits by 1). But if they don't have the same number of bits, we cannot really estimate the error very well. To be on the safe side, we might double the error if the numbers x and y have almost the same number of significant bits, and leave the error constant if the numbers of significant bits of x and y are very different.

The answer expressed as a formula is $p = \min(m,n)$ if $|m-n| \geq D$ and $p = \min(m,n) - 1$ otherwise. Here D is a constant that expresses our tolerance for error. In the current implementation, $D = 1$.

If one of the operands is a floating zero x={`0.,m`} (see below) and x={`x,n`}, then $p = m - \mathrm{BitCount}(x) + 1$. This is the same formula as above, if we pretend that the bit count of {`0.,m`} is equal to $1 - m$.

Division

Division is multiplication by the inverse number. When we take the inverse of $x(1+\delta)$, we obtain approximately $\frac{1}{x}(1-\delta)$. The relative precision does not change when we take the inverse. So the handling of precision is exactly the same as for the multiplication.

Addition

Addition is more complicated because the absolute rather than the relative precision plays the main role, and because there may be roundoff errors associated with subtracting almost equal numbers.

Formally, we have the relative precision r of $x + y$ as

$$r = \frac{\delta x + \epsilon y}{x + y}.$$

We have the bounds on δ and ϵ:

$$\left(|\delta| < 2^{-m}, |\epsilon| < 2^{-n}\right),$$

and we need to find a bit bound on r, i.e. an integer p such that $|r| < 2^{-p}$. But we cannot estimate p without first computing $x + y$ and analyzing the relative magnitude of x and y. To perform this estimate, we need to use the bit counts of x and y and on $x + y$. Let these bit counts be a, b and c, so that $|x| < 2^a$, $|y| < 2^b$, and $2^{c-1} \leq |x + y| < 2^c$. (At first we assume that $x \neq 0$, $y \neq 0$, and $x + y \neq 0$.) Now we can estimate r as

$$r \leq \left|\frac{x \cdot 2^{-m}}{x+y}\right| + \left|\frac{y \cdot 2^{-n}}{x+y}\right| \leq 2^{a+1-m-c} + 2^{b+1-n-c}.$$

This is formally similar to multiplying two numbers with $a + 1 - m - c$ and $b + 1 - m - c$ correct bits. As in the case of multiplication, we may take the minimum of the two numbers, or double one of them if they are almost equal.

Note that there is one important case when we can estimate the precision better than this. Suppose x and y have the same sign; then there is no cancellation when we compute $x + y$. The above formula for r gives an estimate

$$r < \max(|\delta|, |\epsilon|)$$

and therefore the precision of the result is at least $p = \min(m,n)$.

If one of the operands is a floating zero represented by x={`0.,m`} (see below), then the calculation of the error is formally the same as in the case x={`1.,m`}. This is as if the bit count of {`0.,m`} were equal to 1 (unlike the case of multiplication).

Finally, if the sum $x + y$ is a floating zero but $x \neq 0$ and $y \neq 0$, then it must be that $a = b$. In that case we represent $x + y$ as {`0.,p`}, where $p = \min(m,n) - a$.

Computations with a given target precision

Using these rules, we can maintain a bound on the numerical errors of all calculations. But sometimes we know in advance that we shall not be needing any more than a certain number of digits of the answer, and we would like to avoid an unnecessarily high precision and reduce the computation time. How can we combine an explicitly specified precision, for example, in the function

```
x.Add(y,z,100)
```

with the automatic precision tracking?

We should truncate one or both of the arguments to a smaller precision before starting the operation. For the multiplication as well as for the addition, the precision tracking involves a comparison of two binary exponents 2^{-g} and 2^{-h} to obtain an estimate on $2^{-g} + 2^{-h}$. Here g and h are some integers that are easy to obtain during the computation. For instance, the multiplication involves $g = m$ and $h = n$. This comparison will immediately show which of the arguments dominates the error.

The ideal situation would be when one of these exponentials is much smaller than the other, but not very much smaller (that would be a waste of precision). In other words, we should aim for $|g - h| < 8$ or so, where 8 is the number of guard bits we would like to maintain. (Generally it is a good idea to have at least 8 guard bits; somewhat more guard bits do not slow down the calculation very much, but 200 guard bits would be surely an overkill.) Then the number that is much more precise than necessary can be truncated.

For example, if we find that $g = 250$ and $h = 150$, then we can safely truncate x to 160 bits or so; if, in addition, we need only 130 bits of final precision, then we could truncate both x and y to about 140 bits.

Note that when we need to subtract two almost equal numbers, there will be a necessary loss of precision, and it may be impossible to decide on the target precision before performing the subtraction. Therefore the subtraction will have to be performed using all available digits.

The floating zero

There is a difference between an integer zero and a floating-point zero. An integer zero is exact, so the result of `0*1.1` is exactly zero (also an integer). However, `x:=1.1-1.1` is a floating-point zero (a "floating zero" for short) of which we can only be sure about the first digit after the decimal point, i.e. `x=0.0`. The number `x` might represent `0.01` or `-0.02` for all we know.

It is impossible to track the *relative* precision of a floating zero, but it is possible to track the *absolute* precision. Suppose we store the bit count of the absolute precision, just as we store the bit count of the relative precision with nonzero floats. Thus

we represent a floating zero as a pair {0.,n} where n is an integer, and the meaning of this is a number between -2^{-n} and 2^{-n}.

We can now perform some arithmetic operations on the floating zero. Addition and multiplication are handled similarly to the non-zero case, except that we interpret n as the absolute error rather than the relative error. This does not present any problems. For example, the error estimates for addition is the same as if we had a number 1 with relative error 2^{-n} instead of {0.,n}. With multiplication of {x,m} by {0.,n}, the result is again a floating zero {0.,p}, and the new estimate of *absolute* precision is $p = n - \text{BitCount}(x) + 1$.

The division by the floating zero, negative powers, and the logarithm of the floating zero are not representable in our arithmetic because, interpreted as intervals, they would correspond to infinite ranges. The bit count of the floating zero is therefore undefined. However, we can define a positive power of the floating zero (the result is again a floating zero).

The sign of the floating zero is defined as (integer) 0. (Then we can quickly check whether a given number is a zero.)

Comparison of floats

Suppose we need to compare floating-point numbers x and y. In the strict mathematical sense this is an unsolvable problem because we may need in principle arbitrarily many digits of x and y before we can say that they are equal. In other words, "zero-testing is uncomputable". So we need to relax the mathematical rigor somewhat.

Suppose that x=12.0 and y=12.00. Then in fact x might represent a number such as 12.01, while y might represent 11.999. There may be two approaches: first, "12.0" is not equal to "12.00" because x and y *might* represent different numbers. Second, "12.0" is equal to "12.00" because x and y *might* also represent equal numbers. A logical continuation of the first approach is that "12.0" is not even equal to another copy of "12.0" because they *might* represent different numbers, e.g. if we compute x=6.0+6.0 and y=24.0/2.0, the roundoff errors *might* be different.

Here is an illustration in support for the idea that the comparison 12.0=12 should return True. Suppose we are writing an algorithm for computing the power, $x^{\hat{y}}$. This is much faster if y is an integer because we can use the binary squaring algorithm. So we need to detect whether y is an integer. Now suppose we are given x=13.3 and y=12.0. Clearly we should use the integer powering algorithm, even though technically y is a float. (To be sure, we should check that the integer powering algorithm generates enough significant digits.)

However, the opposite approach is also completely possible: no two floating-point numbers should be considered equal, except perhaps when one is a bit-for-bit exact copy of the other and when we haven't yet performed any arithmetic on them.

It seems that no algorithm really needs a test for equality of floats. The two useful comparisons on floats x, y seem to be the following:

1. whether $|x - y| < \epsilon$ where ϵ is a given floating-point number representing the precision,
2. whether x is positive, negative, or zero.

Given these predicates, it seems that any floating-point algorithm can be implemented just as efficiently as with any "reasonable" definition of the floating-point equality.

How to increase of the working precision

Suppose that in a YACAS session we declare Builtin'Precision'Set(5), write x:=0.1, and then increase precision to 10 digits. What is x now? There are several approaches:

1) The number x stays the same but further calculations are done with 10 digits. In terms of the internal binary representation, the number is padded with binary zeros. This means that now e.g. 1+x will not be equal to 1.1 but to something like 1.100000381 (to 10 digits). And actually x itself should evaluate to 0.1000003815 now. This was 0.1 to 5 digits but it looks a little different if we print it to 10 digits. (A "binary-padded decimal".)

This problem may look horrible at first sight – "how come I can't write 0.1 any more??" – but this seems so because we are used to calculations in decimals with a fixed precision, and the operation such as "increase precision by 10 digits" is largely unfamiliar to us except in decimals. This seems to be mostly a cosmetic problem. In a real calculation, we shouldn't be writing "0.1" when we need an exact number 1/10. When we request to increase precision in the middle of a calculation, this mistake surfaces and gives unexpected results.

2) When precision is increased, the number x takes its decimal representation, pads it with zeros, and converts back to the internal representation, just so that the appearance of "1.100000381" does not jar our eyes. (Note that the number x does not become "more precise" if we pad it with decimal zeros instead of binary zeros, unless we made a mistake and wrote "0.1" instead an exact fraction 1/10.)

With this approach, each number x that doesn't currently have enough digits must change in a complicated way. This will mean a performance hit in all calculations that require dynamically changing precision (Newton's method and some other fast algorithms require this). In these calculations, the roundoff error introduced by "1.100000381" is automatically compensated and the algorithm will work equally well no matter how we extend x to more digits; but it's a lot slower to go through the decimal representation every time.

3) The approach currently being implemented in YACAS is a compromise between the above two. We distinguish number objects that were given by the user as decimal strings (and not as results of calculations), for instance x:=1.2, from number objects that are results of calculations, for instance y:=1.2*1.4. Objects of the first kind are interpreted as exact rational numbers given by a decimal fraction, while objects of the second kind are interpreted as inexact floating-point numbers known to a limited precision. Suppose x and y are first assigned as indicated, with the precision of 5 digits each, then the precision is increased to 10 digits and x and y are used in some calculation. At this point x will be converted from the string representation "1.2" to 10 decimal digits, effectively making 1.2 a shorthand for 1.200000000. But the value of y will be binary-padded in some efficient way and may be different from 1.680000000.

In this way, efficiency is not lost (there are no repeated conversions from binary to decimal and back), and yet the cosmetic problem of binary-padded decimals does not appear. An explicitly given decimal string such as "1.2" is interpreted as a shorthand for 1.2000... with as many zeroes as needed for any currently selected precision. But numbers that are results of arithmetic operations are not converted back to a decimal representation for zero-padding. Here are some example calculations:

```
In> Builtin'Precision'Set(5)
Out> True
```

```
In> x:=1.2
Out> 1.2
In> y:=N(1/3)
Out> 0.33333
```

The number `y` is a result of a calculation and has a limited precision. Now we shall increase the precision:

```
In> Builtin'Precision'Set(20)
Out> True
In> y
Out> 0.33333
```

The number `y` is printed with 5 digits, because it knows that it has only 5 correct digits.

```
In> y+0
Out> 0.33333333325572311878
```

In a calculation, `y` was binary-padded, so the last digits are incorrect.

```
In> x+0
Out> 1.2
```

However, `x` has not been padded and remains an "exact" `1.2`.

```
In> z:=y+0
Out> 0.33333333325572311878
In> Builtin'Precision'Set(40)
Out> True
In> z
Out> 0.33333333325572311878
```

Now we can see how the number `z` is padded again:

```
In> z+0
Out> 0.33333333325572311878204345703125
```

The meaning of the `Builtin'Precision'Set()` call

The user calls `Builtin'Precision'Set()` to specify the "wanted number of digits." We could use different interpretations of the user's wish: The first interpretation is that `Builtin'Precision'Set(10)` means "I want all answers of all calculations to contain 10 correct digits". The second interpretation is "I want all calculations with floating-point numbers done using at least 10 digits".

Suppose we have floating-point numbers `x` and `y`, known only to 2 and 3 significant digits respectively. For example, `x=1.6` and `y=2.00`. These `x` and `y` are results of previous calculations and we do not have any more digits than this. If we now say

```
Builtin'Precision'Set(10);
x*y;
```

then clearly the system cannot satisfy the first interpretation because there aren't enough digits of x and y to find 10 digits of xy. But we can satisfy the second interpretation, even if we print "3.2028214767" instead of the expected `3.2`. The garbage after the third digit is unavoidable and harmless unless our calculation really depends on having 10 correct digits of xy. The garbage digits can be suppressed when the number is printed, so that the user will never see them. But if our calculation depends on the way we choose the extra digits, then we are using a bad algorithm.

The first interpretation of `Builtin'Precision'Set()` is only possible to satisfy if we are given a self-contained calculation with integer numbers. For example,

```
N(Sin(Sqrt(3/2)-10^(20)), 50)
```

This can be computed to 50 digits with some effort, but only if we are smart enough to use 70 digits in the calculation of the argument of `Sin()`. (This level of smartness is currently not implemented in the `N` function.) The result of this calculation will have 50 digits and not a digit more; we cannot put the result inside another expression and expect full precision in all cases. This seems to be a separate task, "compute something with `n` digits no matter what", and not a general routine to be followed at all times.

So it seems that the second interpretation of `Builtin'Precision'Set(n)`, namely: "please use `n` digits in all calculations now", is more sensible as a general-purpose prescription.

But this interpretation does not mean that all numbers will be *printed* with `n` digits. Let's look at a particular case (for simplicity we are talking about decimal digits but in the implementation they will be binary digits). Suppose we have `x` precise to 10 digits and `y` precise to 20 digits, and the user says `Builtin'Precision'Set(50)` and `z:=1.4+x*y`. What happens now in this calculation? (Assume that `x` and `y` are small numbers of order 1; the other cases are similar.)

First, the number "1.4" is now interpreted as being precise to 50 digits, i.e. "1.4000000...0" but not more than 50 digits.

Then we compute `x*y` using their internal representations. The result is good only to 10 digits, and it knows this. We do not compute 50 digits of the product `x*y`, it would be pointless and a waste of time.

Then we add `x*y` to 1.4000...0. The sum, however, will be precise only to 10 digits. We can do one of the two things now: (a) we could pad `x*y` with 40 more zero digits and obtain a 50-digit result. However, this result will only be correct to 10 digits. (b) we could truncate 1.4 to 10 digits (1.400000000) and obtain the sum to 10 digits. In both cases the result will "know" that it only has 10 correct digits.

It seems that the option (b) is better because we do not waste time with extra digits.

The result is a number that is precise to 10 digits. However, the user wants to see this result with 50 digits. Even if we chose the option (a), we would have had some bogus digits, in effect, 40 digits of somewhat random round-off error. Should we print 10 correct digits and 40 bogus digits? It seems better to print only 10 correct digits in this case.

If we choose this route, then the only effect of `Builtin'Precision'Set(50)` will be to interpret a literal constant `1.4` as a 50-digit number. All other numbers already know their real precision and will not invent any bogus digits.

In some calculations, however, we do want to explicitly extend the precision of a number to some more digits. For example, in Newton's method we are given a first approximation x_0 to a root of $f(x) = 0$ and we want to have more digits of that root. Then we need to pad x_0 with some more digits and re-evaluate $f(x_0)$ to more digits (this is needed to get a better approximation to the root). This padding operation seems rather special and directed at a particular number, not at all numbers at once. For example, if $f(x)$ itself contains some floating-point numbers, then we should be unable to evaluate it with higher precision than allowed by the precision of these numbers. So it seems that we need access to these two low-level operations: the padding and the query of current precision. The proposed interface is `GetExactBits(x)` and `SetExactBits(x,n)`. These operations are directed at a particular number object `x`.

Summary of arbitrary-precision semantics

1. All integers are always exact; all floats carry an error es-

timate, which is stored as the number of correct bits of mantissa they have. Symbolically, each float is a pair {x,n} where x is a floating-point value and n is a (platform) integer value. If $x \neq 0$, then the relative error of x is estimated as 2^{-n}. A number {x,n} with $x \neq 0$ stands for an interval between $x\left(1-2^{-n}\right)$ and $x\left(1+2^{-n}\right)$. This integer n is returned by GetExactBits(x) and can be modified by SetExactBits(x,n) (see below). Error estimates are not guaranteed to be correct in all cases, but they should give sensible *lower* bounds on the error. For example, if {x,n}={123.456,3}, the error estimate says that x is known to at most 3 bits and therefore the result of $\frac{1}{x-123}$ is completely undefined becase x cannot be distinguished from 0. The purpose of the precision tracking mechanism is to catch catastrophic losses of numerical precision in cases like these, not to provide a precise round-off error estimates. In most cases it is better to let the program continue even with loss of precision than have it aborted due to a false round-off alarm.

2. When printing a float, we print only as many digits as needed to represent the float value to its current precision. When reading a float, we reserve enough precision to preserve all given digits.

3. The number 0 is either an integer 0 or a floating-point 0. (a "floating zero" for short). For a floating zero, the "number of exact bits" means the absolute error, not the relative error. It means that the symbolic pair {0.,n} represents all number x in the interval $-2^{-n} \leq x \leq 2^{-n}$. A floating zero can be obtained either by subtracting almost equal numbers or by squaring a very imprecise number. In both cases the possible result can be close to zero and the precision of the initial numbers is insufficient to distinguish it from zero.

4. An integer and a float are equal only if the float contains this integer value within its precision interval. Two floats are equal only if their values differ by less than the largest of their error estimates (i.e. if their precision intervals intersect). In particular, it means that an integer zero is always equal to any floating zero, and that any two floating zeros are equal. It follows that if x=y, then for any floating zeros x+0.=y+0. and x-y=0. as well. (So this arithmetic is not obviously inconsistent.)

5. The Yacas function IsInteger(x) returns True if x has integer *type*; IsIntValue(x) returns True if x has either integer type or floating type but an integer value within its precision. For example,

```
IsInteger(0)  =True
IsIntValue(1.)=True
IsInteger(1.) =False
```

6. The Yacas function Builtin'Precision'Set(n) sets a global parameter that controls the precision of *newly created floats*. It does not modify previously created floating-point numbers and has no effect on copying floats or on any integer calculations. New float objects can be created in three ways (aside from simple copying from other floats): from literal strings, from integers, and from calculations with other floats. For example,

```
x:=1.33;
y:=x/3;
```

Here x is created from a literal string "1.33", a temporary float is created from an integer 3, and y is created as a result of division of two floats. Converting an integer to a float is similar to converting from a literal string representing the integer. A new number object created from a literal string must have at least as many bits of mantissa as is required to represent the value given by the string. The string might be very long but we want to retain all information from a string, so we may have to make the number much more precise than currently declared with Builtin'Precision'Set. [2] But if the necessary number of digits to represent the string is less than n, then the new number object will have the number of bits that corresponds to n decimal digits. Thus, in creating objects from strings or from integers, Builtin'Precision'Set sets the *minimum* precision of the resulting floating-point number. On the other hand, a new number object created from a calculation will already have an error estimate and will "know" its real precision. But a directive Builtin'Precision'Set(n) indicates that we are only interested in n digits of a result. Therefore, a calculation should not generate *more* digits than n, even if its operands have more digits. Thus, in creating objects from operations, Builtin'Precision'Set sets the *maximum* precision of the resulting floating-point number.

7. SetExactBits(x,n) will make the number x think that it has n exact bits. If x had more exact bits before, then it may be rounded. If x had fewer exact bits before, then it may be padded. (The way the padding is done is up to the internal representation, but the padding operation must be efficient and should not change the value of the number beyond its original precision.)

8. All arithmetic operations and all kernel-supported numerical function calls are performed with precision estimates, so that all results know how precise they really are. Then in most cases it will be unnecessary to call SetExactBits or GetExactBits explicitly. This will be needed only in certain numerical applications that need to control the working precision for efficiency.

Formal definitions of precision tracking

Here we shall consider arithmetic operations on floats x and y, represented as pairs {x,m} and {y,n}. The result of the operation is z, represented as a pair {z,p}. Here x, y, z are floats and m, n, p are integers.

We give formulae for p in terms of x, y, m, and n. Sometimes the bit count of a number x is needed; it is denoted $B(x)$ for brevity.

Formal definitions

A pair {x,m} where x is a floating-point value and m is an integer value (the "number of correct bits") denotes a real number between $x\left(1-2^{-m}\right)$ and $x\left(1+2^{-m}\right)$ when $x \neq 0$, and a real number between -2^{-m} and 2^{-m} when $x = 0$ (a "floating zero").

The bit count $B(x)$ is an integer function of x defined for real $x \neq 0$ by

$$B(x) \equiv 1 + \left\lfloor \frac{\ln|x|}{\ln 2} \right\rfloor .$$

This function also satisfies

$$2^{B(x)-1} \leq |x| < 2^{B(x)}.$$

For example, $B\left(\frac{1}{4}\right) = -1$, $B(1) = B\left(\frac{3}{2}\right) = 1$, $B(4) = 3$. The bit count of zero is arbitrarily set to 1. For integer x, the

[2] Note that the argument n of Builtin'Precision'Set(n) means *decimal* digits, not bits. This is more convenient for YACAS sessions.

value $B(x)$ is the number of bits needed to write the binary representation of x.

The bit count function can be usually computed in *constant* time because the usual representation of long numbers is by arrays of platform integers and a binary exponent. The length of the array of digits is usually available at no computational cost.

The *absolute* error Δ_x of {x,n} is of order $|x| \cdot 2^{-n}$. Given the bit count of x, this can be estimated from as

$$2^{B(x)-n-1} \leq \Delta_x < 2^{B(x)-n}.$$

So the bit count of Δ_x is $B(x) - n$.

`Floor()`

The function `Floor({x,m})` gives an integer result if there are enough digits to determine it exactly, and otherwise returns the unchanged floating-point number. The condition for `Floor({x,m})` to give an exact result is

$$m \geq B(x).$$

`BecomeFloat()`

The function `BecomeFloat(n)` will convert an integer to a float with at least n digits of precision. If `x` is the original integer value, then the result is {x,p} where $p = \max(n, B(x))$.

Underflow check

It is possible to have a number {x,n} with $x \neq 0$ such that {0.,m}={x,n} for some m. This would mean that the floating zero {0.,m} is not precise enough to be distinguished from {x,n}, i.e.

$$|x| < 2^{-m}.$$

This situation is normal. But it would be meaningless to have a number {x,n} with $x \neq 0$ and a precision interval that contains 0. Such {x,n} will in effect be equal to *any* zero {0.,m}, because we do not know enough digits of `x` to distinguish {x,n} from zero.

From the definition of {x,n} with $x \neq 0$ it follows that 0 can be within the precision interval only if $n \leq -1$. Therefore, we should transform any number {x,n} such that $x \neq 0$ and $n \leq -1$ into a floating zero {0.,p} where

$$p = n - B(x).$$

(Now it is not necessarily true that $p \geq 0$.) This check should be performed at any point where a new precision estimate `n` is obtained for a number `x` and where a cancellation may occur (e.g. after a subtraction). Then we may assume that any given float is already reduced to zero if possible.

`Equals()`

We need to compare {x,m} and {y,n}.

First, we can quickly check that the values x and y have the same nonzero signs and the same bit counts, $B(x) = B(y)$. If $x > 0$ and $y < 0$ or vice versa, or if $B(x) = B(y)$, then the two numbers are definitely unequal. We can also check whether both $x = y = 0$; if this is the case, then we know that {x,m}={y,n} because any two zeros are equal.

However, a floating zero can be sometimes equal to a nonzero number. So we should now exclude this possibility: {0.,m}={y,n} if and only if $|y| < 2^{-m}$. This condition is equivalent to

$$B(y) < -m.$$

If these checks do not provide the answer, the only possibility left is when $x \neq 0$ and $y \neq 0$ and $B(x) = B(y)$.

Now we can consider two cases: (1) both x and y are floats, (2) one is a float and the other is an integer.

In the first case, {x,m}={y,n} if and only if the following condition holds:

$$|x - y| < \max\left(2^{-m}|x|, 2^{-n}|y|\right).$$

This is a somewhat complicated condition but its evaluation does not require any long multiplications, only long additions, bit shifts and comparisons.

It is now necessary to compute $x - y$ (one long addition); this computation needs to be done with $\min(m, n)$ bits of precision.

After computing $x - y$, we can avoid the full evaluation of the complicated condition by first checking some easier conditions on $x - y$. If $x - y = 0$ as floating-point numbers ("exact cancellation"), then certainly {x,m}={y,n}. Otherwise we can assume that $x - y \neq 0$ and check:

- A sufficient (but not a necessary) condition: if $B(x - y) \leq B(x) - \min(m, n) - 1$ then {x,m}={y,n}.
- A necessary (but not a sufficient) condition is: if $B(x - y) > B(x) - \min(m, n) + 1$ then {x,m}!={y,n}.

If neither of these conditions can give us the answer, we have to evaluate the full condition by computing $|x| \cdot 2^{-m}$ and $|x| \cdot 2^{-m}$ and comparing with $|x - y|$.

In the second case, one of the numbers is an integer `x` and the other is a float {y,n}. Then x={y,n} if and only if

$$|x - y| < 2^{-n}|y|.$$

For the computation of $x - y$, we need to convert `x` into a float with precision of n digits, i.e. replace the integer `x` by a float {x,n}. Then we may use the procedure for the first case (two floats) instead of implementing a separate comparison procedure for integers.

`LessThan()`

If {x,m}={y,n} according to the comparison function `Equals()`, then the predicate `LessThan` is false. Otherwise it is true if and only if $x < y$ as floats.

`IsIntValue()`

To check whether {x,n} has an integer value within its precision, we first need to check that {x,n} has enough digits to compute $\lfloor x \rfloor$=`Floor(x)` accurately. If not (if $n < B(x)$), then we conclude that x has an integer value. Otherwise we compute $y \equiv x - \lfloor x \rfloor$ as a float value (without precision control) to `n` bits. If y is exactly zero as a float value, then x has an integer value. Otherwise {x,n} has an integer value if and only if $B(y) < -n$.

This procedure is basically the same as comparing {x,n} with `Floor(x)`.

`Sign()`

The sign of {x,n} is defined as the sign of the float value x. (The number {x,n} should have been reduced to a floating zero if necessary.)

Addition and subtraction (`Add`, `Negate`)

We need to add {x,m} and {y,n} to get the result {z,p}. Subtraction is the same as addition, except we negate the second number. When we negate a number, its precision never changes.

First consider the case when $x + y \neq 0$.

If x is zero, i.e. {0.,m} (but $x+y \neq 0$), then the situation with precision is the same as if x were {1.,m}, because then the relative precision is equal to the absolute precision. In that case we take the bit count of x as $B(0) = 1$ and proceed by the same route.

First, we should decide whether it is necessary to add the given numbers. It may be unnecessary if e.g. $x + y \approx x$ within precision of x (we may say that a "total underflow" occurred during addition). To check for this, we need to estimate the absolute errors of x and y:

$$2^{B(x)-m-1} \leq \Delta_x < 2^{B(x)-m},$$

$$2^{B(y)-n-1} \leq \Delta_y < 2^{B(y)-n}.$$

Addition is not necessary if $|x| \leq \Delta_y$ or if $|y| \leq \Delta_x$. Since we should rather perform an addition than wrongly dismiss it as unnecessary, we should use a sufficient condition here: if

$$B(x) \leq B(y) - n - 1$$

then we can neglect x and set $z = y$, $p = n - \mathrm{Dist}(B(x), B(y) - n - 1)$. (We subtract one bit from the precision of y in case the magnitude of x is close to the absolute error of y.) Also, if

$$B(y) \leq B(x) - m - 1$$

then we can neglect y and set $z = x$, $p = m - \mathrm{Dist}(B(y), B(x) - m - 1)$.

Suppose none of these checks were successful. Now, the float value $z = x + y$ needs to be calculated. To find it, we need the target precision of only

$$1 + \max(B(x), B(y)) - \max(B(x) - m, B(y) - n)$$

bits. (An easier upper bound on this is $1 + \max(m, n)$ but this is wasteful when x and y have very different precisions.)

Then we compute $B(z)$ and determine the precision p as

$$p = \min(m - B(x), n - B(y)) + B(z)$$

$$-1 - \mathrm{Dist}(m - B(x), n - B(y)),$$

where the auxiliary function $\mathrm{Dist}(a, b)$ is defined as 0 when $|a - b| > 2$ and 1 otherwise. [3]

In the case when x and y have the same sign, we have a potentially better estimate $p = \min(m, n)$. We should take this value if it is larger than the value obtained from the above formula.

Also, the above formula is underestimating the precision of the result by 1 bit if the result *and* the absolute error are dominated by one of the summands. In this case the absolute error should be unchanged save for the Dist term, i.e. the above formula needs to be incremented by 1. The condition for this is $B(x) > B(y)$ and $B(x) - m > B(y) - n$, or the same for y instead of x.

The result is now {z,p}.

Note that the obtained value of p may be negative (total underflow) even though we have first checked for underflow. In that case, we need to transform {z,p} into a floating zero, as usual.

Now consider the case when $z \equiv x + y = 0$.

This is only possible when $B(x) = B(y)$. Then the result is {0.,p} where p is found as

$$p = 1 + \min(m, n) - B(x) - \mathrm{Dist}(m, n).$$

[3] The definition of $\mathrm{Dist}(a, b)$ is necessarily approximate; if we replace 2 by a larger number, we shall be overestimating the error in more cases.

Note that this is the same formula as in the general case, if we define $B(z) = B(0) \equiv 1$. Therefore with this definition of the bit count one can use one formula for the precision of addition in all cases.

If the addition needs to be performed with a given maximum precision P, and it turns out that $p > P$, then we may truncate the final result to P digits and set its precision to P instead. (It is advisable to leave a few bits untruncated as guard bits.) However, the first operation z:=x+y must be performed with the precision specified above, or else we run the danger of losing significant digits of z.

Adding integers to floats

If an integer x needs to be added to a float {y,n}, then we should formally use the same procedure as if x had infinitely many precise bits. In practice we can take some shortcuts.

It is enough to convert the integer to a float {x,m} with a certain finite precision m and then follow the general procedure for adding floats. The precision m must be large enough so that the absolute error of {x,m} is smaller than the absolute error of {y,n}: $B(x) - m \leq B(y) - n - 1$, hence

$$m \geq 1 + n + B(x) - B(y).$$

In practice we may allow for a few guard bits over the minimum m given by this formula.

Sometimes the formula gives a negative value for the minimum m; this means underflow while adding the integer (e.g. adding 1 to 1.11e150). In this case we do not need to perform any addition at all.

Multiplication

We need to multiply {x,m} and {y,n} to get the result {z,p}.

First consider the case when $x \neq 0$ and $y \neq 0$. The resulting value is $z = xy$ and the precision is

$$p = \min(m, n) - \mathrm{Dist}(m, n).$$

If one of the numbers is an integer x, and the other is a float {y,n}, it is enough to convert x to a float with somewhat more than n bits, e.g. {x,n+3}, so that the Dist function does not decrement the precision of the result.

Now consider the case when {x,m}={0,m} but $y \neq 0$. The result $z = 0$ and the resulting precision is

$$p = m - B(y) + 1.$$

Finally, consider the case when {x,m}={0,m} and {y,n}={0,n}. The result $z = 0$ and the resulting precision is

$$p = m + n.$$

The last two formulae are the same if we defined the bit count of {0.,m} as $1 - m$. This differs from the "standard" definition of $B(0) = 1$. (The "standard" definition is convenient for the handling of addition.) With this non-standard definition, we may use the unified formula

$$p = 2 - B(x) - B(y)$$

for the case when one of x, y is a floating zero.

If the multiplication needs to be performed to a given target precision P which is larger than the estimate p, then we can save time by truncating both operands to P digits before performing the multiplication. (It is advisable to leave a few bits untruncated as guard bits.)

Division

Division is handled essentially in the same way as multiplication. The relative precision of `x/y` is the same as the relative precision of `x*y` as long as both $x \neq 0$ and $y \neq 0$.

When $x = 0$ and $y \neq 0$, the result of division `{0.,m}/{y,n}` is a floating zero `{0.,p}` where $p = m + B(y) - 1$. When x is an integer zero, the result is also an integer zero.

Division by an integer zero or by a floating zero is not permitted. The code should signal a zero division error.

`ShiftLeft()`, `ShiftRight()`

These operations efficiently multiply a number by a positive or negative power of 2. Since 2 is an exact integer, the precision handling is similar to that of multiplication of floats by integers.

If the number `{x,n}` is nonzero, then only x changes by shifting but n does not change; if `{x,n}` is a floating zero, then x does not change and n is decremented (`ShiftLeft`) or incremented (`ShiftRight`) by the shift amount:

```
{x, n} << s = {x<<s, n};
{0.,n} << s = {0., n-s};
{x, n} >> s = {x>>s, n};
{0.,n} >> s = {0., n+s};
```

7.5 Implementation notes

Large exponents

The `BigNumber` API does not support large exponents for floating-point numbers. A floating-point number x is equivalent to two integers M, N such that $x = M \cdot 2^N$. Here M is the (denormalized) mantissa and N is the (binary) exponent. The integer M must be a "big integer" that may represent thousands of significant bits. But the exponent N is a platform signed integer (C++ type `long`) which is at least $2\hat{3}1$, allowing a vastly larger range than platform floating-point types. One would expect that this range of exponents is enough for most real-world applications. In the future this limitation may be relaxed if one uses a 64-bit platform. (A 64-bit platform seems to be a better choice for heavy-duty multiple-precision computations than a 32-bit platform.) However, code should not depend on having 64-bit exponent range.

We could have implemented the exponent N as a big integer but this would be inefficient most of the time, slowing down the calculations. Arithmetic with floating-point numbers requires only very simple operations on their exponents (basically, addition and comparisons). These operations would be dominated by the overhead of dealing with big integers, compared with platform integers.

A known issue with limited exponents is the floating-point overflow and exponent underflow. (This is not the same underflow as with adding 1 to a very small number.) When the exponent becomes too large to be represented by a platform signed integer type, the code must signal an overflow error (e.g. if the exponent is above 2^{31}) or an underflow error (e.g. if the exponent is negative and below -2^{31}).

Library versions of mathematical functions

It is usually the case that a multiple-precision library implements some basic mathematical functions such as the square root. A library implementation may be already available and more efficient than an implementation using the API of the wrapper class `BigNumber`. In this case it is desirable to wrap the library implementation of the mathematical function, rather than use a suboptimal implementation. This could be done in two ways.

First, we recognize that we shall only have one particular numerical library linked with Yacas, and we do not have to compile our implementation of the square root if this library already contains a good implementation. We can use conditional compilation directives (`#ifdef`) to exclude our square root code and to insert a library wrapper instead. This scheme could be automated, so that appropriate `#define`s are automatically created for all functions that are already available in the given multiple-precision library, and the corresponding Yacas kernel code that uses the `BigNumber` API is automatically replaced by library wrappers.

Second, we might compile the library wrapper as a plugin, replacing the script-level square root function with a plugin-supplied function. This solution is easier in some ways because it doesn't require any changes to the Yacas core, only to the script library. However, the library wrapper will only be available to the Yacas scripts and not to the Yacas core functions. The basic assumption of the plugin architecture is that plugins can provide new external objects and functions to the scripts, but plugins cannot modify anything in the kernel. So plugins can replace a function defined in the scripts, but cannot replace a kernel function. Suppose that some other function, such as a computation of the elliptic integral which heavily uses the square root, were implemented in the core using the `BigNumber` API. Then it will not be able to use the square root function supplied by the plugin because it has been already compiled into the Yacas kernel.

Third, we might put all functions that use the basic API (`MathSqrt`, `MathSin` etc.) into the script library and not into the Yacas kernel. When Yacas is compiled with a particular numerical library, the functions available from the library will also be compiled as the kernel versions of `MathSqrt`, `MathPower` and so on (using conditional compilation or configured at build time). Since Yacas tries to call the kernel functions before the script library functions, the available kernel versions of `MathSqrt` etc. will supersede the script versions, but other functions such as `BesselJ` will be used from the script library. The only drawback of this scheme is that a plugin will not be able to use the faster versions of the functions, unless the plugin was compiled specifically with the requirement of the particular numerical library.

So it appears that either the first or the third solution is viable.

Converting from bits to digits and back

One task frequently needed by the arithmetic library is to convert a precision in (decimal) digits to binary bits and back. (We consider the decimal base to be specific; the same considerations apply to conversions between any other bases.) The kernel implements auxiliary routines `bits_to_digits` and `digits_to_bits` for this purpose.

Suppose that the mantissa of a floating-point number is known to d decimal digits. It means that the relative error is no more than $0.5 \cdot 10^{-d}$. The mantissa is represented internally as a binary number. The number b of precise bits of mantissa should be determined from the equation $10^{-d} = 2^{-b}$, which gives $b = d\frac{\ln 10}{\ln 2}$.

One potential problem with the conversions is that of incorrect rounding. It is impossible to represent d decimal digits by some exact number b of binary bits. Therefore the actual value of b must be a little different from the theoretical one. Then suppose we perform the inverse operation on b to obtain

the corresponding number of precise decimal digits; there is a danger that we shall obtain a number d' that is different from d.

To avoid this danger, the following trick is used. The binary base 2 is the least of all possible bases, so successive powers of 2 are more frequent than successive powers of 10 or of any other base. Therefore for any power of 10 there will be a unique power of 2 that is the first one above it.

The recipe to obtain this power of 2 is simple: one should round $d\frac{\ln 10}{\ln 2}$ upwards using the `Ceil` function, but $b\frac{\ln 2}{\ln 10}$ should be rounded downwards using the `Floor` function.

This procedure will make sure that the number of bits b is high enough to represent all information in the d decimal digits; at the same time, the number d will be correctly restored from b. So when a user requests d decimal digits of precision, Yacas may simply compute the corresponding value of b and store it. The precision of b digits is enough to hold the required information, and the precision d can be easily computed given b.

The internal storage of BigNumber objects

An object of type `BigNumber` represents a number (and contains all information relevant to the number), and offers an interface to operations on it, dispatching the operations to an underlying arbitrary precision arithmetic library.

Higher up, Yacas only knows about objects derived from `LispObject`. Specifically, there are objects of class `LispAtom` which represent an atom.

Symbolic and string atoms are uniquely represented by the result returned by the `String()` method. For number atoms, there is a separate class, `LispNumber`. Objects of class `LispNumber` also have a `String()` method in case a string representation of a number is needed, but the main uniquely identifying piece of information is the object of class `BigNumber` stored inside a `LispNumber` object. This object is accessed using the `Number()` method of class `LispNumber`.

The life cycle of a `LispNumber` is as follows:

1. A `LispNumber` can be born when the parser reads in a numeric atom. In such a case an object of type `LispNumber` is created instead of the `LispAtom`. The `LispNumber` constructor stores the string representation but does not yet create an object of type `BigNumber` from the string representation. The `BigNumber` object is later automatically created from the string representation. This is done by the `Number(precision)` method the first time it is requested. String conversion is deferred to save time when reading scripts.

2. Suppose the `Number` method is called; then a `BigNumber` object will be created from the string representation, using the current precision. This is where the string conversion takes place. If later the precision is increased, the string conversion will be performed again. This allows to hold a number such as `1.23` and interpret it effectively as an exact rational `123/100`.

3. For an arithmetic calculation, say addition, two arguments are passed in, and their internal objects should be of class `LispNumber`, so that the function doing the addition can get at the `BigNumber` objects by calling the `Number()` method. [This method will not attempt to create a number from the string representation if a numerical representation is already available.] The function that performs the arithmetic then creates a new `BigNumber`, stores the result of the calculation in it, and creates a new `LispNumber` by constructing it with the new `BigNumber`. The result is a `LispNumber` with a `BigNumber` inside it but without any string representation. Other operations can proceed to use this `BigNumber` stored inside the `LispNumber`. This is in effect the second way a `LispNumber` can be born. Since the string representation is not available in this case, no string conversions are performed any more. If precision is increased, there is no way to obtain any more digits of the number.

4. Right at the end, when a result needs to be printed to screen, the printer will call the `String()` method of the `LispNumber` object to get a string representation. The obtained (decimal) string representation of the number is also stored in the `LispNumber`, to avoid repeated conversions.

In order to fully support the `LispNumber` object, the function in the kernel that determines if two objects are the same needs to know about `LispNumber`. This is required to get valid behaviour. Pattern matching for instance uses comparisons of this type, so comparisons are performed often and need to be efficient.

The other functions working on numbers can, in principle, call the `String()` method, but that induces conversions from `BigNumber` to string, which are relatively expensive operations. For efficiency reasons, the functions dealing with numeric input should call the `Number()` method, operate on the `BigNumber` returned, and return a `LispNumber` constructed with a `BigNumber`. A function can call `String()` and return a `LispNumber` constructed with a string representation, but it will be less efficient.

Precision tracking inside LispNumber

There are various subtle details when dealing with precision. A number gets constructed with a certain precision, but a higher precision might be needed later on. That is the reason there is the `aPrecision` argument to the `Number()` method.

When a `BigNumber` is constructed from a decimal string, one has to specify a desired precision (in decimal digits). Internally, `BigNumber` objects store numbers in binary and will allocate enough bits to cover the desired precision. However, if the given string has more digits than the given precision, the `BigNumber` object will not truncate it but will allocate more bits so that the information given in the decimal string is not lost. If later the string representation of the `BigNumber` object is requested, the produced string will match the string from which the `BigNumber` object was created.

Internally, the `BigNumber` object knows how many precise bits it has. The number of precise digits might be greater than the currently requested precision. But a truncation of precision will only occur when performing arithmetic operations. This behavior is desired, for example:

```
In> Builtin'Precision'Set(6)
Out> True;
In> x:=1.23456789
Out> 1.23456789;
In> x+1.111
Out> 2.345567;
In> x
Out> 1.23456789;
```

In this example, we would like to keep all information we have about `x` and not truncate it to 6 digits. But when we add a number to `x`, the result is only precise to 6 digits.

This behavior is implemented by storing the string representation `"1.23456789"` in the `LispNumber` object `x`. When an arithmetic calculation such as `x+1.111` is requested, the `Number` method is called on `x`. This method, when called for the first time, converts the string representation into a `BigNumber` object.

That `BigNumber` object will have 28 bits to cover the 9 significant digits of the number, not the 19 bits normally required for 6 decimal digits of precision. But the result of an arithmetic calculation is not computed with more than 6 decimal digits. Later when `x` needs to be printed, the full string representation is available so it is printed.

If now we increase precision to 20 digits, the object `x` will be interpreted as 1.23456789 with 12 zeros at the end.

```
In> Builtin'Precision'Set(20)
Out> True;
In> x+0.000000000001
Out> 1.234567890001;
```

This behavior is more intuitive to people who are used to decimal fractions.

Chapter 8

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

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

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

Applicability and Definitions

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

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

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

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

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

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

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

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

Verbatim Copying

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

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

Copying in Quantity

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

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

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

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

Modifications

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

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.
9. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
11. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
13. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
14. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

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

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

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

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

Combining Documents

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

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

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

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

Collections of Documents

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

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

Aggregation With Independent Works

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

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

Translation

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

Termination

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

Future Revisions of This License

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

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

ADDENDUM: How to use this License for your documents

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

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

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

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

Index

`autoconf`, 29
`automake`, 28
`autotools`, 28

build system, 18, 28
  autoconf, 29
  automake, 28
  autotools, 28
  configure, 29
  documentation, 30
  executable, 30
  makemake, 28
  targets for make, 30
  test suite, 31
  Yacas scripts, 30

configuration, 29
CVS, 18
  check out, 18
  commit, 19
  update, 19

documentation, 19
  TEX problems, 25
  building, 30
  comments, 22
  embedded code, 25
  embedded in code, 26
  environments, 21
  equations, 21
  fonts, 21
  footnotes, 22
  indexing, 23
  insert another file, 22
  internal hyperlinks, 21
  lines too wide, 25
  literate programming, 25, 26
  markup debugging, 24
  markup overview, 20
  organization, 20
  outside of Yacas source tree, 27
  reference manual markup, 22
  summary of labels, 23
  summary of syntax, 24
  Web hyperlinks, 21

make
  targets, 30
`makemake`, 28
multiple-precision facility
  requirements, 35

scripts
  installation, 30

testing Yacas, 31

Wester's benchmark, 31