Programming in Yacas, function reference

by the YACAS team [1]

YACAS version: 1.3.6
generated on April 27, 2016

This is the second part of the Yacas function reference. This reference explains functions that are used in the Yacas source code that is underneath what the user sees. The documentation in this section is thus mostly useful to people who are maintaining Yacas.

Contents

Chapter 1

Introduction

This document aims to be a reference for functions that are useful when programming in YACAS, but which are not necessarily useful when using YACAS. There is another document that describes the functions that are useful from a users point of view.

Chapter 2

Programming

This chapter describes functions useful for writing Yacas scripts.

/* — Start of comment

*/ — end of comment

// — Beginning of one-line comment

(YACAS internal)

Calling format:

```
/* comment */
// comment
```

Description:

Introduce a comment block in a source file, similar to C++ comments. `//` makes everything until the end of the line a comment, while `/*` and `*/` may delimit a multi-line comment.

Examples:

```
a+b; // get result
a + /* add them */ b;
```

Prog — block of statements

[— beginning of block of statements

] — end of block of statements

(YACAS internal)

Calling format:

```
Prog(statement1, statement2, ...)
[ statement1; statement2; ... ]
```

Parameters:

`statement1`, `statement2` – expressions

Description:

The `Prog` and the `[ ... ]` construct have the same effect: they evaluate all arguments in order and return the result of the last evaluated expression.

`Prog(a,b);` is the same as typing `[a;b;];` and is very useful for writing out function bodies. The `[ ... ]` construct is a syntactically nicer version of the `Prog` call; it is converted into `Prog(...)` during the parsing stage.

Bodied — define function syntax (bodied function)

Infix — define function syntax (infix operator)

Postfix — define function syntax (postfix operator)

Prefix — define function syntax (prefix operator)

(YACAS internal)

Calling format:

```
Bodied("op", precedence)
Infix("op")
Infix("op", precedence)
Postfix("op")
Postfix("op", precedence)
Prefix("op")
Prefix("op", precedence)
```

Parameters:

`"op"` – string, the name of a function
`precedence` – nonnegative integer (evaluated)

Description:

Declares a special syntax for the function to be parsed as a bodied, infix, postfix, or prefix operator.

"Bodied" functions have all arguments except the first one inside parentheses and the last argument outside, for example:

```
For(pre, condition, post) statement;
```

Here the function `For` has 4 arguments and the last argument is placed outside the parentheses. The `precedence` of a "bodied" function refers to how tightly the last argument is bound to the parentheses. This makes a difference when the last argument contains other operators. For example, when taking the derivative

```
D(x) Sin(x)+Cos(x)
```

both `Sin` and `Cos` are under the derivative because the bodied function `D` binds less tightly than the infix operator "`+`".

"Infix" functions must have two arguments and are syntactically placed between their arguments. Names of infix functions

can be arbitrary, although for reasons of readability they are usually made of non-alphabetic characters.

"Prefix" functions must have one argument and are syntactically placed before their argument. "Postfix" functions must have one argument and are syntactically placed after their argument.

Function name can be any string but meaningful usage and readability would require it to be either made up entirely of letters or entirely of non-letter characters (such as "+", ":" etc.). Precedence is optional (will be set to 0 by default).

Examples:

```
In> YY x := x+1;
CommandLine(1) : Error parsing expression

In> Prefix("YY", 2)
Out> True;
In> YY x := x+1;
Out> True;
In> YY YY 2*3
Out> 12;
In> Infix("##", 5)
Out> True;
In> a ## b ## c
Out> a##b##c;
```

Note that, due to a current parser limitation, a function atom that is declared prefix cannot be used by itself as an argument.

```
In> YY
CommandLine(1) : Error parsing expression
```

See also: `IsBodied`, `OpPrecedence`

`IsBodied` — check for function syntax

`IsInfix` — check for function syntax

`IsPostfix` — check for function syntax

`IsPrefix` — check for function syntax

(YACAS internal)

Calling format:

```
IsBodied("op")
IsInfix("op")
IsPostfix("op")
IsPrefix("op")
```

Parameters:

`"op"` – string, the name of a function

Description:

Check whether the function with given name `"op"` has been declared as a "bodied", infix, postfix, or prefix operator, and return `True` or `False`.

Examples:

```
In> IsInfix("+");
Out> True;
In> IsBodied("While");
Out> True;
In> IsBodied("Sin");
Out> False;
In> IsPostfix("!");
Out> True;
```

See also: `Bodied`, `OpPrecedence`

`OpPrecedence` — get operator precedence

`OpLeftPrecedence` — get operator precedence

`OpRightPrecedence` — get operator precedence

(YACAS internal)

Calling format:

```
OpPrecedence("op")
OpLeftPrecedence("op")
OpRightPrecedence("op")
```

Parameters:

`"op"` – string, the name of a function

Description:

Returns the precedence of the function named "op" which should have been declared as a bodied function or an infix, postfix, or prefix operator. Generates an error message if the string str does not represent a type of function that can have precedence.

For infix operators, right precedence can differ from left precedence. Bodied functions and prefix operators cannot have left precedence, while postfix operators cannot have right precedence; for these operators, there is only one value of precedence.

Examples:

```
In> OpPrecedence("+")
Out> 6;
In> OpLeftPrecedence("!")
Out> 0;
```

`RightAssociative` — declare associativity

(YACAS internal)

Calling format:

```
RightAssociative("op")
```

Parameters:

"op" – string, the name of a function

Description:

This makes the operator right-associative. For example:

```
RightAssociative("*")
```

would make multiplication right-associative. Take care not to abuse this function, because the reverse, making an infix operator left-associative, is not implemented. (All infix operators are by default left-associative until they are declared to be right-associative.)

See also: OpPrecedence

LeftPrecedence — set operator precedence

RightPrecedence — set operator precedence

(YACAS internal)

Calling format:

```
LeftPrecedence("op",precedence)
RightPrecedence("op",precedence)
```

Parameters:

"op" – string, the name of a function
precedence – nonnegative integer

Description:

"op" should be an infix operator. This function call tells the infix expression printer to bracket the left or right hand side of the expression if its precedence is larger than precedence.

This functionality was required in order to display expressions like a-(b-c) correctly. Thus, a+b+c is the same as a+(b+c), but a-(b-c) is not the same as a-b-c.

Note that the left and right precedence of an infix operator does not affect the way Yacas interprets expressions typed by the user. You cannot make Yacas parse a-b-c as a-(b-c) unless you declare the operator "-" to be right-associative.

See also: OpPrecedence, OpLeftPrecedence, OpRightPrecedence, RightAssociative

RuleBase — define function with a fixed number of arguments

(YACAS internal)

Calling format:

```
RuleBase(name,params)
```

Parameters:

name – string, name of function
params – list of arguments to function

Description:

Define a new rules table entry for a function "name", with params as the parameter list. Name can be either a string or simple atom.

In the context of the transformation rule declaration facilities this is a useful function in that it allows the stating of argument names that can he used with HoldArg.

Functions can be overloaded: the same function can be defined with different number of arguments.

See also: MacroRuleBase, RuleBaseListed, MacroRuleBaseListed, HoldArg, Retract

RuleBaseListed — define function with variable number of arguments

(YACAS internal)

Calling format:

```
RuleBaseListed("name", params)
```

Parameters:

"name" – string, name of function
params – list of arguments to function

Description:

The command RuleBaseListed defines a new function. It essentially works the same way as RuleBase, except that it declares a new function with a variable number of arguments. The list of parameters params determines the smallest number of arguments that the new function will accept. If the number of arguments passed to the new function is larger than the number of parameters in params, then the last argument actually passed to the new function will be a list containing all the remaining arguments.

A function defined using RuleBaseListed will appear to have the arity equal to the number of parameters in the param list, and it can accept any number of arguments greater or equal than that. As a consequence, it will be impossible to define a new function with the same name and with a greater arity.

The function body will know that the function is passed more arguments than the length of the param list, because the last argument will then be a list. The rest then works like a RuleBase-defined function with a fixed number of arguments. Transformation rules can be defined for the new function as usual.

Examples:

The definitions

```
RuleBaseListed("f",{a,b,c})
10 # f(_a,_b,{_c,_d}) <--
  Echo({"four args",a,b,c,d});
20 # f(_a,_b,c_IsList) <--
  Echo({"more than four args",a,b,c});
30 # f(_a,_b,_c) <-- Echo({"three args",a,b,c});
```

give the following interaction:

```
In> f(A)
Out> f(A);
In> f(A,B)
Out> f(A,B);
In> f(A,B,C)
three args A B C
```

```
Out> True;
In> f(A,B,C,D)
four args A B C D
Out> True;
In> f(A,B,C,D,E)
more than four args A B {C,D,E}
Out> True;
In> f(A,B,C,D,E,E)
more than four args A B {C,D,E,E}
Out> True;
```

The function `f` now appears to occupy all arities greater than 3:

```
In> RuleBase("f", {x,y,z,t});
CommandLine(1) : Rule base with this arity
  already defined
```

See also: `RuleBase`, `Retract`, `Echo`

`Rule` — define a rewrite rule

(YACAS internal)

Calling format:

```
Rule("operator", arity,
  precedence, predicate) body
```

Parameters:

`"operator"` – string, name of function
`arity`, `precedence` – integers
`predicate` – function returning boolean
`body` – expression, body of rule

Description:

Define a rule for the function "operator" with "arity", "precedence", "predicate" and "body". The "precedence" goes from low to high: rules with low precedence will be applied first.

The arity for a rules database equals the number of arguments. Different rules data bases can be built for functions with the same name but with a different number of arguments.

Rules with a low precedence value will be tried before rules with a high value, so a rule with precedence 0 will be tried before a rule with precedence 1.

`HoldArg` — mark argument as not evaluated

(YACAS internal)

Calling format:

```
HoldArg("operator",parameter)
```

Parameters:

`"operator"` – string, name of a function
`parameter` – atom, symbolic name of parameter

Description:

Specify that parameter should not be evaluated before used. This will be declared for all arities of "operator", at the moment this function is called, so it is best called after all `RuleBase` calls for this operator. "operator" can be a string or atom specifying the function name.

The `parameter` must be an atom from the list of symbolic arguments used when calling `RuleBase`.

See also: `RuleBase`, `HoldArgNr`, `RuleBaseArgList`

`Retract` — erase rules for a function

(YACAS internal)

Calling format:

```
Retract("function",arity)
```

Parameters:

`"function"` – string, name of function
`arity` – positive integer

Description:

Remove a rulebase for the function named `"function"` with the specific `arity`, if it exists at all. This will make Yacas forget all rules defined for a given function. Rules for functions with the same name but different arities are not affected.

Assignment `:=` of a function does this to the function being (re)defined.

See also: `RuleBaseArgList`, `RuleBase`, `:=`

`UnFence` — change local variable scope for a function

(YACAS internal)

Calling format:

```
UnFence("operator",arity)
```

Parameters:

`"operator"` – string, name of function
`arity` – positive integers

Description:

When applied to a user function, the bodies defined for the rules for "operator" with given arity can see the local variables from the calling function. This is useful for defining macro-like procedures (looping and such).

The standard library functions `For` and `ForEach` use `UnFence`.

`HoldArgNr` — specify argument as not evaluated

(standard library)

Calling format:

```
HoldArgNr("function", arity, argNum)
```

Parameters:

`"function"` – string, function name
`arity`, `argNum` – positive integers

Description:

Declares the argument numbered `argNum` of the function named `"function"` with specified `arity` to be unevaluated ("held"). Useful if you don't know symbolic names of parameters, for instance, when the function was not declared using an explicit `RuleBase` call. Otherwise you could use `HoldArg`.

See also: `HoldArg`, `RuleBase`

RuleBaseArgList — obtain list of arguments

(YACAS internal)

Calling format:

```
RuleBaseArgList("operator", arity)
```

Parameters:

`"operator"` – string, name of function
`arity` – integer

Description:

Returns a list of atoms, symbolic parameters specified in the `RuleBase` call for the function named `"operator"` with the specific `arity`.

See also: `RuleBase`, `HoldArgNr`, `HoldArg`

MacroSet — define rules in functions

MacroClear — define rules in functions

MacroLocal — define rules in functions

MacroRuleBase — define rules in functions

MacroRuleBaseListed — define rules in functions

MacroRule — define rules in functions

(YACAS internal)

Description:

These functions have the same effect as their non-macro counterparts, except that their arguments are evaluated before the required action is performed. This is useful in macro-like procedures or in functions that need to define new rules based on parameters.

Make sure that the arguments of `Macro...` commands evaluate to expressions that would normally be used in the non-macro versions!

See also: `Set`, `Clear`, `Local`, `RuleBase`, `Rule`, `Backquoting`

Backquoting — macro expansion (LISP-style backquoting)

(YACAS internal)

Calling format:

```
`(expression)
```

Parameters:

`expression` – expression containing "`@var`" combinations to substitute the value of variable "`var`"

Description:

Backquoting is a macro substitution mechanism. A backquoted `expression` is evaluated in two stages: first, variables prefixed by `@` are evaluated inside an expression, and second, the new expression is evaluated.

To invoke this functionality, a backquote ` needs to be placed in front of an expression. Parentheses around the expression are needed because the backquote binds tighter than other operators.

The expression should contain some variables (assigned atoms) with the special prefix operator `@`. Variables prefixed by `@` will be evaluated even if they are inside function arguments that are normally not evaluated (e.g. functions declared with `HoldArg`). If the `@var` pair is in place of a function name, e.g. "`@f(x)`", then at the first stage of evaluation the function name itself is replaced, not the return value of the function (see example); so at the second stage of evaluation, a new function may be called.

One way to view backquoting is to view it as a parametric expression generator. `@var` pairs get substituted with the value of the variable `var` even in contexts where nothing would be evaluated. This effect can be also achieved using `UnList` and `Hold` but the resulting code is much more difficult to read and maintain.

This operation is relatively slow since a new expression is built before it is evaluated, but nonetheless backquoting is a powerful mechanism that sometimes allows to greatly simplify code.

Examples:

This example defines a function that automatically evaluates to a number as soon as the argument is a number (a lot of functions do this only when inside a `N(...)` section).

```
In> Decl(f1,f2) := \
In>   `(@f1(x_IsNumber) <-- N(@f2(x)));
Out> True;
In> Decl(nSin,Sin)
Out> True;
In> Sin(1)
Out> Sin(1);
In> nSin(1)
Out> 0.8414709848;
```

This example assigns the expression `func(value)` to variable `var`. Normally the first argument of `Set` would be unevaluated.

```
In> SetF(var,func,value) := \
In>     `(Set(@var,@func(@value)));
Out> True;
In> SetF(a,Sin,x)
Out> True;
In> a
Out> Sin(x);
```

See also: `MacroSet`, `MacroLocal`, `MacroRuleBase`, `Hold`, `HoldArg`, `DefMacroRuleBase`

DefMacroRuleBase — define a function as a macro

(standard library)

Calling format:

```
DefMacroRuleBase(name,params)
```

Parameters:

`name` – string, name of a function
`params` – list of arguments

Description:

`DefMacroRuleBase` is similar to `RuleBase`, with the difference that it declares a macro, instead of a function. After this call, rules can be defined for the function "`name`", but their interpretation will be different.

With the usual functions, the evaluation model is that of the ¡i¿applicative-order model of substitution¡/i¿, meaning that first the arguments are evaluated, and then the function is applied to the result of evaluating these arguments. The function is entered, and the code inside the function can not access local variables outside of its own local variables.

With macros, the evaluation model is that of the *normal-order model of substitution*, meaning that all occurrences of variables in an expression are first substituted into the body of the macro, and only then is the resulting expression evaluated ¡i¿in its calling environment¡/i¿. This is important, because then in principle a macro body can access the local variables from the calling environment, whereas functions can not do that.

As an example, suppose there is a function `square`, which squares its argument, and a function `add`, which adds its arguments. Suppose the definitions of these functions are:

```
add(x,y) <-- x+y;
```

and

```
square(x) <-- x*x;
```

In applicative-order mode (the usual way functions are evaluated), in the following expression

```
add(square(2),square(3))
```

first the arguments to `add` get evaluated. So, first `square(2)` is evaluated. To evaluate this, first `2` is evaluated, but this evaluates to itself. Then the `square` function is applied to it, `2*2`, which returns 4. The same is done for `square(3)`, resulting in `9`. Only then, after evaluating these two arguments, `add` is applied to them, which is equivalent to

```
add(4,9)
```

resulting in calling `4+9`, which in turn results in `13`.

In contrast, when `add` is a macro, the arguments to `add` are first expanded. So

```
add(square(2),square(3))
```

first expands to

```
square(2) + square(3)
```

and then this expression is evaluated, as if the user had written it directly. In other words, `square(2)` is not evaluated before the macro has been fully expanded.

Macros are useful for customizing syntax, and compilers can potentially greatly optimize macros, as they can be inlined in the calling environment, and optimized accordingly.

There are disadvantages, however. In interpreted mode, macros are slower, as the requirement for substitution means that a new expression to be evaluated has to be created on the fly. Also, when one of the parameters to the macro occur more than once in the body of the macro, it is evaluated multiple times.

When defining transformation rules for macros, the variables to be substituted need to be preceded by the `@` operator, similar to the back-quoting mechanism. Apart from that, the two are similar, and all transformation rules can also be applied to macros.

Macros can co-exist with functions with the same name but different arity. For instance, one can have a function `foo(a,b)` with two arguments, and a macro `foo(a,b,c)` with three arguments.

Example:

The following example defines a function `myfor`, and shows one use, referencing a variable `a` from the calling environment.

```
In> DefMacroRuleBase("myfor",{init,pred,inc,body})
Out> True;
In> myfor(_init,_pred,_inc,_body)<--[@init;While(@pred)[@bo
Out> True;
In> a:=10
Out> 10;
In> myfor(i:=1,i<10,i++,Echo(a*i))
10
20
30
40
50
60
70
80
90
Out> True;
In> i
Out> 10;
```

See also: `RuleBase`, `Backquoting`, `DefMacroRuleBaseListed`

DefMacroRuleBaseListed — define macro with variable number of arguments

(YACAS internal)

Calling format:

```
DefMacroRuleBaseListed("name", params)
```

Parameters:

`"name"` – string, name of function
`params` – list of arguments to function

Description:

This does the same as `DefMacroRuleBase` (define a macro), but with a variable number of arguments, similar to `RuleBaseListed`.

See also: `RuleBase`, `RuleBaseListed`, `Backquoting`, `DefMacroRuleBase`

ExtraInfo'Set, ExtraInfo'Get — annotate objects with additional information

(YACAS internal)

Calling format:

```
ExtraInfo'Set(expr,tag)
ExtraInfo'Get(expr)
```

Parameters:

`expr` – any expression
`tag` – tag information (any other expression)

Description:

Sometimes it is useful to be able to add extra tag information to "annotate" objects or to label them as having certain "properties". The functions `ExtraInfo'Set` and `ExtraInfo'Get` enable this.

The function `ExtraInfo'Set` returns the tagged expression, leaving the original expression alone. This means there is a common pitfall: be sure to assign the returned value to a variable, or the tagged expression is lost when the temporary object is destroyed.

The original expression is left unmodified, and the tagged expression returned, in order to keep the atomic objects small. To tag an object, a new type of object is created from the old object, with one added property (the tag). The tag can be any expression whatsoever.

The function `ExtraInfo'Get(x)` retrieves this tag expression from an object `x`. If an object has no tag, it looks the same as if it had a tag with value `False`.

No part of the Yacas core uses tags in a way that is visible to the outside world, so for specific purposes a programmer can devise a format to use for tag information. Association lists (hashes) are a natural fit for this, although it is not required and a tag can be any object (except the atom `False` because it is indistinguishable from having no tag information). Using association lists is highly advised since it is most likely to be the format used by other parts of the library, and one needs to avoid clashes with other library code. Typically, an object will either have no tag or a tag which is an associative list (perhaps empty). A script that uses tagged objects will check whether an object has a tag and if so, will add or modify certain entries of the association list, preserving any other tag information.

Note that `FlatCopy` currently does *not* copy the tag information (see examples).

Examples:

```
In> a:=2*b
Out> 2*b;
In> a:=ExtraInfo'Set(a,{{"type","integer"}})
Out> 2*b;
In> a
Out> 2*b;
In> ExtraInfo'Get(a)
Out> {{"type","integer"}};
In> ExtraInfo'Get(a)["type"]
Out> "integer";
In> c:=a
Out> 2*b;
In> ExtraInfo'Get(c)
Out> {{"type","integer"}};
In> c
Out> 2*b;
In> d:=FlatCopy(a);
Out> 2*b;
In> ExtraInfo'Get(d)
Out> False;
```

See also: `Assoc`, `:=`

GarbageCollect — do garbage collection on unused memory

(YACAS internal)

Calling format:

```
GarbageCollect()
```

Description:

`GarbageCollect` garbage-collects unused memory. The Yacas system uses a reference counting system for most objects, so this call is usually not necessary.

Reference counting refers to bookkeeping where in each object a counter is held, keeping track of the number of parts in the system using that object. When this count drops to zero, the object is automatically removed. Reference counting is not the fastest way of doing garbage collection, but it can be implemented in a very clean way with very little code.

Among the most important objects that are not reference counted are the strings. `GarbageCollect` collects these and disposes of them when they are not used any more.

`GarbageCollect` is useful when doing a lot of text processing, to clean up the text buffers. It is not highly needed, but it keeps memory use low.

FindFunction — find the library file where a function is defined

(YACAS internal)

Calling format:

```
FindFunction(function)
```

Parameters:

`function` – string, the name of a function

Description:

This function is useful for quickly finding the file where a standard library function is defined. It is likely to only be useful for developers. The function `FindFunction` scans the `.def` files that were loaded at start-up. This means that functions that are not listed in `.def` files will not be found with `FindFunction`.

Examples:

```
In> FindFunction("Sum")
Out> "sums.rep/code.ys";
In> FindFunction("Integrate")
Out> "integrate.rep/code.ys";
```

See also: `Vi`

`Secure` — guard the host OS

(YACAS internal)

Calling format:

```
Secure(body)
```

Parameters:

`body` – expression

Description:

`Secure` evaluates `body` in a "safe" environment, where files cannot be opened and system calls are not allowed. This can help protect the system when e.g. a script is sent over the Internet to be evaluated on a remote computer, which is potentially unsafe.

See also: `SystemCall`

Chapter 3

Arbitrary-precision numerical programming

This chapter contains functions that help programming numerical calculations with arbitrary precision.

MultiplyNum — optimized numerical multiplication

(standard library)

Calling format:

```
MultiplyNum(x,y)
MultiplyNum(x,y,z,...)
MultiplyNum({x,y,z,...})
```

Parameters:

x, y, z – integer, rational or floating-point numbers to multiply

Description:

The function `MultiplyNum` is used to speed up multiplication of floating-point numbers with rational numbers. Suppose we need to compute $\frac{p}{q}x$ where p, q are integers and x is a floating-point number. At high precision, it is faster to multiply x by an integer p and divide by an integer q than to compute $\frac{p}{q}$ to high precision and then multiply by x. The function `MultiplyNum` performs this optimization.

The function accepts any number of arguments (not less than two) or a list of numbers. The result is always a floating-point number (even if `InNumericMode()` returns False).

See also: `MathMultiply`

CachedConstant — precompute multiple-precision constants

(standard library)

Calling format:

```
CachedConstant(cache, Cname, Cfunc)
```

Parameters:

`cache` – atom, name of the cache
`Cname` – atom, name of the constant
`Cfunc` – expression that evaluates the constant

Description:

This function is used to create precomputed multiple-precision values of constants. Caching these values will save time if they are frequently used.

The call to `CachedConstant` defines a new function named `Cname()` that returns the value of the constant at given precision. If the precision is increased, the value will be recalculated as necessary, otherwise calling `Cname()` will take very little time.

The parameter `Cfunc` must be an expression that can be evaluated and returns the value of the desired constant at the current precision. (Most arbitrary-precision mathematical functions do this by default.)

The associative list `cache` contains elements of the form {`Cname, prec, value`}, as illustrated in the example. If this list does not exist, it will be created.

This mechanism is currently used by `N()` to precompute the values of π and γ (and the golden ratio through `GoldenRatio`, and `Catalan`). The name of the cache for `N()` is `CacheOfConstantsN`. The code in the function `N()` assigns unevaluated calls to `Internal'Pi()` and `Internal'gamma()` to the atoms `Pi` and `gamma` and declares them to be lazy global variables through `SetGlobalLazyVariable` (with equivalent functions assigned to other constants that are added to the list of cached constants).

The result is that the constants will be recalculated only when they are used in the expression under `N()`. In other words, the code in `N()` does the equivalent of

```
SetGlobalLazyVariable(mypi,Hold(Internal'Pi()));
SetGlobalLazyVariable(mygamma,Hold(Internal'gamma()));
```

After this, evaluating an expression such as `1/2+gamma` will call the function `Internal'gamma()` but not the function `Internal'Pi()`.

Example:

```
In> CachedConstant( my'cache, Ln2, Internal'LnNum(2) )
Out> True;
In> Internal'Ln2()
Out> 0.6931471806;
In> V(N(Internal'Ln2(),20))
CachedConstant: Info: constant Ln2 is being
  recalculated at precision 20
Out> 0.69314718055994530942;
In> my'cache
Out> {{"Ln2",20,0.69314718055994530942}};
```

See also: `N`, `Builtin'Precision'Set`, `Pi`, `GoldenRatio`, `Catalan`, `gamma`

NewtonNum — low-level optimized Newton's iterations

(standard library)

Calling format:

```
NewtonNum(func, x0, prec0, order)
NewtonNum(func, x0, prec0)
NewtonNum(func, x0)
```

Parameters:

`func` – a function specifying the iteration sequence
`x0` – initial value (must be close enough to the root)
`prec0` – initial precision (at least 4, default 5)
`order` – convergence order (typically 2 or 3, default 2)

Description:

This function is an optimized interface for computing Newton's iteration sequences for numerical solution of equations in arbitrary precision.

`NewtonNum` will iterate the given function starting from the initial value, until the sequence converges within current precision. Initially, up to 5 iterations at the initial precision `prec0` is performed (the low precision is set for speed). The initial value `x0` must be close enough to the root so that the initial iterations converge. If the sequence does not produce even a single correct digit of the root after these initial iterations, an error message is printed. The default value of the initial precision is 5.

The `order` parameter should give the convergence order of the scheme. Normally, Newton iteration converges quadratically (so the default value is `order`=2) but some schemes converge faster and you can speed up this function by specifying the correct order. (Caution: if you give `order`=3 but the sequence is actually quadratic, the result will be silently incorrect. It is safe to use `order`=2.)

Example:

```
In> Builtin'Precision'Set(20)
Out> True;
In> NewtonNum({{x}, x+Sin(x)}, 3, 5, 3)
Out> 3.14159265358979323846;
```

See also: `Newton`

SumTaylorNum — optimized numerical evaluation of Taylor series

(standard library)

Calling format:

```
SumTaylorNum(x, NthTerm, order)
SumTaylorNum(x, NthTerm, TermFactor, order)
SumTaylorNum(x, ZerothTerm, TermFactor, order)
```

Parameters:

`NthTerm` – a function specifying n-th coefficient of the series
`ZerothTerm` – value of the 0-th coefficient of the series
`x` – number, value of the expansion variable
`TermFactor` – a function specifying the ratio of n-th term to the previous one
`order` – power of x in the last term

Description:

`SumTaylorNum` computes a Taylor series $\sum_{k=0}^{n} a_k x^k$ numerically. This function allows very efficient computations of functions given by Taylor series, although some tweaking of the parameters is required for good results.

The coefficients a_k of the Taylor series are given as functions of one integer variable (k). It is convenient to pass them to `SumTaylorNum` as closures. For example, if a function `a(k)` is defined, then

```
SumTaylorNum(x, {{k}, a(k)}, n)
```

computes the series $\sum_{k=0}^{n} a\,(k)\,x^k$.

Often a simple relation between successive coefficients a_{k-1}, a_k of the series is available; usually they are related by a rational factor. In this case, the second form of `SumTaylorNum` should be used because it will compute the series faster. The function `TermFactor` applied to an integer $k \geq 1$ must return the ratio a_k/a_{k-1}. (If possible, the function `TermFactor` should return a rational number and not a floating-point number.) The function `NthTerm` may also be given, but the current implementation only calls `NthTerm(0)` and obtains all other coefficients by using `TermFactor`. Instead of the function `NthTerm`, a number giving the 0-th term can be given.

The algorithm is described elsewhere in the documentation. The number of terms `order`+1 must be specified and a sufficiently high precision must be preset in advance to achieve the desired accuracy. (The function `SumTaylorNum` does not change the current precision.)

Examples:

To compute 20 digits of exp (1) using the Taylor series, one needs 21 digits of working precision and 21 terms of the series.

```
In> Builtin'Precision'Set(21)
Out> True;
In> SumTaylorNum(1, {{k},1/k!}, 21)
Out> 2.718281828459045235351;
In> SumTaylorNum(1, 1, {{k},1/k}, 21)
Out> 2.71828182845904523535;
In> SumTaylorNum(1, {{k},1/k!}, {{k},1/k}, 21)
Out> 2.71828182845904523535;
In> RoundTo(N(Ln(%)),20)
Out> 1;
```

See also: `Taylor`

IntPowerNum — optimized computation of integer powers

(standard library)

Calling format:

```
IntPowerNum(x, n, mult, unity)
```

Parameters:

`x` – a number or an expression
`n` – a non-negative integer (power to raise `x` to)
`mult` – a function that performs one multiplication
`unity` – value of the unity with respect to that multiplication

Description:

`IntPowerNum` computes the power x^n using the fast binary algorithm. It can compute integer powers with $n \geq 0$ in any ring where multiplication with unity is defined. The multiplication function and the unity element must be specified. The number of multiplications is no more than $2\frac{\ln n}{\ln 2}$.

Mathematically, this function is a generalization of `MathPower` to rings other than that of real numbers.

In the current implementation, the `unity` argument is only used when the given power `n` is zero.

Examples:

For efficient numerical calculations, the `MathMultiply` function can be passed:

```
In> IntPowerNum(3, 3, MathMultiply,1)
Out> 27;
```

Otherwise, the usual `*` operator suffices:

```
In> IntPowerNum(3+4*I, 3, *,1)
Out> Complex(-117,44);
In> IntPowerNum(HilbertMatrix(2), 4, *,
  Identity(2))
Out> {{289/144,29/27},{29/27,745/1296}};
```

Compute 3^{100} mod 7:

```
In> IntPowerNum(3,100,{{x,y},Mod(x*y,7)},1)
Out> 4;
```

See also: `MultiplyNum`, `MathPower`, `MatrixPower`

BinSplitNum — computations of series by the binary splitting method

BinSplitData — computations of series by the binary splitting method

BinSplitFinal — computations of series by the binary splitting method

(standard library)

Calling format:

```
BinSplitNum(n1, n2, a, b, c, d)
BinSplitData(n1,n2, a, b, c, d)
BinSplitFinal({P,Q,B,T})
```

Parameters:

`n1`, `n2` – integers, initial and final indices for summation

`a`, `b`, `c`, `d` – functions of one argument, coefficients of the series

`P`, `Q`, `B`, `T` – numbers, intermediate data as returned by `BinSplitData`

Description:

The binary splitting method is an efficient way to evaluate many series when fast multiplication is available and when the series contains only rational numbers. The function `BinSplitNum` evaluates a series of the form

$$S(n_1, n_2) = \sum_{k=n_1}^{n_2} \frac{a(k)}{b(k)} \frac{p(0)}{q(0)} \cdots \frac{p(k)}{q(k)}.$$

Most series for elementary and special functions at rational points are of this form when the functions $a(k)$, $b(k)$, $p(k)$, $q(k)$ are chosen appropriately.

The last four arguments of `BinSplitNum` are functions of one argument that give the coefficients $a(k)$, $b(k)$, $p(k)$, $q(k)$. In most cases these will be short integers that are simple to determine. The binary splitting method will work also for non-integer coefficients, but the calculation will take much longer in that case.

Note: the binary splitting method outperforms the straightforward summation only if the multiplication of integers is faster than quadratic in the number of digits. See *the algorithm documentation*, in *The Yacas book of algorithms, Chapter 3, Section 14* for more information.

The two other functions are low-level functions that allow a finer control over the calculation. The use of the low-level routines allows checkpointing or parallelization of a binary splitting calculation.

The binary splitting method recursively reduces the calculation of $S(n_1, n_2)$ to the same calculation for the two halves of the interval $[n_1,\ n_2]$. The intermediate results of a binary splitting calculation are returned by `BinSplitData` and consist of four integers P, Q, B, T. These four integers are converted into the final answer S by the routine `BinSplitFinal` using the relation

$$S = \frac{T}{BQ}.$$

Examples:

Compute the series for $e = \exp(1)$ using binary splitting. (We start from $n = 1$ to simplify the coefficient functions.)

```
In> Builtin'Precision'Set(21)
Out> True;
In>  BinSplitNum(1,21, {{k},1},
  {{k},1},{{k},1},{{k},k})
Out> 1.718281828459045235359;
In> N(Exp(1)-1)
Out> 1.71828182845904523536;
In>  BinSplitData(1,21, {{k},1},
  {{k},1},{{k},1},{{k},k})
Out> {1,51090942171709440000,1,
  87788637532500240022};
In> BinSplitFinal(%)
Out> 1.718281828459045235359;
```

See also: `SumTaylorNum`

MathSetExactBits — manipulate precision of floating-point numbers

MathGetExactBits — manipulate precision of floating-point numbers

(Yacas internal)

Calling format:

```
MathGetExactBits(x)
MathSetExactBits(x,bits)
```

Parameters:

`x` – an expression evaluating to a floating-point number

`bits` – integer, number of bits

Description:

Each floating-point number in Yacas has an internal precision counter that stores the number of exact bits in the mantissa. The number of exact bits is automatically updated after each arithmetic operation to reflect the gain or loss of precision due to round-off. The functions `MathGetExactBits`, `MathSetExactBits` allow to query or set the precision flags of individual number objects.

`MathGetExactBits(x)` returns an integer number n such that `x` represents a real number in the interval $[x\left(1-2^{-n}\right), x\left(1+2^{-n}\right)]$ if $x \neq 0$ and in the interval $[-2^{-n}, 2^{-n}]$ if $x = 0$. The integer n is always nonnegative unless `x` is zero (a "floating zero"). A floating zero can have a negative value of the number n of exact bits.

These functions are only meaningful for floating-point numbers. (All integers are always exact.) For integer `x`, the function `MathGetExactBits` returns the bit count of `x` and the function `MathSetExactBits` returns the unmodified integer `x`.

Examples:

The default precision of 10 decimals corresponds to 33 bits:

```
In> MathGetExactBits(1000.123)
Out> 33;
In> x:=MathSetExactBits(10., 20)
Out> 10.;
In> MathGetExactBits(x)
Out> 20;
```

Prepare a "floating zero" representing an interval [-4, 4]:

```
In> x:=MathSetExactBits(0., -2)
Out> 0.;
In> x=0
Out> True;
```

See also: `Builtin'Precision'Set`, `Builtin'Precision'Get`

InNumericMode — determine if currently in numeric mode

NonN — calculate part in non-numeric mode

(standard library)

Calling format:

```
NonN(expr)
InNumericMode()
```

Parameters:

`expr` – expression to evaluate
`prec` – integer, precision to use

Description:

When in numeric mode, `InNumericMode()` will return `True`, else it will return `False`. YACAS is in numeric mode when evaluating an expression with the function `N`. Thus when calling `N(expr)`, `InNumericMode()` will return `True` while `expr` is being evaluated.

`InNumericMode()` would typically be used to define a transformation rule that defines how to get a numeric approximation of some expression. One could define a transformation rule

```
f(_x)_InNumericMode() <- [... some code to get a numeric ap
```

`InNumericMode()` usually returns `False`, so transformation rules that check for this predicate are usually left alone.

When in numeric mode, `NonN` can be called to switch back to non-numeric mode temporarily.

`NonN` is a macro. Its argument `expr` will only be evaluated after the numeric mode has been set appropriately.

Examples:

```
In> InNumericMode()
Out> False
In> N(InNumericMode())
Out> True
In> N(NonN(InNumericMode()))
Out> False
```

See also: `N`, `Builtin'Precision'Set`, `Builtin'Precision'Get`, `Pi`, `CachedConstant`

IntLog — integer part of logarithm

(standard library)

Calling format:

```
IntLog(n, base)
```

Parameters:

`n`, `base` – positive integers

Description:

`IntLog` calculates the integer part of the logarithm of `n` in base `base`. The algorithm uses only integer math and may be faster than computing

$$\frac{\ln n}{\ln \text{base}}$$

with multiple precision floating-point math and rounding off to get the integer part.

This function can also be used to quickly count the digits in a given number.

Examples:

Count the number of bits:

```
In> IntLog(257^8, 2)
Out> 64;
```

Count the number of decimal digits:

```
In> IntLog(321^321, 10)
Out> 804;
```

See also: `IntNthRoot`, `Div`, `Mod`, `Ln`

IntNthRoot — integer part of n-th root

(standard library)

Calling format:

```
IntNthRoot(x, n)
```

Parameters:

`x`, `n` – positive integers

Description:

`IntNthRoot` calculates the integer part of the n-th root of x. The algorithm uses only integer math and may be faster than computing $\sqrt[n]{x}$ with floating-point and rounding.

This function is used to test numbers for prime powers.

Example:

```
In> IntNthRoot(65537^111, 37)
Out> 281487861809153;
```

See also: `IntLog`, `MathPower`, `IsPrimePower`

NthRoot — calculate/simplify nth root of an integer

(standard library)

Calling format:

```
NthRoot(m,n)
```

Parameters:

`m` – a non-negative integer ($m > 0$)
`n` – a positive integer greater than 1 ($n > 1$)

Description:

`NthRoot(m,n)` calculates the integer part of the n-th root $\sqrt[n]{m}$ and returns a list {`f`,`r`}. `f` and `r` are both positive integers that satisfy $f^n r{=}m$. In other words, f is the largest integer such that m divides f^n and r is the remaining factor.

For large `m` and small `n` `NthRoot` may work quite slowly. Every result {`f`,`r`} for given `m`, `n` is saved in a lookup table, thus subsequent calls to `NthRoot` with the same values `m`, `n` will be executed quite fast.

Example:

```
In> NthRoot(12,2)
Out> {2,3};
In> NthRoot(81,3)
Out> {3,3};
In> NthRoot(3255552,2)
Out> {144,157};
In> NthRoot(3255552,3)
Out> {12,1884};
```

See also: `IntNthRoot`, `Factors`, `MathPower`

ContFracList — manipulate continued fractions

ContFracEval — manipulate continued fractions

(standard library)

Calling format:

```
ContFracList(frac)
ContFracList(frac, depth)
ContFracEval(list)
ContFracEval(list, rest)
```

Parameters:

`frac` – a number to be expanded
`depth` – desired number of terms
`list` – a list of coefficients
`rest` – expression to put at the end of the continued fraction

Description:

The function `ContFracList` computes terms of the continued fraction representation of a rational number `frac`. It returns a list of terms of length `depth`. If `depth` is not specified, it returns all terms.

The function `ContFracEval` converts a list of coefficients into a continued fraction expression. The optional parameter `rest` specifies the symbol to put at the end of the expansion. If it is not given, the result is the same as if `rest=0`.

Examples:

```
In> A:=ContFracList(33/7 + 0.000001)
Out> {4,1,2,1,1,20409,2,1,13,2,1,4,1,1,3,3,2};
In> ContFracEval(Take(A, 5))
Out> 33/7;
In> ContFracEval(Take(A,3), remainder)
Out> 1/(1/(remainder+2)+1)+4;
```

See also: `ContFrac`, `GuessRational`

GuessRational — find optimal rational approximations

NearRational — find optimal rational approximations

BracketRational — find optimal rational approximations

(standard library)

Calling format:

```
GuessRational(x)
GuessRational(x, digits)
NearRational(x)
NearRational(x, digits)
BracketRational(x, eps)
```

Parameters:

`x` – a number to be approximated (must be already evaluated to floating-point)
`digits` – desired number of decimal digits (integer)
`eps` – desired precision

Description:

The functions `GuessRational(x)` and `NearRational(x)` attempt to find "optimal" rational approximations to a given value `x`. The approximations are "optimal" in the sense of having smallest numerators and denominators among all rational numbers close to `x`. This is done by computing a continued fraction representation of `x` and truncating it at a suitably chosen term. Both functions return a rational number which is an approximation of `x`.

Unlike the function `Rationalize()` which converts floating-point numbers to rationals without loss of precision, the functions `GuessRational()` and `NearRational()` are intended to find the best rational that is *approximately* equal to a given value.

The function `GuessRational()` is useful if you have obtained a floating-point representation of a rational number and you know approximately how many digits its exact representation should contain. This function takes an optional second parameter `digits` which limits the number of decimal digits in the denominator of the resulting rational number. If this parameter is not given, it defaults to half the current precision. This function truncates the continuous fraction expansion when it encounters an unusually large value (see example). This procedure does not always give the "correct" rational number; a rule of thumb is that the floating-point number should have at least as many digits as the combined number of digits in the numerator and the denominator of the correct rational number.

The function `NearRational(x)` is useful if one needs to approximate a given value, i.e. to find an "optimal" rational number that lies in a certain small interval around a certain value `x`. This function takes an optional second parameter `digits` which has slightly different meaning: it specifies the number of digits of precision of the approximation; in other words, the difference between `x` and the resulting rational number should be at most one digit of that precision. The parameter `digits` also defaults to half of the current precision.

The function `BracketRational(x,eps)` can be used to find approximations with a given relative precision from above and from below. This function returns a list of two rational numbers {`r1`,`r2`} such that $r_1 < x < r_2$ and $|r_2 - r_1| < |x\text{eps}|$. The argument `x` must be already evaluated to enough precision so that this approximation can be meaningfully found. If the approximation with the desired precision cannot be found, the function returns an empty list.

Examples:

Start with a rational number and obtain a floating-point approximation:

```
In> x:=N(956/1013)
Out> 0.9437314906
In> Rationalize(x)
Out> 4718657453/5000000000;
In> V(GuessRational(x))

GuessRational: using 10 terms of the
  continued fraction
Out> 956/1013;
In> ContFracList(x)
Out> {0,1,16,1,3,2,1,1,1,1,508848,3,1,2,1,2,2};
```

The first 10 terms of this continued fraction correspond to the correct continued fraction for the original rational number.

```
In> NearRational(x)
Out> 218/231;
```

This function found a different rational number closeby because the precision was not high enough.

```
In> NearRational(x, 10)
Out> 956/1013;
```

Find an approximation to $\ln 10$ good to 8 digits:

```
In> BracketRational(N(Ln(10)), 10^(-8))
Out> {12381/5377,41062/17833};
```

See also: `ContFrac`, `ContFracList`, `Rationalize`

TruncRadian — remainder modulo 2π

(standard library)

Calling format:

```
TruncRadian(r)
```

Parameters:

`r` – a number

Description:

`TruncRadian` calculates r mod (2π), returning a value between 0 and 2π. This function is used in the trigonometry functions, just before doing a numerical calculation using a Taylor series. It greatly speeds up the calculation if the value passed is a large number.

The library uses the formula

$$\text{TruncRadian}\,(r) = r - \left\lfloor \frac{r}{2\pi} \right\rfloor \cdot 2\pi,$$

where r and 2π are calculated with twice the precision used in the environment to make sure there is no rounding error in the significant digits.

Examples:

```
In> 2*Internal'Pi()
Out> 6.283185307;
In> TruncRadian(6.28)
Out> 6.28;
In> TruncRadian(6.29)
Out> 0.0068146929;
```

See also: `Sin`, `Cos`, `Tan`

Builtin'Precision'Set — set the precision

(YACAS internal)

Calling format:

```
Builtin'Precision'Set(n)
```

Parameters:

`n` – integer, new value of precision

Description:

This command sets the number of decimal digits to be used in calculations. All subsequent floating point operations will allow for at least `n` digits of mantissa.

This is not the number of digits after the decimal point. For example, `123.456` has 3 digits after the decimal point and 6 digits of mantissa. The number `123.456` is adequately computed by specifying `Builtin'Precision'Set(6)`.

The call `Builtin'Precision'Set(n)` will not guarantee that all results are precise to `n` digits.

When the precision is changed, all variables containing previously calculated values remain unchanged. The `Builtin'Precision'Set` function only makes all further calculations proceed with a different precision.

Also, when typing floating-point numbers, the current value of `Builtin'Precision'Set` is used to implicitly determine the number of precise digits in the number.

Examples:

```
In> Builtin'Precision'Set(10)
Out> True;
In> N(Sin(1))
Out> 0.8414709848;
In> Builtin'Precision'Set(20)
Out> True;
In> x:=N(Sin(1))
Out> 0.84147098480789650665;
```

The value `x` is not changed by a `Builtin'Precision'Set()` call:

```
In> [ Builtin'Precision'Set(10); x; ]
Out> 0.84147098480789650665;
```

The value `x` is rounded off to 10 digits after an arithmetic operation:

```
In> x+0.
Out> 0.8414709848;
```

In the above operation, `0.` was interpreted as a number which is precise to 10 digits (the user does not need to type `0.0000000000` for this to happen). So the result of `x+0.` is precise only to 10 digits.

See also: `Builtin'Precision'Get`, `N`

`Builtin'Precision'Get` — get the current precision

(YACAS internal)

Calling format:

```
Builtin'Precision'Get()
```

Description:

This command returns the current precision, as set by `Builtin'Precision'Set`.

Examples:

```
In> Builtin'Precision'Get();
Out> 10;
In> Builtin'Precision'Set(20);
Out> True;
In> Builtin'Precision'Get();
Out> 20;
```

See also: `Builtin'Precision'Set`, `N`

Chapter 4

Error reporting

This chapter contains commands useful for reporting errors to the user.

Check — report "hard" errors

TrapError — trap "hard" errors

GetCoreError — get "hard" error string

(YACAS internal)

Calling format:

```
Check(predicate,"error text")
TrapError(expression,errorHandler)
GetCoreError()
```

Parameters:

`predicate` – expression returning `True` or `False`
`"error text"` – string to print on error
`expression` – expression to evaluate (causing potential error)
`errorHandler` – expression to be called to handle error

Description:

If `predicate` does not evaluate to `True`, the current operation will be stopped, the string `"error text"` will be printed, and control will be returned immediately to the command line. This facility can be used to assure that some condition is satisfied during evaluation of expressions (guarding against critical internal errors).

A "soft" error reporting facility that does not stop the execution is provided by the function `Assert`.

Example:

```
In> [Check(1=0,"bad value"); Echo(OK);]
In function "Check" :
CommandLine(1) : "bad value"
```

Note that `OK` is not printed.

TrapError evaluates its argument `expression`, returning the result of evaluating `expression`. If an error occurs, `errorHandler` is evaluated, returning its return value in stead.

GetCoreError returns a string describing the core error. TrapError and GetCoreError can be used in combination to write a custom error handler.

See also: `Assert`

Assert — signal "soft" custom error

(standard library)

Calling format:

```
Assert("str", expr) pred
Assert("str") pred
Assert() pred
```

Precedence: 60000

Parameters:

`pred` – predicate to check
`"str"` – string to classify the error
`expr` – expression, error object

Description:

`Assert` is a global error reporting mechanism. It can be used to check for errors and report them. An error is considered to occur when the predicate `pred` evaluates to anything except `True`. In this case, the function returns `False` and an error object is created and posted to the global error tableau. Otherwise the function returns `True`.

Unlike the "hard" error function `Check`, the function `Assert` does not stop the execution of the program.

The error object consists of the string `"str"` and an arbitrary expression `expr`. The string should be used to classify the kind of error that has occurred, for example "domain" or "format". The error object can be any expression that might be useful for handling the error later; for example, a list of erroneous values and explanations. The association list of error objects is currently obtainable through the function `GetErrorTableau()`.

If the parameter `expr` is missing, `Assert` substitutes `True`. If both optional parameters `"str"` and `expr` are missing, `Assert` creates an error of class `"generic"`.

Errors can be handled by a custom error handler in the portion of the code that is able to handle a certain class of errors. The functions `IsError`, `GetError` and `ClearError` can be used.

Normally, all errors posted to the error tableau during evaluation of an expression should be eventually printed to the screen. This is the behavior of prettyprinters `DefaultPrint`, `Print`, `PrettyForm` and `TeXForm` (but not of the inline prettyprinter, which is enabled by default); they call `DumpErrors` after evaluating the expression.

Examples:

```
In> Assert("bad value", "must be zero") 1=0
Out> False;
In> Assert("bad value", "must be one") 1=1
```

```
Out> True;
In> IsError()
Out> True;
In> IsError("bad value")
Out> True;
In> IsError("bad file")
Out> False;
In> GetError("bad value");
Out> "must be zero";
In> DumpErrors()
Error: bad value: must be zero
Out> True;
```

No more errors left:

```
In> IsError()
Out> False;
In> DumpErrors()
Out> True;
```

See also: IsError, DumpErrors, Check, GetError, ClearError, ClearErrors, GetErrorTableau

DumpErrors — simple error handlers

ClearErrors — simple error handlers

(standard library)

Calling format:

```
DumpErrors()
ClearErrors()
```

Description:

DumpErrors is a simple error handler for the global error reporting mechanism. It prints all errors posted using Assert and clears the error tableau.

ClearErrors is a trivial error handler that does nothing except it clears the tableau.

See also: Assert, IsError

IsError — check for custom error

(standard library)

Calling format:

```
IsError()
IsError("str")
```

Parameters:

"str" – string to classify the error

Description:

IsError() returns True if any custom errors have been reported using Assert. The second form takes a parameter "str" that designates the class of the error we are interested in. It returns True if any errors of the given class "str" have been reported.

See also: GetError, ClearError, Assert, Check

GetError — custom errors handlers

ClearError — custom errors handlers

GetErrorTableau — custom errors handlers

(standard library)

Calling format:

```
GetError("str")
ClearError("str")
GetErrorTableau()
```

Parameters:

"str" – string to classify the error

Description:

These functions can be used to create a custom error handler.

GetError returns the error object if a custom error of class "str" has been reported using Assert, or False if no errors of this class have been reported.

ClearError("str") deletes the same error object that is returned by GetError("str"). It deletes at most one error object. It returns True if an object was found and deleted, and False otherwise.

GetErrorTableau() returns the entire association list of currently reported errors.

Examples:

```
In> x:=1
Out> 1;
In> Assert("bad value", {x,"must be zero"}) x=0
Out> False;
In> GetError("bad value")
Out> {1, "must be zero"};
In> ClearError("bad value");
Out> True;
In> IsError()
Out> False;
```

See also: IsError, Assert, Check, ClearErrors

CurrentFile — return current input file

CurrentLine — return current line number on input

(Yacas internal)

Calling format:

```
CurrentFile()
CurrentLine()
```

Description:

The functions `CurrentFile` and `CurrentLine` return a string with the file name of the current file and the current line of input respectively.

These functions are most useful in batch file calculations, where there is a need to determine at which line an error occurred. One can define a function

```
tst() := Echo({CurrentFile(),CurrentLine()});
```

which can then be inserted into the input file at various places, to see how far the interpreter reaches before an error occurs.

See also: `Echo`

Chapter 5

Built-in (core) functions

Yacas comes with a small core of built-in functions and a large library of user-defined functions. Some of these core functions are documented in this chapter.

It is important for a developer to know which functions are built-in and cannot be redefined or `Retract`-ed. Also, core functions may be somewhat faster to execute than functions defined in the script library. All core functions are listed in the file `corefunctions.h` in the `src/` subdirectory of the Yacas source tree. The declarations typically look like this:

```
SetCommand(LispSubtract, "MathSubtract");
```

Here `LispSubtract` is the Yacas internal name for the function and `MathSubtract` is the name visible to the Yacas language. Built-in bodied functions and infix operators are declared in the same file.

`MathNot` — built-in logical "not"

(YACAS internal)

Calling format:

```
MathNot(expression)
```

Description:

Returns "False" if "expression" evaluates to "True", and vice versa.

`MathAnd` — built-in logical "and"

Calling format:

```
MathAnd(...)
```

Description:

Lazy logical `And`: returns `True` if all args evaluate to `True`, and does this by looking at first, and then at the second argument, until one is `False`. If one of the arguments is `False`, `And` immediately returns `False` without evaluating the rest. This is faster, but also means that none of the arguments should cause side effects when they are evaluated.

`MathOr` — built-in logical "or"

(YACAS internal)

Calling format:

```
MathOr(...)
```

`MathOr` is the basic logical "or" function. Similarly to `And`, it is lazy-evaluated. `And(...)` and `Or(...)` do also exist, defined in the script library. You can redefine them as infix operators yourself, so you have the choice of precedence. In the standard scripts they are in fact declared as infix operators, so you can write `expr1 And expr`.

`BitAnd` — bitwise and operation

`BitOr` — bitwise or operation

`BitXor` — bitwise xor operation

(YACAS internal)

Calling format:

```
BitAnd(n,m)
BitOr(n,m)
BitXor(n,m)
```

Description:

These functions return bitwise "and", "or" and "xor" of two numbers.

`Equals` — check equality

(YACAS internal)

Calling format:

```
Equals(a,b)
```

Description:

Compares evaluated `a` and `b` recursively (stepping into expressions). So "Equals(a,b)" returns "True" if the expressions would be printed exactly the same, and "False" otherwise.

`GreaterThan` — comparison predicate

`LessThan` — comparison predicate

(YACAS internal)

Calling format:

```
GreaterThan(a,b)
LessThan(a,b)
```

Parameters:

`a`, `b` – numbers or strings

Description:

Comparing numbers or strings (lexicographically).

Example:

```
In> LessThan(1,1)
Out> False;
In> LessThan("a","b")
Out> True;
```

Math... — arbitrary-precision math functions

(YACAS internal)

Calling format:

```
MathGcd(n,m)      (Greatest Common Divisor)
MathAdd(x,y)      (add two numbers)
MathSubtract(x,y) (subtract two numbers)
MathMultiply(x,y) (multiply two numbers)
MathDivide(x,y)   (divide two numbers)
MathSqrt(x)    (square root, must be x>=0)
MathFloor(x)   (largest integer not larger than x)
MathCeil(x)    (smallest integer not smaller than x)
MathAbs(x)     (absolute value of x, or |x| )
MathExp(x)     (exponential, base 2.718...)
MathLog(x)     (natural logarithm, for x>0)
MathPower(x,y) (power, x ^ y)
MathSin(x)     (sine)
MathCos(x)     (cosine)
MathTan(x)     (tangent)
MathSinh(x)     (hyperbolic sine)
MathCosh(x)     (hyperbolic cosine)
MathTanh(x)     (hyperbolic tangent)
MathArcSin(x)   (inverse sine)
MathArcCos(x)   (inverse cosine)
MathArcTan(x)   (inverse tangent)
MathArcSinh(x)  (inverse hyperbolic sine)
MathArcCosh(x)  (inverse hyperbolic cosine)
MathArcTanh(x)  (inverse hyperbolic tangent)
MathDiv(x,y)    (integer division, result is an integer)
MathMod(x,y)    (remainder of division, or x mod y)
```

Description:

These commands perform the calculation of elementary mathematical functions. The arguments *must* be numbers. The reason for the prefix `Math` is that the library needs to define equivalent non-numerical functions for symbolic computations, such as `Exp`, `Sin` and so on.

Note that all functions, such as the `MathPower`, `MathSqrt`, `MathAdd` etc., accept integers as well as floating-point numbers. The resulting values may be integers or floats. If the mathematical result is an exact integer, then the integer is returned. For example, `MathSqrt(25)` returns the integer `5`, and `MathPower(2,3)` returns the integer `8`. In such cases, the integer result is returned even if the calculation requires more digits than set by `Builtin'Precision'Set`. However, when the result is mathematically not an integer, the functions return a floating-point result which is correct only to the current precision.

Example:

```
In> Builtin'Precision'Set(10)
Out> True
In> Sqrt(10)
Out> Sqrt(10)
In> MathSqrt(10)
Out> 3.16227766
In> MathSqrt(490000*2^150)
Out> 264452523040700131966976000
In> MathSqrt(490000*2^150+1)
Out> 0.264452523e26
In> MathPower(2,3)
Out> 8
In> MathPower(2,-3)
Out> 0.125
```

Fast... — double-precision math functions

(YACAS internal)

Calling format:

FastLog(x) (natural logarithm), FastPower(x,y), FastArcSin(x)

Description:

Versions of these functions using the C++ library. These should then at least be faster than the arbitrary precision versions.

ShiftLeft — built-in bitwise shift left operation

ShiftRight — built-in bitwise shift right operation

(YACAS internal)

Calling format:

```
ShiftLeft(expr,bits)
ShiftRight(expr,bits)
```

Description:

Shift bits to the left or to the right.

IsPromptShown — test for the Yacas prompt option

(YACAS internal)

Calling format:

```
IsPromptShown()
```

Description:

Returns `False` if Yacas has been started with the option to suppress the prompt, and `True` otherwise.

GetTime — measure the time taken by an evaluation

(YACAS internal)

Calling format:

```
GetTime(expr)
```

Parameters:

`expr` – any expression

Description:

The function `GetTime(expr)` evaluates the expression `expr` and returns the time needed for the evaluation. The result is returned as a floating-point number of seconds. The value of the expression `expr` is lost.

The result is the "user time" as reported by the OS, not the real ("wall clock") time. Therefore, any CPU-intensive processes running alongside Yacas will not significantly affect the result of `GetTime`.

Example:

```
In> GetTime(Simplify((a*b)/(b*a)))
Out> 0.09;
```

See also: `Time`

Chapter 6

Generic objects

Generic objects are objects that are implemented in C++, but can be accessed through the Yacas interpreter.

IsGeneric — check for generic object

(YACAS internal)

Calling format:

```
IsGeneric(object)
```

Description:

Returns `True` if an object is of a generic object type.

GenericTypeName — get type name

(YACAS internal)

Calling format:

```
GenericTypeName(object)
```

Description:

Returns a string representation of the name of a generic object. EG

```
In> GenericTypeName(Array'Create(10,1))
Out> "Array";
```

Array'Create — create array

(YACAS internal)

Calling format:

```
Array'Create(size,init)
```

Description:

Creates an array with `size` elements, all initialized to the value `init`.

Array'Size — get array size

(YACAS internal)

Calling format:

```
Array'Size(array)
```

Description:

Returns the size of an array (number of elements in the array).

Array'Get — fetch array element

(YACAS internal)

Calling format:

```
Array'Get(array,index)
```

Description:

Returns the element at position index in the array passed. Arrays are treated as base-one, so `index` set to 1 would return the first element.

Arrays can also be accessed through the `[]` operators. So `array[index]` would return the same as `Array'Get(array, index)`.

Array'Set — set array element

(YACAS internal)

Calling format:

```
Array'Set(array,index,element)
```

Description:

Sets the element at position index in the array passed to the value passed in as argument to element. Arrays are treated as base-one, so `index` set to 1 would set first element.

Arrays can also be accessed through the `[]` operators. So `array[index] := element` would do the same as `Array'Set(array, index,element)`.

Array'CreateFromList — convert list to array

(YACAS internal)

Calling format:

```
Array'CreateFromList(list)
```

Description:

Creates an array from the contents of the list passed in.

`Array'ToList` — convert array to list

(YACAS internal)

Calling format:

```
Array'ToList(array)
```

Description:

Creates a list from the contents of the array passed in.

Chapter 7

The Yacas test suite

This chapter describes commands used for verifying correct performance of Yacas.

Yacas comes with a test suite which can be found in the directory `tests/`. Typing

```
make test
```

on the command line after Yacas was built will run the test. This test can be run even before `make install`, as it only uses files in the local directory of the Yacas source tree. The default extension for test scripts is `.yts` (Yacas test script).

The verification commands described in this chapter only display the expressions that do not evaluate correctly. Errors do not terminate the execution of the Yacas script that uses these testing commands, since they are meant to be used in test scripts.

`Verify` — verifying equivalence of two expressions

`TestYacas` — verifying equivalence of two expressions

`LogicVerify` — verifying equivalence of two expressions

`LogicTest` — verifying equivalence of two expressions

(standard library)

Calling format:

```
Verify(question,answer)
TestYacas(question,answer)
LogicVerify(question,answer)
LogicTest(variables,expr1,expr2)
```

Parameters:

`question` – expression to check for
`answer` – expected result after evaluation
`variables` – list of variables
`exprN` – Some boolean expression

Description:

The commands `Verify`, `TestYacas`, `LogicVerify` and `LogicTest` can be used to verify that an expression is *equivalent* to a correct answer after evaluation. All three commands return `True` or `False`.

For some calculations, the demand that two expressions are *identical* syntactically is too stringent. The Yacas system might change at various places in the future, but $1 + x$ would still be equivalent, from a mathematical point of view, to $x + 1$.

The general problem of deciding that two expressions a and b are equivalent, which is the same as saying that $a - b = 0$, is generally hard to decide on. The following commands solve this problem by having domain-specific comparisons.

The comparison commands do the following comparison types:

- `Verify` – verify for literal equality. This is the fastest and simplest comparison, and can be used, for example, to test that an expression evaluates to 2.
- `TestYacas` – compare two expressions after simplification as multivariate polynomials. If the two arguments are equivalent multivariate polynomials, this test succeeds. `TestYacas` uses `Simplify`. Note: `TestYacas` currently should not be used to test equality of lists.
- `LogicVerify` – Perform a test by using `CanProve` to verify that from `question` the expression `answer` follows. This test command is used for testing the logic theorem prover in Yacas.
- `LogicTest` – Generate a truth table for the two expressions and compare these two tables. They should be the same if the two expressions are logically the same.

Examples:

```
In> Verify(1+2,3)
Out> True;
In> Verify(x*(1+x),x^2+x)
******************
x*(x+1) evaluates to x*(x+1) which differs
  from x^2+x
******************
Out> False;
In> TestYacas(x*(1+x),x^2+x)
Out> True;
In> Verify(a And c Or b And Not c,a Or b)
******************
 a And c Or b And Not c evaluates to  a And c
  Or b And Not c which differs from  a Or b
******************
Out> False;
In> LogicVerify(a And c Or b And Not c,a Or b)
Out> True;
```

```
In> LogicVerify(a And c Or b And Not c,b Or a)
Out> True;
In> LogicTest({A,B,C},Not((Not A) And (Not B)),A Or B)
Out> True
In> LogicTest({A,B,C},Not((Not A) And (Not B)),A Or C)
******************
CommandLine: 1

$TrueFalse4({A,B,C},Not(Not A And Not B))
 evaluates to
{{{False,False},{True,True}},{{True,True},{True,True}}}
 which differs from
{{{False,True},{False,True}},{{True,True},{True,True}}}
******************
Out> False
```

See also: `Simplify`, `CanProve`, `KnownFailure`

`KnownFailure` — Mark a test as a known failure

(standard library)

Calling format:

```
KnownFailure(test)
```

Parameters:

`test` – expression that should return `False` on failure

Description:

The command `KnownFailure` marks a test as known to fail by displaying a message to that effect on screen.

This might be used by developers when they have no time to fix the defect, but do not wish to alarm users who download Yacas and type `make test`.

Examples:

```
In> KnownFailure(Verify(1,2))
Known failure:
******************
 1 evaluates to  1 which differs from  2
******************
Out> False;
In> KnownFailure(Verify(1,1))
Known failure:
Failure resolved!
Out> True;
```

See also: `Verify`, `TestYacas`, `LogicVerify`

`RoundTo` — Round a real-valued result to a set number of digits

(standard library)

Calling format:

```
RoundTo(number,precision)
```

Parameters:

`number` – number to round off

`precision` – precision to use for round-off

Description:

The function `RoundTo` rounds a floating point number to a specified precision, allowing for testing for correctness using the `Verify` command.

Examples:

```
In> N(RoundTo(Exp(1),30),30)
Out> 2.71828182110230114951959786552;
In> N(RoundTo(Exp(1),20),20)
Out> 2.71828182796964237096;
```

See also: `Verify`, `VerifyArithmetic`, `VerifyDiv`

`VerifyArithmetic` — Special purpose arithmetic verifiers

`RandVerifyArithmetic` — Special purpose arithmetic verifiers

`VerifyDiv` — Special purpose arithmetic verifiers

(standard library)

Calling format:

```
VerifyArithmetic(x,n,m)
RandVerifyArithmetic(n)
VerifyDiv(u,v)
```

Parameters:

`x`, `n`, `m`, `u`, `v` – integer arguments

Description:

The commands `VerifyArithmetic` and `VerifyDiv` test a mathematic equality which should hold, testing that the result returned by the system is mathematically correct according to a mathematically provable theorem.

`VerifyArithmetic` verifies for an arbitrary set of numbers x, n and m that

$$(x^n - 1)(x^m - 1) = x^{n+m} - x^n - x^m + 1.$$

The left and right side represent two ways to arrive at the same result, and so an arithmetic module actually doing the calculation does the calculation in two different ways. The results should be exactly equal.

`RandVerifyArithmetic(n)` calls `VerifyArithmetic` with random values, `n` times.

`VerifyDiv(u,v)` checks that

$$u = v\mathrm{Div}(u,v) + u \bmod v.$$

Examples:

```
In> VerifyArithmetic(100,50,60)
Out> True;
In> RandVerifyArithmetic(4)
Out> True;
In> VerifyDiv(x^2+2*x+3,x+1)
Out> True;
In> VerifyDiv(3,2)
Out> True;
```

See also: `Verify`

Chapter 8

Glossary

This is a short glossary of terms frequently used in the YACAS documentation.

arity

"Arity" is the number of arguments of a function. For example, the function `Cos(x)` has one argument and so we say that "`Cos` has arity 1". Arity of a function can be 0, 1, 2, ...

YACAS allows to define functions with the same name but different arities, and different rules corresponding to these arities will be used. Also, it is possible to define a function with optional arguments, for example, `Plot2D` is one such function. Such functions can have any arity larger or equal to a certain minimum arity (the number of non-optional arguments).

See also: `Function`, `OpPrecedence`, `Rule`

array

An array is a container object that can hold a fixed number of other Yacas objects in it. Individual elements of an array can be accessed using the `[]` operation. Most list operations also work on arrays.

Arrays are faster than lists but the array size cannot be changed.

See also: `Array'Create`

atoms

Atoms are basic Yacas objects that are used to represent symbols, numbers, and function names. An atom has a string representation which is shown when it is displayed. For example, `3.14159`, `x`, `A123`, `+`, `"good morning"` are atoms.

Atoms can be of type string, number, or symbol. For example, `y1` is a symbolic atom, `954` is a number atom, and `""` is an (empty) string atom. Symbolic atoms are normally used in YACAS to denote mathematical unknowns and function names. Number and string atoms are used to denote values.

A symbolic atom can be bound to a value (in which case it becomes a variable), or to a rule or several rules (in which case it becomes a function). An atom can also have a property object.

See also: `Atom`, `String`

CAS

Abbreviation for "computer algebra system". YACAS is a CAS.

constants

Constants such as `I`, `Pi` or `GoldenRatio` are symbolic atoms that are specially interpreted by YACAS. For example, there are simplification rules that transform expressions such as `Sin(Pi)` into `0`. When requesting a numerical evaluation of a constant, the numerical value is given to the current value as set with `N`.

Some constants take a long time to compute and therefore they are cached at the highest precision computed so far. These are the "cached constants".

See also: `N`, `CachedConstant`, `Pi`, `GoldenRatio`, `CatalanConstant`, `gamma`, `I`

equations

To denote symbolic equations, the operator `==` is used. This operator does not assign or compare its sides. For example, the expression `Sin(x)==1` is kept unevaluated and can be passed as argument to functions. For example,

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

The symbolic equation operator `==` is also useful to represent solutions of equations or to specify substitutions, give options, and so on.

See also: `Solve`, `Where`, `Plot2D`

functions

A function is a symbolic atom that is bound to a rule or several rules. A function can have none, one, or more arguments. Functions can also have a variable number of arguments. Arguments of functions are arbitrary Yacas objects.

Functions can be evaluated, that is, the rules bound to them may be executed. For example, `Cos(Pi+0)` is an expression that contains two functions and four atoms. The atom `Pi` is a symbolic atom which is normally not bound to anything. The atom `0` is a numeric atom.

The atoms `Cos` and `+` are symbolic atoms which are bound to appropriate simplification rules. So these two atoms are functions. Note that these functions have different syntax. `Cos` is a normal function which takes its arguments in parentheses. The atom `+` is a function with special syntax because "`+`" is placed between its arguments and no parentheses are used.

The rules to which `+` is bound are such that the expression `Pi+0` is evaluated to the symbolic atom `Pi`. The rules for `Cos` are such that the expression `Cos(Pi)` is evaluated to the numeric atom `-1`. The example YACAS session is:

```
In> Cos(Pi+0)
Out> -1;
```

Some functions are built-in and implemented in C++, while others are library functions.

The built-in functions are usually less flexible than the library functions because they cannot be left unevaluated. Given incorrect arguments, a built-in function will generate an error. However, a user-defined function may simply return unevaluated in such cases.

See also: `Function`, `Rule`, `<--`

lists

A list is a basic YACAS container object. A list is written as e.g. `{a, b, c}` or {} (empty list). Individual elements of a list can be accessed using the `[]` operation. Lists can be concatenated, and individual elements can be removed or inserted.

Lists are ubiquitous in YACAS. Most data structures in the standard library is based on lists.

Lists are also used internally to represent YACAS expressions. For example, the expression `Cos(x+1)` is represented internally as a nested list:

```
In> FullForm( Cos(x+1) )
(Cos
    (+ x 1 ))
Out> Cos(x+1);
```

See also: `List`, `Listify`, `UnList`, `Length`, `FullForm`

matrices

A matrix is represented as a list of lists. Matrices are represented in the "row-major" order: a matrix is a list of rows, and each row is a list of its elements.

Some basic linear algebra operations on matrices are supported.

See also: `Determinant`, `Identity`, `IsDiagonal`, `EigenValues`

operators

Operators are functions that have special syntax declared for them. An operator can be "bodied", infix, prefix or postfix. Because of this, operators must have precedence.

Apart from the syntax, operators are exactly the same as any other functions, they can have rules bound to them in the same way.

See also: `Bodied`, `Infix`, `Prefix`, `Postfix`

plotting

Plotting is currently supported via the `Plot2D` and `Plot3DS` functions. Functions of one or two variables can be plotted on given intervals with a given precision. YACAS generates all numerical data for plots.

See also: `Plot2D`, `Plot3DS`

precedence

Precedence is a property of the syntax of an operator that specifies how it is parsed. Only operators, i.e. functions with special syntax, can have precedence. Precedence values are nonnegative integers: 0, 1, ... Lower numbers bind more tightly.

For example, the operator "`+`" binds less tightly (i.e. has a *higher* precedence value) than the operator "`*`" and so the expression `a+b*c` is parsed as `a+(b*c)`, as one would expect.

Infix operators can have different left-side and right-side precedence. For example, the infix operator "`-`" has left precedence 70 and right precedence 40 – this allows us to parse expressions such as `a-b+c` correctly, as $a-b+c$, and not as $a-(b+c)$.

See also: `Bodied`, `OpPrecedence`, `OpLeftPrecedence`, `OpRightPrecedence`

properties

Properties are special additional objects (tags) that can be tied to expressions. For example, the expression `1+x` may be tagged by an expression `y` by the command

```
In> a:= ExtraInfo'Set(1+x,y);
Out> 1+x;
```

Now `a` refers to an expression `1+x` which is different from all other copies of `1+x` because it is tagged by `y`.

See also: `ExtraInfo'Get`, `ExtraInfo'Set`

rules

Rules are the principal mechanism of expression evaluation in YACAS. A rule specifies that a certain symbolic expression is to be replaced by another expression. If no rule that matches a given symbolic expression can be found, the expression is left unevaluated. This is usually the desired behavior for a CAS. For example, a user can type

```
In> func1(x+0)
Out> func1(x);
```

and use an undefined function `func1`. Since no rules are defined for the function `func1`, it is not evaluated, but its argument has been simplified.

Only expressions containing functions can be evaluated by rules. (Atoms are evaluated only if they are bound to a value.)

Several rules can be defined for a given function. Rules can be erased or added at run time.

See also: `Rule`, `<--`, `Retract`

strings

A string is an atom with character string value, for example, `"abcd"`. Individual characters of the string can be accessed using the `[]` operation. Some string manipulation functions are supported.

See also: `String`, `StringMid'Get`, `StringMid'Set`

syntax

YACAS uses an infix syntax similar to C or Fortran. However, the syntax is entirely user-defined and very flexible. Infix, prefix, postfix operators can be defined, as well as "bodied" functions. This allows to write mathematical expressions more comfortably, for example,

```
In> D(x) Sin(x)+1
Out> Cos(x);
```

Functions with special syntax can have different precedence.

See also: `Bodied`, `Infix`, `Prefix`, `Postfix`, `OpPrecedence`

variables

Variables are symbolic atoms bound to a "value". Value is any Yacas object, such as an atom or a list. For example, after executing

```
In> a := 1
Out> 1;
```

the symbolic atom `a` becomes a variable bound to a value, the numeric atom `1`.

See also: `Eval`, `:=`, `Clear`

warranty

YACAS is Free Software ("logiciel libre") and comes with `NO WARRANTY`. See *the appropriate section of the GPL*, in *The Yacas Programmer's Function Reference, Chapter 1, Section 2* for full information.

Chapter 9

GNU General Public License

This chapter contains the GNU General Public License (GPL). This information is important for everyone using or modifying the Yacas source code or binaries.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

9.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands inter-

actively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program 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.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the

terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

9.2 NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Chapter 10

GNU Free Documentation License

Version 1.1, March 2000

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

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. 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