Programming in Yacas

by the Yacas team [1]

Yacas version: 1.3.6
generated on May 12, 2017

This document should get you started programming in Yacas. There are some basic explanations and hands-on tutorials.

[1]This text is part of the Yacas software package. Copyright 2000–2002. Principal documentation authors: Ayal Zwi Pinkus, Serge Winitzki, Jitse Niesen. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Contents

1 Yacas under the hood **3**
- 1.1 The Yacas architecture . . . 3
- 1.2 Startup, scripts and `.def` files . . . 3
- 1.3 Object types . . . 3
- 1.4 Yacas evaluation scheme . . . 3
- 1.5 Rules . . . 4
- 1.6 Examples of using rules . . . 5
- 1.7 Structured programming and control flow . . . 5
- 1.8 Additional syntactic sugar . . . 6
- 1.9 Using "Macro rules" (e.g. `NFunction`) . . . 6
- 1.10 Macro expansion . . . 7
- 1.11 Scope of variable bindings . . . 7

2 Evaluation of expressions **8**
- 2.1 The LISP heritage . . . 8
- 2.2 YACAS-specific extensions for CAS implementations . . . 9
- 2.3 Destructive operations . . . 10

3 Coding style **12**
- 3.1 Introduction . . . 12
- 3.2 Interactions of rules and types . . . 12
- 3.3 Ordering of rules . . . 12
- 3.4 Writing new library functions . . . 12
- 3.5 Reporting errors . . . 13

4 Advanced example 1: parsing expressions (`CForm`) **15**
- 4.1 Recursive parsing of expression trees . . . 15
- 4.2 Handling precedence of infix operations . . . 15

5 Yacas programming pitfalls **17**
- 5.1 All rules are global . . . 17
- 5.2 Objects that look like functions . . . 17
- 5.3 Guessing when arguments are evaluated and when not . . . 18
- 5.4 Evaluating variables in the wrong scope . . . 19

6 Debugging in Yacas **20**
- 6.1 Introduction . . . 20
- 6.2 The trace facilities . . . 20

7 Custom evaluation facilities **21**
- 7.1 The basic infrastructure for custom evaluation . . . 21
- 7.2 A simple interactive debugger . . . 22
- 7.3 Profiling . . . 23
- 7.4 The Yacas Debugger . . . 23

8 Advanced example 2: implementing a non-commutative algebra **24**
8.1 The problem . . . 24
8.2 First steps . . . 24
8.3 Structure of expressions . . . 24
8.4 Implementing the canonical form . . . 25
8.5 Implementing commutation relations . . . 26
8.6 Avoiding infinite recursion . . . 26
8.7 Implementing VEV() . . . 26

9 GNU Free Documentation License **28**

Chapter 1

Yacas under the hood

This part of the manual is a somewhat in-depth explanation of the Yacas programming language and environment. It assumes that you have worked through the introductory tutorial. You should consult the function reference about how to use the various Yacas functions mentioned here.

1.1 The Yacas architecture

Yacas is designed as a small core engine that interprets a library of scripts. The core engine provides the syntax parser and a number of hard-wired functions, such as `Set()` or `MathExp()` which cannot be redefined by the user. The script library resides in the scripts directory "`scripts/`" and contains higher-level definitions of functions and constants. The library scripts are on equal footing with any code the user executes interactively or any files the user loads.

Generally, all core functions have plain names and almost all are not "bodied" or infix operators. The file `corefunctions.h` in the source tree lists declarations of all kernel functions callable from Yacas; consult it for reference. For many of the core functions, the script library already provides convenient aliases. For instance, the addition operator "`+`" is defined in the script `scripts/standard` while the actual addition of numbers is performed through the built-in function `MathAdd`.

1.2 Startup, scripts and `.def` files

When Yacas is first started or restarted, it executes the script `yacasinit.ys` in the scripts directory. This script may load some other scripts. In order to start up quickly, Yacas does not execute all other library scripts at first run or at restart. It only executes the file `yacasinit.ys` and all `.def` files in the scripts. The `.def` files tell the system where it can find definitions for various library functions. Library is divided into "packages" stored in "repository" directories. For example, the function `ArcTan` is defined in the `stdfuncs` package; the library file is `stdfuncs.rep/code.ys` and the `.def` file is `stdfuncs.rep/code.ys.def`. The function `ArcTan` mentioned in the `.def` file, therefore Yacas will know to load the package `stdfuncs` when the user invokes `ArcTan`. This way Yacas knows where to look for any given function without actually loading the file where the function is defined.

There is one exception to the strategy of delayed loading of the library scripts. Namely, the syntax definitions of infix, prefix, postfix and bodied functions, such as `Infix("*",4)` cannot be delayed (it is currently in the file `stdopers.ys`). If it were delayed, the Yacas parser would encounter `1+2` (typed by the user) and generate a syntax error before it has a chance to load the definition of the operator "`+`".

1.3 Object types

Yacas supports two basic kinds of objects: atoms and compounds. Atoms are (integer or real, arbitrary-precision) numbers such as `2.71828`, symbolic variables such as `A3` and character strings. Compounds include functions and expressions, e.g. `Cos(a-b)` and lists, e.g. {`1+a,2+b,3+c`}.

The type of an object is returned by the built-in function `Type`, for example:

```
In> Type(a);
Out> "";
In> Type(F(x));
Out> "F";
In> Type(x+y);
Out> "+";
In> Type({1,2,3});
Out> "List";
```

Internally, atoms are stored as strings and compounds as lists. (The Yacas lexical analyzer is case-sensitive, so `List` and `list` are different atoms.) The functions `String()` and `Atom()` convert between atoms and strings. A Yacas list {`1,2,3`} is internally a list `(List 1 2 3)` which is the same as a function call `List(1,2,3)` and for this reason the "type" of a list is the string `"List"`. During evaluation, atoms can be interpreted as numbers, or as variables that may be bound to some value, while compounds are interpreted as function calls.

Note that atoms that result from an `Atom()` call may be invalid and never evaluate to anything. For example, `Atom(3X)` is an atom with string representation "3X" but with no other properties.

Currently, no other lowest-level objects are provided by the core engine besides numbers, atoms, strings, and lists. There is, however, a possibility to link some externally compiled code that will provide additional types of objects. Those will be available in Yacas as "generic objects." For example, fixed-size arrays are implemented in this way.

1.4 Yacas evaluation scheme

Evaluation of an object is performed either explicitly by the built-in command `Eval()` or implicitly when assigning variables or calling functions with the object as argument (except when a function does not evaluate that argument). Evaluation of an object can be explicitly inhibited using `Hold()`. To make a function not evaluate one of its arguments, a `HoldArg(funcname, argname)` must be declared for that function.

Internally, all expressions are either atoms or lists (perhaps nested). Use `FullForm()` to see the internal form of an expression. A Yacas list expression written as {`a, b`} is repre-

sented internally as `(List a b)`, equivalently to a function call `List(a,b)`.

Evaluation of an atom goes as follows: if the atom is bound locally as a variable, the object it is bound to is returned, otherwise, if it is bound as a global variable then that is returned. Otherwise, the atom is returned unevaluated. Note that if an atom is bound to an expression, that expression is considered as final and is not evaluated again.

Internal lists of atoms are generally interpreted in the following way: the first atom of the list is some command, and the atoms following in the list are considered the arguments. The engine first tries to find out if it is a built-in command (core function). In that case, the function is executed. Otherwise, it could be a user-defined function (with a "rule database"), and in that case the rules from the database are applied to it. If none of the rules are applicable, or if no rules are defined for it, the object is returned unevaluated.

Application of a rule to an expression transforms it into a different expression to which other rules may be applicable. Transformation by matching rules continues until no more rules are applicable, or until a "terminating" rule is encountered. A "terminating" rule is one that returns `Hold()` or `UnList()` of some expression. Calling these functions gives an unevaluated expression because it terminates the process of evaluation itself.

The main properties of this scheme are the following. When objects are assigned to variables, they generally are evaluated (except if you are using the `Hold()` function) because assignment `var := value` is really a function call to `Set(var, value)` and this function evaluates its second argument (but not its first argument). When referencing that variable again, the object which is its value will not be re-evaluated. Also, the default behavior of the engine is to return the original expression if it could not be evaluated. This is a desired behavior if evaluation is used for simplifying expressions.

One major design flaw in Yacas (one that other functional languages like LISP also have) is that when some expression is re-evaluated in another environment, the local variables contained in the expression to be evaluated might have a different meaning. In this case it might be useful to use the functions `LocalSymbols` and `TemplateFunction`. Calling

```
LocalSymbols(a,b)
a*b;
```

results in "a" and "b" in the multiplication being substituted with unique symbols that can not clash with other variables that may be used elsewhere. Use `TemplateFunction` instead of `Function` to define a function whose parameters should be treated as unique symbols.

Consider the following example:

```
In> f1(x):=Apply("+",{x,x});
Out> True
```

The function `f1` simply adds its argument to itself. Now calling this function with some argument:

```
In> f1(Sin(a))
Out> 2*Sin(a)
```

yields the expected result. However, if we pass as an argument an expression containing the variable `x`, things go wrong:

```
In> f1(Sin(x))
Out> 2*Sin(Sin(x))
```

This happens because within the function, `x` is bound to `Sin(x)`, and since it is passed as an argument to `Apply` it will be re-evaluated, resulting in `Sin(Sin(x))`. `TemplateFunction` solves this by making sure the arguments can not collide like this (by using `LocalSymbols`:

```
In> TemplateFunction("f2",{x}) Apply("+",{x,x});
Out> True
In> f2(Sin(a))
Out> 2*Sin(a)
In> f2(Sin(x))
Out> 2*Sin(x)
```

In general one has to be careful when functions like `Apply`, `Map` or `Eval` (or derivatives) are used.

1.5 Rules

Rules are special properties of functions that are applied when the function object is being evaluated. A function object could have just one rule bound to it; this is similar to a "subroutine" having a "function body" in usual procedural languages. However, Yacas function objects can also have several rules bound to them. This is analogous of having several alternative "function bodies" that are executed under different circumstances. This design is more suitable for symbolic manipulations.

A function is identified by its name as returned by `Type` and the number of arguments, or "arity". The same name can be used with different arities to define different functions: `f(x)` is said to "have arity 1" and `f(x,y)` has arity 2. Each of these functions may possess its own set of specific rules, which we shall call a "rule database" of a function.

Each function should be first declared with the built-in command `RuleBase` as follows:

```
RuleBase("FunctionName",{argument list});
```

So, a new (and empty) rule database for `f(x,y)` could be created by typing `RuleBase("f",{x,y})`. The names for the arguments "x" and "y" here are arbitrary, but they will be globally stored and must be later used in descriptions of particular rules for the function `f`. After the new rulebase declaration, the evaluation engine of Yacas will begin to really recognize `f` as a function, even though no function body or equivalently no rules have been defined for it yet.

The shorthand operator `:=` for creating user functions that we illustrated in the tutorial is actually defined in the scripts and it makes the requisite call to the `RuleBase()` function. After a `RuleBase()` call you can specify parsing properties for the function; for example, you could make it an infix or bodied operator.

Now we can add some rules to the rule database for a function. A rule simply states that if a specific function object with a specific arity is encountered in an expression and if a certain predicate is true, then Yacas should replace this function with some other expression. To tell Yacas about a new rule you can use the built-in `Rule` command. This command is what does the real work for the somewhat more aesthetically pleasing `... # ... <-- ...` construct we have seen in the tutorial. You do not have to call `RuleBase()` explicitly if you use that construct.

Here is the general syntax for a `Rule()` call:

```
Rule("foo", arity, precedence, pred) body;
```

This specifies that for function `foo` with given `arity` (`foo(a,b)` has arity 2), there is a rule that if `pred` is true, then `body` should be evaluated, and the original expression replaced by the result. Predicate and body can use the symbolic names of arguments that were declared in the `RuleBase` call.

All rules for a given function can be erased with a call to `Retract(funcname, arity)`. This is useful, for instance, when too many rules have been entered in the interactive mode. This call undefines the function and also invalidates the `RuleBase` declaration.

You can specify that function arguments are not evaluated before they are bound to the parameter: `HoldArg("foo",a)` would then declare that the a arguments in both `foo(a)` and `foo(a,b)` should not be evaluated before bound to `a`. Here the argument name `a` should be the same as that used in the `RuleBase()` call when declaring these functions. Inhibiting evaluation of certain arguments is useful for procedures performing actions based partly on a variable in the expression, such as integration, differentiation, looping, etc., and will be typically used for functions that are algorithmic and procedural by nature.

Rule-based programming normally makes heavy use of recursion and it is important to control the order in which replacement rules are to be applied. For this purpose, each rule is given a *precedence*. Precedences go from low to high, so all rules with precedence 0 will be tried before any rule with precedence 1.

You can assign several rules to one and the same function, as long as some of the predicates differ. If none of the predicates are true, the function is returned with its arguments evaluated.

This scheme is slightly slower for ordinary functions that just have one rule (with the predicate `True`), but it is a desired behavior for symbolic manipulation. You can gradually build up your own functions, incrementally testing their properties.

1.6 Examples of using rules

As a simple illustration, here are the actual `RuleBase()` and `Rule()` calls needed to define the factorial function:

```
In> RuleBase("f",{n});
Out> True;
In> Rule("f", 1, 10, n=0) 1;
Out> True;
In> Rule("f", 1, 20, IsInteger(n) \
  And n>0) n*f(n-1);
Out> True;
```

This definition is entirely equivalent to the one in the tutorial. `f(4)` should now return 24, while `f(a)` should return just `f(a)` if `a` is not bound to any value.

The `Rule` commands in this example specified two rules for function `f` with arity 1: one rule with precedence 10 and predicate `n=0`, and another with precedence 20 and the predicate that returns `True` only if `n` is a positive integer. Rules with lowest precedence get evaluated first, so the rule with precedence 10 will be tried before the rule with precedence 20. Note that the predicates and the body use the name "n" declared by the `RuleBase()` call.

After declaring `RuleBase()` for a function, you could tell the parser to treat this function as a postfix operator:

```
In> Postfix("f");
Out> True;
In> 4 f;
Out> 24;
```

There is already a function `Function` defined in the standard scripts that allows you to construct simple functions. An example would be

```
Function ("FirstOf", {list})  list[1] ;
```

which simply returns the first element of a list. This could also have been written as

```
Function("FirstOf", {list})
[
  list[1] ;
];
```

As mentioned before, the brackets `[ ]` are also used to combine multiple operations to be performed one after the other. The result of the last performed action is returned.

Finally, the function `FirstOf` could also have been defined by typing

```
FirstOf(list):=list[1] ;
```

1.7 Structured programming and control flow

Some functions useful for control flow are already defined in Yacas's standard library. Let's look at a possible definition of a looping function `ForEach`. We shall here consider a somewhat simple-minded definition, while the actual `ForEach` as defined in the standard script "controlflow" is a little more sophisticated.

```
Function("ForEach",{foreachitem,
  foreachlist,foreachbody})
[
   Local(foreachi,foreachlen);
   foreachlen:=Length(foreachlist);
   foreachi:=0;
   While (foreachi < foreachlen)
   [
     foreachi++;
     MacroLocal(foreachitem);
     MacroSet(foreachitem,
   foreachlist[foreachi]);
     Eval(foreachbody);
   ];
];

Bodied("ForEach");
UnFence("ForEach",3);
HoldArg("ForEach",foreachitem);
HoldArg("ForEach",foreachbody);
```

Functions like this should probably be defined in a separate file. You can load such a file with the command `Load("file")`. This is an example of a macro-like function. Let's first look at the last few lines. There is a `Bodied(...)` call, which states that the syntax for the function `ForEach()` is `ForEach(item,{list}) body;` – that is, the last argument to the command `ForEach` should be outside its brackets. `UnFence(...)` states that this function can use the local variables of the calling function. This is necessary, since the body to be evaluated for each item will probably use some local variables from that surrounding.

Finally, `HoldArg("function",argument)` specifies that the argument "`argument`" should not be evaluated before being bound to that variable. This holds for `foreachitem` and `foreachbody`, since `foreachitem` specifies a variable to be set to that value, and `foreachbody` is the expression that should be evaluated *after* that variable is set.

Inside the body of the function definition there are calls to `Local(...)`. `Local()` declares some local variable that will only be visible within a block `[ ... ]`. The command `MacroLocal()` works almost the same. The difference is that it evaluates its arguments before performing the action on it. This is needed in this case, because the variable `foreachitem` is bound to a variable to be used as the loop iterator, and it is *the variable it is bound to* that we want to make local, not `foreachitem` itself. `MacroSet()` works similarly: it does the same as `Set()` except that it also first evaluates the first argument, thus setting the variable requested by the user of this

function. The `Macro...` functions in the built-in functions generally perform the same action as their non-macro versions, apart from evaluating an argument it would otherwise not evaluate.

To see the function in action, you could type:

```
ForEach(i,{1,2,3}) [Write(i); NewLine();];
```

This should print 1, 2 and 3, each on a new line.

Note: the variable names "foreach..." have been chosen so they won't get confused with normal variables you use. This is a major design flaw in this language. Suppose there was a local variable `foreachitem`, defined in the calling function, and used in `foreachbody`. These two would collide, and the interpreter would use only the last defined version. In general, when writing a function that calls `Eval()`, it is a good idea to use variable names that can not collide with user's variables. This is generally the single largest cause of bugs when writing programs in Yacas. This issue should be addressed in the future.

1.8 Additional syntactic sugar

The parser is extended slightly to allow for fancier constructs.

- Lists, e.g. `{a,b}`. This then is parsed into the internal notation `(List a b)` , but will be printed again as `{a,b};`
- Statement blocks such as `[ statement1 ; statement2;];`. This is parsed into a Lisp object `(Prog (statement1 ) (statement2 ))`, and printed out again in the proper form.
- Object argument accessors in the form of `expr[ index ]`. These are mapped internally to `Nth(expr,index)`. The value of `index`=0 returns the operator of the object, `index`=1 the first argument, etc. So, if `expr` is `foo(bar)`, then `expr[0]` returns `foo`, and `expr[1]` returns `bar`. Since lists of the form `{...}` are essentially the same as `List(...)`, the same accessors can be used on lists.
- Function blocks such as

  ```
  While (i < 10)
    [
  Write(i);
  i:=i+1;
    ];
  ```

  The expression directly following the `While(...)` block is added as a last argument to the `While(...)` call. So `While(a)b;` is parsed to the internal form `(While a b)`.

This scheme allows coding the algorithms in an almost C-like syntax.

Strings are generally represented with quotes around them, e.g. "this is a string". Backslash
in a string will unconditionally add the next character to the string, so a quote can be added with
`\"` (a backslash-quote sequence).

1.9 Using "Macro rules" (e.g. `NFunction`)

The Yacas language allows to have rules whose definitions are generated at runtime. In other words, it is possible to write rules (or "functions") that, as a side-effect, will define other rules, and those other rules will depend on some parts of the expression the original function was applied to.

This is accomplished using functions `MacroRuleBase`, `MacroRule`, `MacroRulePattern`. These functions evaluate their arguments (including the rule name, predicate and body) and define the rule that results from this evaluation.

Normal, "non-Macro" calls such as `Rule()` will not evaluate their arguments and this is a desired feature. For example, suppose we defined a new predicate like this,

```
RuleBase("IsIntegerOrString, {x});
Rule("IsIntegerOrString", 1, 1, True)
IsInteger(x) And IsString(x);
```

If the `Rule()` call were to evaluate its arguments, then the "body" argument, `IsInteger(x) And IsString(x)`, would be evaluated to `False` since `x` is an atom, so we would have defined the predicate to be always `False`, which is not at all what we meant to do. For this reason, the `Rule` calls do not evaluate their arguments.

Consider however the following situation. Suppose we have a function `f(arglist)` where `arglist` is its list of arguments, and suppose we want to define a function `Nf(arglist)` with the same arguments which will evaluate `f(arglist)` and return only when all arguments from `arglist` are numbers, and return unevaluated `Nf(arglist)` otherwise. This can of course be done by a usual rule such as

```
Rule("Nf", 3, 0, IsNumericList({x,y,z}))
  <-- "f" @ {x,y,z};
```

Here `IsNumericList` is a predicate that checks whether all elements of a given list are numbers. (We deliberately used a `Rule` call instead of an easier-to-read `<--` operator to make it easier to compare with what follows.)

However, this will have to be done for every function `f` separately. We would like to define a procedure that will define `Nf`, given *any* function `f`. We would like to use it like this:

```
NFunction("Nf", "f", {x,y,z});
```

After this function call we expect to be able to use the function `Nf`.

Here is how we could naively try to implement `NFunction` (and fail):

```
NFunction(new'name, old'name, arg'list) := [
  MacroRuleBase(new'name, arg'list);
  MacroRule(new'name, Length(arg'list), 0,
    IsNumericList(arg'list)
    )
  new'name @ arg'list;
];
```

Now, this just does not do anything remotely right. `MacroRule` evaluates its arguments. Since `arg'list` is an atom and not a list of numbers at the time we are defining this, `IsNumericList(arg'list)` will evaluate to `False` and the new rule will be defined with a predicate that is always `False`, i.e. it will be never applied.

The right way to figure this out is to realize that the `MacroRule` call evaluates all its arguments and passes the results to a `Rule` call. So we need to see exactly what `Rule()` call we need to produce and then we need to prepare the arguments of `MacroRule` so that they evaluate to the right values. The `Rule()` call we need is something like this:

```
Rule("actual new name", <actual # of args>, 0,
  IsNumericList({actual arg list})
)  "actual new name" @ {actual arg list};
```

Note that we need to produce expressions such as `"new name" @ arg'list` and not *results* of evaluation of these expressions. We can produce these expressions by using `UnList()`, e.g.

```
UnList({Atom("@"), "Sin", {x}})
```

produces

```
"Sin" @ {x};
```

but not `Sin(x)`, and

```
UnList({IsNumericList, {1,2,x}})
```

produces the expression

```
IsNumericList({1,2,x});
```

which is not further evaluated.

Here is a second version of `NFunction()` that works:

```
NFunction(new'name, old'name, arg'list) := [
  MacroRuleBase(new'name, arg'list);
  MacroRule(new'name, Length(arg'list), 0,
    UnList({IsNumericList, arg'list})
  )
    UnList({Atom("@"), old'name, arg'list});
];
```

Note that we used `Atom("@")` rather than just the bare atom `@` because `@` is a prefix operator and prefix operator names as bare atoms do not parse (they would be confused with applications of a prefix operator to what follows).

Finally, there is a more concise (but less general) way of defining `NFunction()` for functions with known number of arguments, using the backquoting mechanism. The backquote operation will first substitute variables in an expression, without evaluating anything else, and then will evaluate the resulting expression a second time. The code for functions of just one variable may look like this:

```
N1Function(new'name, old'name) :=
`( @new'name(x_IsNumber) <-- @old'name(x) );
```

This executes a little slower than the above version, because the backquote needs to traverse the expression twice, but makes for much more readable code.

1.10 Macro expansion

Yacas supports macro expansion (back-quoting). An expression can be back-quoted by putting a ` in front of it. Within the back-quoted expression, all atoms that have a `@` in front of them get replaced with the value of that atom (treated as a variable), and then the resulting expression is evaluated:

```
In> x:=y
Out> y;
In> `(@x:=2)
Out> 2;
In> x
Out> y;
In> y
Out> 2;
```

This is useful in cases where within an expression one subexpression is not evaluated. For instance, transformation rules can be built dynamically, before being declared. This is a particularly powerful feature that allows a programmer to write programs that write programs. The idea is borrowed from Lisp.

As the above example shows, there are similarities with the `Macro...` functions, that serve the same purpose for specific expressions. For example, for the above code, one could also have called `MacroSet`:

```
In> MacroSet(x,3)
Out> True;
In> x
Out> y;
In> y
Out> 3;
```

The difference is that `MacroSet`, and in general the `Macro...` functions, are faster than their back-quoted counterparts. This is because with back-quoting, first a new expression is built before it is evaluated. The advantages of back-quoting are readability and flexibility (the number of `Macro...` functions is limited, whereas back-quoting can be used anywhere).

When an `@` operator is placed in front of a function call, the function call is replaced:

```
In> plus:=Add
Out> Add;
In> `(@plus(1,2,3))
Out> 6;
```

Application of pure functions is also possible (as of version 1.0.53) by using macro expansion:

```
In> pure:={{a,b},a+b};
Out> {{a,b},a+b};
In> ` @pure(2,3);
Out> 5;
```

Pure (nameless) functions are useful for declaring a temporary function, that has functionality depending on the current environment it is in, or as a way to call driver functions. In the case of drivers (interfaces to specific functionality), a variable can be bound to a function to be evaluated to perform a specific task. That way several drivers can be around, with one bound to the variables holding the functions that will be called.

1.11 Scope of variable bindings

When setting variables or retrieving variable values, variables are automatically bound global by default. You can explicitly specify variables to be local to a block such as a function body; this will make them invisible outside the block. Blocks have the form `[ statement1; statement2; ]` and local variables are declared by the `Local()` function.

When entering a block, a new stack frame is pushed for the local variables; it means that the code inside a block doesn't see the local variables of the *caller* either! You can tell the interpreter that a function should see local variables of the calling environment; to do this, declare

```
UnFence(funcname, arity)
```

on that function.

Chapter 2

Evaluation of expressions

When programming in some language, it helps to have a mental model of what goes on behind the scenes when evaluating expressions, or in this case simplifying expressions.

This section aims to explain how evaluation (and simplification) of expressions works internally, in YACAS.

2.1 The LISP heritage

Representation of expressions

Much of the inner workings is based on how LISP-like languages are built up. When an expression is entered, or composed in some fashion, it is converted into a prefix form much like you get in LISP:

```
a+b    ->    (+ a b)
Sin(a) ->    (Sin a)
```

Here the sub-expression is changed into a list of so-called "atoms", where the first atom is a function name of the function to be invoked, and the atoms following are the arguments to be passed in as parameters to that function.

YACAS has the function `FullForm` to show the internal representation:

```
In> FullForm(a+b)
(+ a b )
Out> a+b;
In> FullForm(Sin(a))
(Sin a )
Out> Sin(a);
In> FullForm(a+b+c)
(+ (+ a b )c )
Out> a+b+c;
```

The internal representation is very close to what `FullForm` shows on screen. `a+b+c` would be `(+ (+ a b )c )` internally, or:

```
()
|
|
+  -> () -> c
       |
       |
       + -> a -> b
```

Evaluation

An expression like described above is done in the following manner: first the arguments are evaluated (if they need to be evaluated, YACAS can be told to not evaluate certain parameters to functions), and only then are these arguments passed in to the function for evaluation. They are passed in by binding local variables to the values, so these arguments are available as local values.

For instance, suppose we are evaluating `2*3+4`. This first gets changed to the internal representation `(+ (* 2 3 )4 )`. Then, during evaluation, the top expression refers to function "`+`". Its arguments are `(* 2 3)` and `4`. First `(* 2 3)` gets evaluated. This is a function call to the function "`*`" with arguments `2` and `3`, which evaluate to themselves. Then the function "`*`" is invoked with these arguments. The YACAS standard script library has code that accepts numeric input and performs the multiplication numerically, resulting in `6`.

The second argument to the top-level "`+`" is `4`, which evaluates to itself.

Now, both arguments to the "`+`" function have been evaluated, and the results are `6` and `4`. Now the "`+`" function is invoked. This function also has code in the script library to actually perform the addition when the arguments are numeric, so the result is 10:

```
In> FullForm(Hold(2*3+4))
(+ (* 2 3 )4 )
Out> 2*3+4;
In> 2*3+4
Out> 10;
```

Note that in YACAS, the script language does not define a "`+`" function in the core. This and other functions are all implemented in the script library. The feature "when the arguments to "`+`" are numeric, perform the numeric addition" is considered to be a "policy" which should be configurable. It should not be a part of the core language.

It is surprisingly difficult to keep in mind that evaluation is bottom up, and that arguments are evaluated before the function call is evaluated. In some sense, you might feel that the evaluation of the arguments is part of evaluation of the function. It is not. Arguments are evaluated before the function gets called.

Suppose we define the function `f`, which adds two numbers, and traces itself, as:

```
In> f(a,b):= \
In> [\
In> Local(result);\
In> Echo("Enter f with arguments ",a,b);\
In> result:=a+b;\
In> Echo("Leave f with result ",result);\
In> result;\
In> ];
Out> True;
```

Then the following interaction shows this principle:

```
In> f(f(2,3),4)
Enter f with arguments 2 3
Leave f with result 5
Enter f with arguments 5 4
Leave f with result 9
Out> 9;
```

The first Enter/Leave combination is for `f(2,3)`, and only then is the outer call to `f` entered.

This has important consequences for the way YACAS simplifies expressions: the expression trees are traversed bottom up, as the lowest parts of the expression trees are simplified first, before being passed along up to the calling function.

2.2 Yacas-specific extensions for CAS implementations

YACAS has a few language features specifically designed for use when implementing a CAS.

The transformation rules

Working with transformation rules is explained in the introduction and tutorial book. This section mainly deals with how YACAS works with transformation rules under the hood.

A transformation rule consists of two parts: a condition that an expression should match, and a result to be substituted for the expression if the condition holds. The most common way to specify a condition is a pattern to be matched to an expression.

A pattern is again simply an expression, stored in internal format:

```
In> FullForm(a_IsInteger+b_IsInteger*(_x))
(+ (_ a IsInteger )(* (_ b IsInteger )(_ x )))
Out> a _IsInteger+b _IsInteger*_x;
```

YACAS maintains structures of transformation rules, and tries to match them to the expression being evaluated. It first tries to match the structure of the pattern to the expression. In the above case, it tries to match to `a+b*x`. If this matches, local variables `a`, `b` and `x` are declared and assigned the sub-trees of the expression being matched. Then the predicates are tried on each of them: in this case, `IsInteger(a)` and `IsInteger(b)` should both return `True`.

Not shown in the above case, are post-predicates. They get evaluated afterwards. This post-predicate must also evaluate to `True`. If the structure of the expression matches the structure of the pattern, and all predicates evaluate to `True`, the pattern matches and the transformation rule is applied, meaning the right hand side is evaluated, with the local variables mentioned in the pattern assigned. This evaluation means all transformation rules are re-applied to the right-hand side of the expression.

Note that the arguments to a function are evaluated first, and only then is the function itself called. So the arguments are evaluated, and then the transformation rules applied on it. The main function defines its parameters also, so these get assigned to local variables also, before trying the patterns with their associated local variables.

Here is an example making the fact that the names in a pattern are local variables more explicit:

```
In> f1(_x,_a) <-- x+a
Out> True;
In> f2(_x,_a) <-- [Local(a); x+a;];
Out> True;
In> f1(1,2)
Out> 3;
In> f2(1,2)
Out> a+1;
```

Using different rules in different cases

In a lot of cases, the algorithm to be invoked depends on the type of the arguments. Or the result depends on the form of the input expression. This results in the typical "case" or "switch" statement, where the code to evaluate to determine the result depends on the form of the input expression, or the type of the arguments, or some other conditions.

YACAS allows to define several transformation rules for one and the same function, if the rules are to be applied under different conditions.

Suppose the function `f` is defined, a factorial function:

```
10 # f(0) <-- 1;
20 # f(n_IsPositiveInteger) <-- n*f(n-1);
```

Then interaction can look like:

```
In> f(3)
Out> 6;
In> f(a)
Out> f(a);
```

If the left hand side is matched by the expression being considered, then the right hand side is evaluated. A subtle but important thing to note is that this means that the whole body of transformation rules is thus re-applied to the right-hand side of the `<--` operator.

Evaluation goes bottom-up, evaluating (simplifying) the lowest parts of a tree first, but for a tree that matches a transformation rule, the substitution essentially means return the result of evaluating the right-hand side. Transformation rules are re-applied, on the right hand side of the transformation rule, and the original expression can be thought of as been substituted by the result of evaluating this right-hand side, which is supposed to be a "simpler" expression, or a result closer to what the user wants.

Internally, the function `f` is built up to resemble the following pseudo-code:

```
f(n)
{
   if (n = 1)
     return 1;
   else if (IsPositiveInteger(n))
     return n*f(n-1);
   else return f(n) unevaluated;
}
```

The transformation rules are thus combined into one big statement that gets executed, with each transformation rule being a if-clause in the statement to be evaluated. Transformation rules can be spread over different files, and combined in functional groups. This adds to the readability. The alternative is to write the full body of each function as one big routine, which becomes harder to maintain as the function becomes larger and larger, and hard or impossible to extend.

One nice feature is that functionality is easy to extend without modifying the original source code:

```
In> Ln(x*y)
Out> Ln(x*y);
In> Ln(_x*_y) <-- Ln(x) + Ln(y)
Out> True;
In> Ln(x*y)
Out> Ln(x)+Ln(y);
```

This is generally not advisable, due to the fact that it alters the behavior of the entire system. But it can be useful in some instances. For instance, when introducing a new function `f(x)`, one can decide to define a derivative explicitly, and a way to simplify it numerically:

```
In> f(_x)_InNumericMode() <-- Exp(x)
Out> True;
In> (Deriv(_x)f(_y)) <-- f(y)*(Deriv(x)y);
Out> True;
In> f(2)
Out> f(2);
In> N(f(2))
Out> 7.3890560989;
In> Exp(2)
Out> Exp(2);
In> N(Exp(2))
Out> 7.3890560989;
In> D(x)f(a*x)
Out> f(a*x)*a;
```

The "Evaluation is Simplification" hack

One of the ideas behind the YACAS scripting language is that evaluation is used for simplifying expressions. One consequence of this is that objects can be returned unevaluated when they can not be simplified further. This happens to variables that are not assigned, functions that are not defined, or function invocations where the arguments passed in as parameters are not actually handled by any code in the scripts. An integral that can not be performed by YACAS should be returned unevaluated:

```
In> 2+3
Out> 5;
In> a+b
Out> a+b;
In> Sin(a)
Out> Sin(a);
In> Sin(0)
Out> 0;
In> Integrate(x)Ln(x)
Out> x*Ln(x)-x;
In> Integrate(x)Ln(Sin(x))
Out> Integrate(x)Ln(Sin(x));
In> a!
Out> a!;
In> 3!
Out> 6;
```

Other languages usually do not allow evaluation of unbound variables, or undefined functions. In YACAS, these are interpreted as some yet undefined global variables or functions, and returned unevaluated.

2.3 Destructive operations

YACAS tries to keep as few copies of objects in memory as possible. Thus when assigning the value of one variable to another, a reference is copied, and both variables refer to the same memory, physically. This is relevant for programming; for example, one should use `FlatCopy` to actually make a new copy of an object. Another feature relevant to reference semantics is "destructive operations"; these are functions that modify their arguments rather than work on a copy. Destructive operations on lists are generally recognized because their name starts with "Destructive", e.g. `DestructiveDelete`. One other destructive operation is assignment of a list element through `list[index] := ...`.

Some examples to illustrate destructive operations on lists:

```
In> x1:={a,b,c}
Out> {a,b,c};
```

A list `x1` is created.

```
In> FullForm(x1)
(List a b c )
Out> {a,b,c};
In> x2:=z:x1
Out> {z,a,b,c};
```

A new list `x2` is `z` appended to `x1`. The `:` operation creates a copy of `x1` before appending, so `x1` is unchanged by this.

```
In> FullForm(x2)
(List z a b c )
Out> {z,a,b,c};
In> x2[1]:=y
Out> True;
```

We have modified the first element of `x2`, but `x1` is still the same.

```
In> x2
Out> {y,a,b,c};
In> x1
Out> {a,b,c};
In> x2[2]:=A
Out> True;
```

We have modified the second element of `x2`, but `x1` is still the same.

```
In> x2
Out> {y,A,b,c};
In> x1
Out> {a,b,c};
In> x2:=x1
Out> {A,b,c};
```

Now `x2` and `x1` refer to the same list.

```
In> x2[1]:=A
Out> True;
```

We have modified the first element of `x2`, and `x1` is also modified.

```
In> x2
Out> {A,b,c};
In> x1
Out> {A,b,c};
```

A programmer should always be cautious when dealing with destructive operations. Sometimes it is not desirable to change the original expression. The language deals with it this way because of performance considerations. Operations can be made non-destructive by using `FlatCopy`:

```
In> x1:={a,b,c}
Out> {a,b,c};
In> DestructiveReverse(x1)
Out> {c,b,a};
In> x1
Out> {a};
In> x1:={a,b,c}
Out> {a,b,c};
In> DestructiveReverse(FlatCopy(x1))
Out> {c,b,a};
In> x1
Out> {a,b,c};
```

`FlatCopy` copies the elements of an expression only at the top level of nesting. This means that if a list contains sub-lists, they are not copied, but references to them are copied instead:

```
In> dict1:={}
Out> {};
In> dict1["name"]:="John";
Out> True;
In> dict2:=FlatCopy(dict1)
Out> {{"name","John"}};
In> dict2["name"]:="Mark";
Out> True;
In> dict1
Out> {{"name","Mark"}};
```

A workaround for this is to use `Subst` to copy the entire tree:

```
In> dict1:={}
Out> {};
In> dict1["name"]:="John";
Out> True;
In> dict2:=Subst(a,a)(dict1)
Out> {{"name","John"}};
In> dict2["name"]:="Mark";
Out> True;
In> dict1
Out> {{"name","John"}};
In> dict2
Out> {{"name","Mark"}};
```

Chapter 3

Coding style

3.1 Introduction

This chapter intends to describe the coding style and conventions applied in Yacas in order to make sure the engine always returns the correct result. This is an attempt at fending off such errors by combining rule-based programming with a clear coding style which should make help avoid these mistakes.

3.2 Interactions of rules and types

One unfortunate disadvantage of rule-based programming is that rules can sometimes cooperate in unwanted ways.

One example of how rules can produce unwanted results is the rule `a*0 <-- 0`. This would always seem to be true. However, when a is a vector, e.g. `a:={b,c,d}`, then `a*0` should actually return `{0,0,0}`, that is, a zero vector. The rule `a*0 <-- 0` actually changes the type of the expression from a vector to an integer! This can have severe consequences when other functions using this expressions as an argument expect a vector, or even worse, have a definition of how to work on vectors, and a different one for working on numbers.

When writing rules for an operator, it is assumed that the operator working on arguments, e.g. `Cos` or `*`, will always have the same properties regardless of the arguments. The Taylor series expansion of $\cos a$ is the same regardless of whether a is a real number, complex number or even a matrix. Certain trigonometric identities should hold for the `Cos` function, regardless of the type of its argument.

If a function is defined which does not adhere to these rules when applied to another type, a different function name should be used, to avoid confusion.

By default, if a variable has not been bound yet, it is assumed to be a number. If it is in fact a more complex object, e.g. a vector, then you can declare it to be an "incomplete type" vector, using `Object("IsVector",x)` instead of `x`. This expression will evaluate to `x` if and only if `x` is a vector at that moment of evaluation. Otherwise it returns unevaluated, and thus stays an incomplete type.

So this means the type of a variable is numeric unless otherwise stated by the user, using the "`Object`" command. No rules should ever work on incomplete types. It is just meant for delayed simplification.

The topic of implicit type of an object is important, since many rules need to assume something about their argument types.

3.3 Ordering of rules

The implementor of a rule set can specify the order in which rules should be tried. This can be used to let the engine try more specific rules (those involving more elements in the pattern) before trying less specific rules. Ordering of rules can be also explicitly given by precedence numbers. The Yacas engine will split the expression into subexpressions, and will try to apply all matching rules to a given subexpression in order of precedence.

A rule with precedence 100 is defined by the syntax such as

```
100 # f(_x + _y) <-- f(x) + f(y);
```

The problem mentioned above with a rule for vectors and scalars could be solved by making two rules:

1. ab (if b is a vector and a is a number) `<--` return vector of each component multiplied by a.
2. $a \cdot 0$ `<--` 0

So vector multiplication would be tried first.

The ordering of the precedence of the rules in the standard math scripts is currently:

- 50-60: Args are numbers: directly calculate. These are put in the beginning, so they are tried first. This is useful for quickly obtaining numeric results if all the arguments are numeric already, and symbolic transformations are not necessary.
- 100-199: tautologies. Transformations that do not change the type of the argument, and are always true.
- 200-399: type-specific transformations. Transformations for specific types of objects.
- 400-599: transformations on scalars (variables are assumed to be scalars). Meaning transformations that can potentially change the type of an argument.

3.4 Writing new library functions

When you implement new library functions, you need to make your new code compatible and consistent with the rest of the library. Here are some relevant considerations.

To evaluate or not to evaluate

Currently, a general policy in the library is that functions do nothing unless their arguments actually allow something to be evaluated. For instance, if the function expects a variable name but instead gets a list, or expects a list but instead gets a string, in most cases it seems to be a good idea to do nothing and return unevaluated. The unevaluated expression will propagate and will be easy to spot. Most functions can accomplish this by using type-checking predicates such as `IsInteger` in rules.

When dealing with numbers, Yacas tries to maintain exact answers as much as possible and evaluate to floating-point only

when explicitly told so (using `N()`). The general evaluation strategy for numerical functions such as `Sin` or `Gamma` is the following:

1. If `InNumericMode()` returns `True` and the arguments are numbers (perhaps complex numbers), the function should evaluate its result in floating-point to current precision.
2. Otherwise, if the arguments are such that the result can be calculated exactly, it should be evaluated and returned. E.g. `Sin(Pi/2)` returns `1`.
3. Otherwise the function should return unevaluated (but usually with its arguments evaluated).

Here are some examples of this behavior:

```
In> Sin(3)
Out> Sin(3);
In> Gamma(8)
Out> 5040;
In> Gamma(-11/2)
Out> (64*Sqrt(Pi))/10395;
In> Gamma(8/7)
Out> Gamma(8/7);
In> N(Gamma(8/7))
Out> 0.9354375629;
In> N(Gamma(8/7+x))
Out> Gamma(x+1.1428571428);
In> Gamma(12/6+x)
Out> Gamma(x+2);
```

To implement this behavior, `Gamma` and other mathematical functions usually have two variants: the "symbolic" one and the "numerical" one. For instance, there are `Sin` and `MathSin`, `Ln` and `Internal'LnNum`, `Gamma` and `Internal'GammaNum`. (Here `MathSin` happens to be a core function but it is not essential.) The "numerical" functions always evaluate to floating-point results. The "symbolic" function serves as a front-end; it evaluates when the result can be expressed exactly, or calls the "numerical" function if `InNumericMode()` returns `True`, and otherwise returns unevaluated.

The "symbolic" function usually has multiple rules while the "numerical" function is usually just one large block of number-crunching code.

Using `N()` and `InNumericMode()` in scripts

As a rule, `N()` should be avoided in code that implements basic numerical algorithms. This is because `N()` itself is implemented in the library and it may need to use some of these algorithms. Arbitrary-precision math can be handled by core functions such as `MathDivide`, `MathSin` and so on, without using `N()`. For example, if your code needs to evaluate $\sqrt{\pi}$ to many digits as an intermediate result, it is better to write `MathSqrt(Internal'Pi())` than `N(Sqrt(Pi))` because it makes for faster, more reliable code.

Using `Builtin'Precision'Set()`

The usual assumption is that numerical functions will evaluate floating-point results to the currently set precision. For intermediate calculations, a higher working precision is sometimes needed. In this case, your function should set the precision back to the original value at the end of the calculation and round off the result.

Using verbose mode

For routines using complicated algorithms, or when evaluation takes a long time, it is usually helpful to print some diagnostic information, so that the user can at least watch some progress. The current convention is that if `InVerboseMode()` returns `True`, functions may print diagnostic information. (But do not print too much!). Verbose mode is turned on by using the function `V(expression)`. The expression is evaluated in verbose mode.

Procedural programming or rule-based programming?

Two considerations are relevant to this decision. First, whether to use multiple rules with predicates or one rule with multiple `If()`s. Consider the following sample code for the "double factorial" function $n!! \equiv n\,(n-2)\,...$ written using predicates and rules:

```
1# 0 !! <-- 1;
1# 1 !! <-- 1;
2# (n_IsEven) !! <-- 2^(n/2)*n!;
3# (n_IsOdd) !! <-- n*(n-2)!!;
```

and an equivalent code with one rule:

```
n!! := If(n=0 Or n=1, 1,
  If(IsEven(n), 2^(n/2)*n!,
  If(IsOdd(n), n*(n-2)!!, Hold(n!!)))
);
```

(Note: This is not the way $n!!$ is implemented in the library.) The first version is a lot more clear. Yacas is very quick in rule matching and evaluation of predicates, so the first version is (marginally) faster. So it seems better to write a few rules with predicates than one rule with multiple `If()` statements.

The second question is whether to use recursion or loops. Recursion makes code more elegant but it is slower and limited in depth. Currently the default recursion depth of 1000 is enough for most casual calculations and yet catches infinite recursion errors relatively quickly. Because of clearer code, it seems better to use recursion in situations where the number of list elements will never become large. In numerical applications, such as evaluation of Taylor series, recursion usually does not pay off.

3.5 Reporting errors

Errors occurring because of invalid argument types should be reported only if absolutely necessary. (In the future there may be a static type checker implemented that will make explicit checking unnecessary.)

Errors of invalid values, e.g. a negative argument of real logarithm function, or a malformed list, mean that a human has probably made a mistake, so the errors need to be reported. "Internal errors", i.e. program bugs, certainly need to be reported.

There are currently two facilities for reporting errors: a "hard" one and a "soft" one.

The "hard" error reporting facility is the function `Check`. For example, if `x=-1`, then

```
Check(x>0,"bad x");
```

will immediately halt the execution of a Yacas script and print the error messsage. This is implemented as a C++ exception. A drawback of this mechanism is that the Yacas stack unwinding is not performed by the Yacas interpreter, so global variables such as `InNumericMode()`, `Verbose`, `Builtin'Precision'Set()`

may keep the intermediate values they had been assigned just before the error occurred. Also, sometimes it is better for the program to be able to catch the error and continue.

The "soft" error reporting is provided by the functions `Assert` and `IsError`, e.g.

```
Assert("domain", x) x>0;
If(IsError("domain"), ...);
```

The error will be reported but execution will continue normally until some other function "handles" the error (prints the error message or does something else appropriate for that error). Here the string `"domain"` is the "error type" and `x` will be the information object for this error. The error object can be any expression, but it is probably a good idea to choose a short and descriptive string for the error type.

The function `GetErrorTableau()` returns an associative list that accumulates all reported error objects. When errors are "handled", their objects should be removed from the list. The utility function `DumpErrors()` is a simple error handler that prints all errors and clears the list. Other handlers are `GetError` and `ClearError`. These functions may be used to handle errors when it is safe to do so.

The "soft" error reporting facility is safer and more flexible than the "hard" facility. However, the disadvantage is that errors are not reported right away and pointless calculations may continue for a while until an error is handled.

Chapter 4

Advanced example 1: parsing expressions (CForm)

In this chapter we show how Yacas represents expressions and how one can build functions that work on various types of expressions. Our specific example will be `CForm()`, a standard library function that converts Yacas expressions into C or C++ code. Although the input format of Yacas expressions is already very close to C and perhaps could be converted to C by means of an external text filter, it is instructive to understand how to use Yacas to parse its own expressions and produce the corresponding C code. Here we shall only design the core mechanism of `CForm()` and build a limited version that handles only expressions using the four arithmetic actions.

4.1 Recursive parsing of expression trees

As we have seen in the tutorial, Yacas represents all expressions as trees, or equivalently, as lists of lists. For example, the expression "`a+b+c+d+e`" is for Yacas a tree of depth 4 that could be visualized as

```
  "+"
a  "+"
  b  "+"
    c  "+"
      d   e
```

or as a nested list: `("+" a ("+" b ("+" c ("+" d e))))`.

Complicated expressions are thus built from simple ones in a general and flexible way. If we want a function that acts on sums of any number of terms, we only need to define this function on a single atom and on a sum of two terms, and the Yacas engine will recursively perform the action on the entire tree.

So our first try is to define rules for transforming an atom and for transforming sums and products. The result of `CForm()` will always be a string. We can use recursion like this:

```
In> 100 # CForm(a_IsAtom) <-- String(a);
Out> True;
In> 100 # CForm(_a + _b) <-- CForm(a) : \
  " + " : CForm(b);
Out> True;
In> 100 # CForm(_a * _b) <-- CForm(a) : \
  " * " : CForm(b);
Out> True;
```

We used the string concatenation operator "`:`" and we added spaces around the binary operators for clarity. All rules have the same precedence 100 because there are no conflicts in rule ordering so far: these rules apply in mutually exclusive cases. Let's try converting some simple expressions now:

```
In> CForm(a+b*c);
Out> "a + b * c";
In> CForm(a+b*c*d+e+1+f);
Out> "a + b * c * d + e + 1 + f";
```

With only three rules, we were able to process even some complicated expressions. How did it work? We could illustrate the steps Yacas went through when simplifying `CForm(a+b*c)` roughly like this:

```
CForm(a+b*c)
    ... apply 2nd rule
CForm(a) : " + " : CForm(b*c)
    ... apply 1st rule and 3rd rule
"a" : " + " : CForm(b) : " * " : CForm(c)
    ... apply 1st rule
"a" : " + " : "b" : " * " : "c"
    ... concatenate strings
"a + b * c"
```

4.2 Handling precedence of infix operations

It seems that recursion will do all the work for us. The power of recursion is indeed great and extensive use of recursion is built into the design of Yacas. We might now add rules for more operators, for example, the unary addition, subtraction and division:

```
100 # CForm(+ _a) <-- "+ " : CForm(a);
100 # CForm(- _a) <-- "- " : CForm(a);
100 # CForm(_a - _b) <-- CForm(a) : " - "
  : CForm(b);
100 # CForm(_a / _b) <-- CForm(a) : " / "
  : CForm(b);
```

However, soon we find that we forgot about operator precedence. Our simple-minded `CForm()` gives wrong C code for expressions like this:

```
In> CForm( (a+b) * c );
Out> "a + b * c";
```

We need to get something like "(a+b)*c" in this case. How would we add a rule to insert parentheses around subexpressions? A simple way out would be to put parentheses around every subexpression, replacing our rules by something like this:

```
100 # CForm(_a + _b) <-- "(" : CForm(a)
  : " + " : CForm(b) : ")";
100 # CForm(- _a) <-- "(- " : CForm(a)
  : ")";
```

and so on. This will always produce correct C code, e.g. in our case "((a+b)*c)", but generally the output will be full of unnecessary parentheses. It is instructive to find a better solution.

We could improve the situation by inserting parentheses only if the higher-order expression requires them; for this to work, we need to make a call such as `CForm(a+b)` aware that the enveloping expression has a multiplication by `c` around the addition `a+b`. This can be implemented by passing an extra argument to `CForm()` that will indicate the precedence of the enveloping operation. A compound expression that uses an infix operator must be bracketed if the precedence of that infix operator is higher than the precedence of the enveloping infix operation.

We shall define an auxiliary function also named "CForm" but with a second argument, the precedence of the enveloping infix operation. If there is no enveloping operation, we shall set the precedence to a large number, e.g. 60000, to indicate that no parentheses should be inserted around the whole expression. The new "CForm(expr, precedence)" will handle two cases: either parentheses are necessary, or unnecessary. For clarity we shall implement these cases in two separate rules. The initial call to "CForm(expr)" will be delegated to "CForm(expr, precedence)".

The precedence values of infix operators such as "`+`" and "`*`" are defined in the Yacas library but may change in a future version. Therefore, we shall not hard-code these precedence values but instead use the function `OpPrecedence()` to determine them. The new rules for the "`+`" operation could look like this:

```
PlusPrec := OpPrecedence("+");
100 # CForm(_expr) <-- CForm(expr, 60000);
100 # CForm(_a + _b, _prec)_(PlusPrec>prec)
  <-- "(" : CForm(a, PlusPrec) : " + "
  : CForm(b, PlusPrec) : ")";
120 # CForm(_a + _b, _prec) <--
    CForm(a, PlusPrec) : " + "
: CForm(b, PlusPrec);
```

and so on. We omitted the predicate for the last rule because it has a later precedence than the preceding rule.

The way we wrote these rules is unnecessarily repetitive but straightforward and it illustrates the central ideas of expression processing in Yacas. The standard library implements `CForm()` essentially in this way. In addition the library implementation supports standard mathematical functions, arrays and so on, and is somewhat better organized to allow easier extensions and avoid repetition of code.

Chapter 5

Yacas programming pitfalls

No programming language is without programming pitfalls, and YACAS has its fair share of pitfalls.

5.1 All rules are global

All rules are global, and a consequence is that rules can clash or silently shadow each other, if the user defines two rules with the same patterns and predicates but different bodies.

For example:

```
In> f(0) <-- 1
Out> True;
In> f(x_IsConstant) <-- Sin(x)/x
Out> True;
```

This can happen in practice, if care is not taken. Here two transformation rules are defined which both have the same precedence (since their precedence was not explicitly set). In that case YACAS gets to decide which one to try first. Such problems can also occur where one transformation rule (possibly defined in some other file) has a wrong precedence, and thus masks another transformation rule. It is necessary to think of a scheme for assigning precedences first. In many cases, the order in which transformation rules are applied is important.

In the above example, because YACAS gets to decide which rule to try first, it is possible that f(0) invokes the second rule, which would then mask the first so the first rule is never called. Indeed, in YACAS version 1.0.51,

```
In> f(0)
Out> Undefined;
```

The order the rules are applied in is undefined if the precedences are the same. The precedences should only be the same if order does not matter. This is the case if, for instance, the two rules apply to different argument patters that could not possibly mask each other.

The solution could have been either:

```
In> 10 # f(0) <-- 1
Out> True;
In> 20 # f(x_IsConstant) <-- Sin(x)/x
Out> True;
In> f(0)
Out> 1;
```

or

```
In> f(0) <-- 1
Out> True;
In> f(x_IsConstant)_(x != 0) <-- Sin(x)/x
Out> True;
In> f(0)
Out> 1;
```

So either the rules should have distinct precedences, or they should have mutually exclusive predicates, so that they do not collide.

5.2 Objects that look like functions

An expression that looks like a "function", for example `AbcDef(x,y)`, is in fact either a call to a "core function" or to a "user function", and there is a huge difference between the behaviors. Core functions immediately evaluate to something, while user functions are really just symbols to which evaluation rules may or may not be applied.

For example:

```
In> a+b
Out> a+b;
In> 2+3
Out> 5;
In> MathAdd(a,b)
In function "MathAdd" :
bad argument number 1 (counting from 1)
The offending argument a evaluated to a
CommandLine(1) : Invalid argument

In> MathAdd(2,3)
Out> 5;
```

The `+` operator will return the object unsimplified if the arguments are not numeric. The `+` operator is defined in the standard scripts. `MathAdd`, however, is a function defined in the "core" to performs the numeric addition. It can only do this if the arguments are numeric and it fails on symbolic arguments. (The `+` operator calls `MathAdd` after it has verified that the arguments passed to it are numeric.)

A core function such as `MathAdd` can never return unevaluated, but an operator such as "`+`" is a "user function" which might or might not be evaluated to something.

A user function does not have to be defined before it is used. A consequence of this is that a typo in a function name or a variable name will always go unnoticed. For example:

```
In> f(x_IsInteger,y_IsInteger) <-- Mathadd(x,y)
Out> True;
In> f(1,2)
Out> Mathadd(1,2);
```

Here we made a typo: we should have written `MathAdd`, but wrote `Mathadd` instead. YACAS happily assumed that we mean a new and (so far) undefined "user function" `Mathadd` and returned the expression unevaluated.

In the above example it was easy to spot the error. But this feature becomes more dangerous when it this mistake is made in

a part of some procedure. A call that should have been made to an internal function, if a typo was made, passes silently without error and returns unevaluated. The real problem occurs if we meant to call a function that has side-effects and we not use its return value. In this case we shall not immediately find that the function was not evaluated, but instead we shall encounter a mysterious bug later.

5.3 Guessing when arguments are evaluated and when not

If your new function does not work as expected, there is a good chance that it happened because you did not expect some expression which is an argument to be passed to a function to be evaluated when it is in fact evaluated, or vice versa.

For example:

```
In> p:=Sin(x)
Out> Sin(x);
In> D(x)p
Out> Cos(x);
In> y:=x
Out> x;
In> D(y)p
Out> 0;
```

Here the first argument to the differentiation function is not evaluated, so `y` is not evaluated to `x`, and `D(y)p` is indeed 0.

The confusing effect of `HoldArg`

The problem of distinguishing evaluated and unevaluated objects becomes worse when we need to create a function that does not evaluate its arguments.

Since in YACAS evaluation starts from the bottom of the expression tree, all "user functions" will appear to evaluate their arguments by default. But sometimes it is convenient to prohibit evaluation of a particular argument (using `HoldArg` or `HoldArgNr`).

For example, suppose we need a function `A(x,y)` that, as a side-effect, assigns the variable `x` to the sum of `x` and `y`. This function will be called when `x` already has some value, so clearly the argument `x` in `A(x,y)` should be unevaluated. It is possible to make this argument unevaluated by putting `Hold()` on it and always calling `A(Hold(x), y)`, but this is not very convenient and easy to forget. It would be better to define `A` so that it always keeps its first argument unevaluated.

If we define a rule base for `A` and declare `HoldArg`,

```
Function() A(x,y);
HoldArg("A", x);
```

then we shall encounter a difficulty when working with the argument `x` inside of a rule body for `A`. For instance, the simple-minded implementation

```
A(_x, _y) <-- (x := x+y);
```

does not work:

```
In> [ a:=1; b:=2; A(a,b);]
Out> a+2;
```

In other words, the `x` inside the body of `A(x,y)` did not evaluate to `1` when we called the function `:=`. Instead, it was left unevaluated as the atom `x` on the left hand side of `:=`, since `:=` does not evaluate its left argument. It however evaluates its right argument, so the `y` argument was evaluated to `2` and the `x+y` became `a+2`.

The evaluation of `x` in the body of `A(x,y)` was prevented by the `HoldArg` declaration. So in the body, `x` will just be the atom `x`, unless it is evaluated again. If you pass `x` to other functions, they will just get the atom `x`. Thus in our example, we passed `x` to the function `:=`, thinking that it will get `a`, but it got an unevaluated atom `x` on the left side and proceeded with that.

We need an explicit evaluation of `x` in this case. It can be performed using `Eval`, or with backquoting, or by using a core function that evaluates its argument. Here is some code that illustrates these three possibilities:

```
A(_x, _y) <-- [ Local(z); z:=Eval(x); z:=z+y; ]
```

(using explicit evaluation) or

```
A(_x, _y) <-- `(@x := @x + y);
```

(using backquoting) or

```
A(_x, _y) <-- MacroSet(x, x+y);
```

(using a core function `MacroSet` that evaluates its first argument).

However, beware of a clash of names when using explicit evaluations (as explained above). In other words, the function `A` as defined above will not work correctly if we give it a variable also named `x`. The `LocalSymbols` call should be used to get around this problem.

Another caveat is that when we call another function that does not evaluate its argument, we need to substitute an explicitly evaluated `x` into it. A frequent case is the following: suppose we have a function `B(x,y)` that does not evaluate `x`, and we need to write an interface function `B(x)` which will just call `B(x,0)`. We should use an explicit evaluation of `x` to accomplish this, for example

```
B(_x) <-- `B(@x,0);
```

or

```
B(_x) <-- B @ {x, 0};
```

Otherwise `B(x,y)` will not get the correct value of its first parameter `x`.

Special behavior of `Hold`, `UnList` and `Eval`

When an expression is evaluated, all matching rules are applied to it repeatedly until no more rules match. Thus an expression is "completely" evaluated. There are, however, two cases when recursive application of rules is stopped at a certain point, leaving an expression not "completely" evaluated:

1. The expression which is the result of a call to a Yacas core function is not evaluated further, even if some rules apply to it.
2. The expression is a variable that has a value assigned to it; for example, the variable `x` might have the expression `y+1` as the value. That value is not evaluated again, so even if `y` has been assigned another value, say, `y=2` a Yacas expression such as `2*x+1` will evaluate to `2*(y+1)+1` and not to `7`. Thus, a variable can have some unevaluated expression as its value and the expression will not be re-evaluated when the variable is used.

The first possibility is mostly without consequence because almost all core functions return a simple atom that does not require further evaluation. However, there are two core functions that can return a complicated expression: `Hold` and `UnList`. Thus, these functions can produce arbitrarily complicated Yacas expressions that will be left unevaluated. For example, the result of

```
UnList({Sin, 0})
```

is the same as the result of

```
Hold(Sin(0))
```

and is the unevaluated expression `Sin(0)` rather than `0`.

Typically you want to use `UnList` because you need to construct a function call out of some objects that you have. But you need to call `Eval(UnList(...))` to actually evaluate this function call. For example:

```
In> UnList({Sin, 0})
Out> Sin(0);
In> Eval(UnList({Sin, 0}))
Out> 0;
```

In effect, evaluation can be stopped with `Hold` or `UnList` and can be explicitly restarted by using `Eval`. If several levels of unevaluation are used, such as `Hold(Hold(...))`, then the same number of `Eval` calls will be needed to fully evaluate an expression.

```
In> a:=Hold(Sin(0))
Out> Sin(0);
In> b:=Hold(a)
Out> a;
In> c:=Hold(b)
Out> b;
In> Eval(c)
Out> a;
In> Eval(Eval(c))
Out> Sin(0);
In> Eval(Eval(Eval(c)))
Out> 0;
```

A function `FullEval` can be defined for "complete" evaluation of expressions, as follows:

```
LocalSymbols(x,y)
[
  FullEval(_x) <-- FullEval(x,Eval(x));
  10 # FullEval(_x,_x) <-- x;
  20 # FullEval(_x,_y) <-- FullEval(y,Eval(y));
];
```

Then the example above will be concluded with:

```
In> FullEval(c);
Out> 0;
```

Correctness of parameters to functions is not checked

Because YACAS does not enforce type checking of arguments, it is possible to call functions with invalid arguments. The default way functions in YACAS should deal with situations where an action can not be performed, is to return the expression unevaluated. A function should know when it is failing to perform a task. The typical symptoms are errors that seem obscure, but just mean the function called should have checked that it can perform the action on the object.

For example:

```
In> 10 # f(0) <-- 1;
Out> True;
In> 20 # f(_n) <-- n*f(n-1);
Out> True;
In> f(3)
Out> 6;
In> f(1.3)
CommandLine(1): Max evaluation stack depth reached.
```

Here, the function `f` is defined to be a factorial function, but the function fails to check that its argument is a positive integer, and thus exhausts the stack when called with a non-integer argument. A better way would be to write

```
In> 20 # f(n_IsPositiveInteger) <-- n*f(n-1);
```

Then the function would have returned unevaluated when passed a non-integer or a symbolic expression.

5.4 Evaluating variables in the wrong scope

There is a subtle problem that occurs when `Eval` is used in a function, combined with local variables. The following example perhaps illustrates it:

```
In> f1(x):=[Local(a);a:=2;Eval(x);];
Out> True;
In> f1(3)
Out> 3;
In> f1(a)
Out> 2;
```

Here the last call should have returned `a`, but it returned `2`, because `x` was assigned the value `a`, and `a` was assigned locally the value of `2`, and `x` gets re-evaluated. This problem occurs when the expression being evaluated contains variables which are also local variables in the function body. The solution is to use the `LocalSymbols` function for all local variables defined in the body.

The following illustrates this:

```
In> f2(x):=LocalSymbols(a)[Local(a);a:=2;Eval(x);];
Out> True;
In> f1(3)
Out> 3;
In> f2(a)
Out> a;
```

Here `f2` returns the correct result. `x` was assigned the value `a`, but the `a` within the function body is made distinctly different from the one referred to by `x` (which, in a sense, refers to a global `a`), by using `LocalSymbols`.

This problem generally occurs when defining functions that re-evaluate one of its arguments, typically functions that perform a loop of some sort, evaluating a body at each iteration.

Chapter 6

Debugging in Yacas

6.1 Introduction

When writing a code segment, it is generally a good idea to separate the problem into many small functions. Not only can you then reuse these functions on other problems, but it makes debugging easier too.

For debugging a faulty function, in addition to the usual trial-and-error method and the "print everything" method, Yacas offers some trace facilities. You can try to trace applications of rules during evaluation of the function (`TraceRule()`, `TraceExp()`) or see the stack after an error has occurred (`TraceStack()`).

There is also an interactive debugger, which shall be introduced in this chapter.

Finally, you may want to run a debugging version of Yacas. This version of the executable maintains more information about the operations it performs, and can report on this.

This chapter will start with the interactive debugger, as it is the easiest and most useful feature to use, and then proceed to explain the trace and profiling facilities. Finally, the internal workings of the debugger will be explained. It is highly customizable (in fact, most of the debugging code is written in Yacas itself), so for bugs that are really difficult to track one can write custom code to track it.

6.2 The trace facilities

The trace facilities are:

1. `TraceExp` : traces the full expression, showing all calls to user- or system-defined functions, their arguments, and the return values. For complex functions this can become a long list of function calls.
2. `TraceRule` : traces one single user-defined function (rule). It shows each invocation, the arguments passed in, and the returned values. This is useful for tracking the behavior of that function in the environment it is intended to be used in.
3. `TraceStack` : shows a few last function calls before an error has occurred.
4. `Profile` : report on statistics (number of times functions were called, etc.). Useful for performance analysis.

The online manual pages (e.g. `?TraceStack`) have more information about the use of these functions.

An example invocation of `TraceRule` is

```
In> TraceRule(x+y)2+3*5+4;
```

Which should then show something to the effect of

```
  TrEnter(2+3*5+4);
    TrEnter(2+3*5);
       TrArg(2,2);
          TrArg(3*5,15);
       TrLeave(2+3*5,17);
        TrArg(2+3*5,17);
        TrArg(4,4);
    TrLeave(2+3*5+4,21);
Out> 21;
```

Chapter 7

Custom evaluation facilities

Yacas supports a special form of evaluation where hooks are placed when evaluation enters or leaves an expression.

This section will explain the way custom evaluation is supported in YACAS, and will proceed to demonstrate how it can be used by showing code to trace, interactively step through, profile, and write custom debugging code.

Debugging, tracing and profiling has been implemented in the debug.rep/ module, but a simplification of that code will be presented here to show the basic concepts.

7.1 The basic infrastructure for custom evaluation

The name of the function is `CustomEval`, and the calling sequence is:

```
CustomEval(enter,leave,error,expression);
```

Here, `expression` is the expression to be evaluated, `enter` some expression that should be evaluated when entering an expression, and `leave` an expression to be evaluated when leaving evaluation of that expression.

The `error` expression is evaluated when an error occurred. If an error occurs, this is caught high up, the `error` expression is called, and the debugger goes back to evaluating `enter` again so the situation can be examined. When the debugger needs to stop, the `error` expression is the place to call `CustomEval'Stop()` (see explanation below).

The `CustomEval` function can be used to write custom debugging tools. Examples are:

1. a trace facility following entering and leaving functions
2. interactive debugger for stepping through evaluation of an expression.
3. profiler functionality, by having the callback functions do the bookkeeping on counts of function calls for instance.

In addition, custom code can be written to for instance halt evaluation and enter interactive mode as soon as some very specific situation occurs, like "stop when function foo is called while the function bar is also on the call stack and the value of the local variable x is less than zero".

As a first example, suppose we define the functions TraceEnter(), TraceLeave() and `TraceExp()` as follows:

```
TraceStart() := [indent := 0;];
TraceEnter() :=
[
   indent++;
   Space(2*indent);
   Echo("Enter ",CustomEval'Expression());
];
TraceLeave() :=
[
   Space(2*indent);
   Echo("Leave ",CustomEval'Result());
   indent--;
];
Macro(TraceExp,{expression})
[
   TraceStart();
   CustomEval(TraceEnter(),
              TraceLeave(),
              CustomEval'Stop(),@expression);
];
```

allows us to have tracing in a very basic way. We can now call:

```
In> TraceExp(2+3)
  Enter 2+3
    Enter 2
    Leave 2
    Enter 3
    Leave 3
    Enter IsNumber(x)
      Enter x
      Leave 2
    Leave True
    Enter IsNumber(y)
      Enter y
      Leave 3
    Leave True
    Enter True
    Leave True
    Enter MathAdd(x,y)
      Enter x
      Leave 2
      Enter y
      Leave 3
    Leave 5
  Leave 5
Out> 5;
```

This example shows the use of `CustomEval'Expression` and `CustomEval'Result`. These functions give some extra access to interesting information while evaluating the expression. The functions defined to allow access to information while evaluating are:

1. `CustomEval'Expression()` - return expression currently on the top call stack for evaluation.
2. `CustomEval'Result()` - when the `leave` argument is called this function returns what the evaluation of the top expression will return.

3. `CustomEval'Locals()` - returns a list with the current local variables.
4. `CustomEval'Stop()` - stop debugging execution

7.2 A simple interactive debugger

The following code allows for simple interactive debugging:

```
DebugStart():=
[
   debugging:=True;
   breakpoints:={};
];
DebugRun():= [debugging:=False;];
DebugStep():=[debugging:=False;nextdebugging:=True;];
DebugAddBreakpoint(fname_IsString) <--
   [ breakpoints := fname:breakpoints;];
BreakpointsClear() <-- [ breakpoints := {};];
Macro(DebugEnter,{})
[
   Echo(">>> ",CustomEval'Expression());
   If(debugging = False And
      IsFunction(CustomEval'Expression()) And
      Contains(breakpoints,
      Type(CustomEval'Expression())),
        debugging:=True);
   nextdebugging:=False;
   While(debugging)
   [
      debugRes:=
        Eval(FromString(
          ReadCmdLineString("Debug> ");")
          Read());
      If(debugging,Echo("DebugOut> ",debugRes));
   ];
   debugging:=nextdebugging;
];
Macro(DebugLeave,{})
[
   Echo(CustomEval'Result(),
        " <-- ",CustomEval'Expression());
];
Macro(Debug,{expression})
[
   DebugStart();
   CustomEval(DebugEnter(),
              DebugLeave(),
              debugging:=True,@expression);
];
```

This code allows for the following interaction:

```
In> Debug(2+3)
>>> 2+3
Debug>
```

The console now shows the current expression being evaluated, and a debug prompt for interactive debugging. We can enter `DebugStep()`, which steps to the next expression to be evaluated:

```
Debug> DebugStep();
>>> 2
Debug>
```

This shows that in order to evaluate `2+3` the interpreter first needs to evaluate `2`. If we step further a few more times, we arrive at:

```
>>> IsNumber(x)
Debug>
```

Now we might be curious as to what the value for `x` is. We can dynamically obtain the value for `x` by just typing it on the command line.

```
>>> IsNumber(x)
Debug> x
DebugOut> 2
```

`x` is set to `2`, so we expect `IsNumber` to return `True`. Stepping again:

```
Debug> DebugStep();
>>> x
Debug> DebugStep();
2  <-- x
True  <-- IsNumber(x)
>>> IsNumber(y)
```

So we see this is true. We can have a look at which local variables are currently available by calling `CustomEval'Locals()`:

```
Debug> CustomEval'Locals()
DebugOut> {arg1,arg2,x,y,aLeftAssign,aRightAssign}
```

And when bored, we can proceed with `DebugRun()` which will continue the debugger until finished in this case (a more sophisticated debugger can add breakpoints, so running would halt again at for instance a breakpoint).

```
Debug> DebugRun()
>>> y
3  <-- y
True  <-- IsNumber(y)
>>> True
True  <-- True
>>> MathAdd(x,y)
>>> x
2  <-- x
>>> y
3  <-- y
5  <-- MathAdd(x,y)
5  <-- 2+3
Out> 5;
```

The above bit of code also supports primitive breakpointing, in that one can instruct the evaluator to stop when a function will be entered. The debugger then stops just before the arguments to the function are evaluated. In the following example, we make the debugger stop when a call is made to the `MathAdd` function:

```
In> Debug(2+3)
>>> 2+3
Debug> DebugAddBreakpoint("MathAdd")
DebugOut> {"MathAdd"}
Debug> DebugRun()
>>> 2
2  <-- 2
>>> 3
3  <-- 3
>>> IsNumber(x)
>>> x
2  <-- x
True  <-- IsNumber(x)
>>> IsNumber(y)
>>> y
3  <-- y
```

```
True  <-- IsNumber(y)
>>> True
True  <-- True
>>> MathAdd(x,y)
Debug>
```

The arguments to `MathAdd` can now be examined, or execution continued.

One great advantage of defining much of the debugger in script code can be seen in the `DebugEnter` function, where the breakpoints are checked, and execution halts when a breakpoint is reached. In this case the condition for stopping evaluation is rather simple: when entering a specific function, stop. However, nothing stops a programmer from writing a custom debugger that could stop on any condition, halting at e very special case.

7.3 Profiling

A simple profiler that counts the number of times each function is called can be written such:

```
ProfileStart():=
[
   profilefn:={};
];
10 # ProfileEnter()
     _(IsFunction(CustomEval'Expression())) <--
[
   Local(fname);
   fname:=Type(CustomEval'Expression());
   If(profilefn[fname]=Empty,profilefn[fname]:=0);
   profilefn[fname] := profilefn[fname]+1;
];
Macro(Profile,{expression})
[
   ProfileStart();
   CustomEval(ProfileEnter(),True,
              CustomEval'Stop(),@expression);
   ForEach(item,profilefn)
     Echo("Function ",item[1]," called ",
          item[2]," times");
];
```

which allows for the interaction:

```
In> Profile(2+3)
Function MathAdd called 1  times
Function IsNumber called 2  times
Function + called 1  times
Out> True;
```

7.4 The Yacas Debugger

Why introduce a debug version?

The reason for introducing a debug version is that for a debugger it is often necessary to introduce features that make the interpreter slower. For the main kernel this is unacceptable, but for a debugging version this is defendable. It is good for testing small programs, to see where a calculation breaks. Having certain features only in the debug version keeps the release executable can be kept lean and mean, while still offering advanced debug features.

How to build the debug version of Yacas ?

The debug version has to be built separately from the "production" version of Yacas (all source files have to be recompiled).

To build the debug version of yacas, run configure with

```
./configure --enable-debug
```

and after that

```
make
```

as usual.

What does the debug version of yacas offer?

The Yacas debugger is in development still, but already has some useful features.

When you build the debug version of yacas, and run a command, it will:

- keep track of the memory allocated and freed, and show any memory leaks when you quit the program.
- show which files are loaded to read function definitions and when. This is only done when the `--verbose-debug` flag is passed to the program at startup.
- keep a file name and line number for each object loaded from file, for debugging purposes.
- show you the stack trace when evaluation goes into an infinite recursion (equivalent of always using `TraceStack`) and print file names and line numbers for all rules.

Future versions will have the ability to step through code and to watch local and global variables while executing, modifying them on the fly.

Chapter 8

Advanced example 2: implementing a non-commutative algebra

We need to understand how to simplify expressions in Yacas, and the best way is to try writing our own algebraic expression handler. In this chapter we shall consider a simple implementation of a particular non-commutative algebra called the Heisenberg algebra. This algebra was introduced by Dirac to develop quantum field theory. We won't explain any physics here, but instead we shall to delve somewhat deeper into the workings of Yacas.

8.1 The problem

Suppose we want to define special symbols $A(k)$ and $B(k)$ that we can multiply with each other or by a number, or add to each other, but not commute with each other, i.e. $A(k)B(k) \neq B(k)A(k)$. Here k is merely a label to denote that $A(1)$ and $A(2)$ are two different objects. (In physics, these are called "creation" and "annihilation" operators for "bosonic quantum fields".) Yacas already assumes that the usual multiplication operator "`*`" is commutative. Rather than trying to redefine `*`, we shall introduce a special multiplication sign "`**`" that we shall use with the objects $A(k)$ and $B(k)$; between usual numbers this would be the same as normal multiplication. The symbols $A(k)$, $B(k)$ will never be evaluated to numbers, so an expression such as `2 ** A(k1) ** B(k2) ** A(k3)` is just going to remain like that. (In physics, commuting numbers are called "classical quantities" or "c-numbers" while non-commuting objects made up of A(k) and B(k) are called "quantum quantities" or "q-numbers".) There are certain commutation relations for these symbols: the A's commute between themselves, $A(k)A(l) = A(l)A(k)$, and also the B's, $B(k)B(l) = B(l)B(k)$. However, the A's don't commute with the B's: $A(k)B(l) - B(l)A(k) = \delta(k-l)$. Here the "`delta`" is a "classical" function (called the "Dirac δ-function") but we aren't going to do anything about it, just leave it symbolic.

We would like to be able to manipulate such expressions, expanding brackets, collecting similar terms and so on, while taking care to always keep the non-commuting terms in the correct order. For example, we want Yacas to automatically simplify `2**B(k1)**3**A(k2)` to `6**B(k1)**A(k2)`. Our goal is not to implement a general package to tackle complicated non-commutative operations; we merely want to teach Yacas about these two kinds of "quantum objects" called `A(k)` and `B(k)`, and we shall define one function that a physicist would need to apply to these objects. This function applied to any given expression containing A's and B's will compute something called a "vacuum expectation value", or "VEV" for short, of that expression. This function has "classical", i.e. commuting, values and is defined as follows: VEV of a commuting number is just that number, e.g. $\mathrm{VEV}(4) = 4$, $\mathrm{VEV}(\delta(k-l)) = \delta(k-l)$; and $\mathrm{VEV}(\mathrm{X}A(k)) = 0$, $\mathrm{VEV}(B(k)\mathrm{X}) = 0$ where X is any expression, commutative or not. It is straightforward to compute VEV of something that contains A's and B's: one just uses the commutation relations to move all B's to the left of all A's, and then applies the definition of VEV, simply throwing out any remaining q-numbers.

8.2 First steps

The first thing that comes to mind when we start implementing this in Yacas is to write a rule such as

```
10 # A(_k)**B(_l) <-- B(l)**A(k)
  + delta(k-l);
```

However, this is not going to work right away. In fact this will immediately give a syntax error because Yacas doesn't know yet about the new multiplication `**`. Let's fix that: we shall define a new infix operator with the same precedence as multiplication.

```
RuleBase("**", {x,y});
Infix("**", OpPrecedence("*"));
```

Now we can use this new multiplication operator in expressions, and it doesn't evaluate to anything – exactly what we need. But we find that things don't quite work:

```
In> A(_k)**B(_l) <-- B(l)**A(k)+delta(k-l);
Out> True;
In> A(x)**B(y)
Out> B(l)**A(k)+delta(k-l);
```

Yacas doesn't grok that `delta(k)`, `A(k)` and `B(k)` are functions. This can be fixed by declaring

```
RuleBase("A", {k});
RuleBase("B", {k});
RuleBase("delta", {k});
```

Now things work as intended:

```
In> A(y)**B(z)*2
Out> 2*(B(z)**A(y)+delta(y-z));
```

8.3 Structure of expressions

Are we done yet? Let's try to calculate more things with our A's and B's:

```
In> A(k)*2**B(l)
Out> 2*A(k)**B(l);
In> A(x)**A(y)**B(z)
Out> A(x)**A(y)**B(z);
In> (A(x)+B(x))**2**B(y)*3
Out> 3*(A(x)+B(x))**2**B(y);
```

After we gave it slightly more complicated input, Yacas didn't fully evaluate expressions containing the new `**` operation: it didn't move constants `2` and `3` together, didn't expand brackets, and, somewhat mysteriously, it didn't apply the rule in the first line above – although it seems like it should have. Before we hurry to fix these things, let's think some more about how Yacas represents our new expressions. Let's start with the first line above:

```
In> FullForm( A(k)*2**B(l) )
(** (* 2 (A k ))(B 1 ))
Out> 2*A(k)**B(l);
```

What looks like `2*A(k)**B(l)` on the screen is really `(2*A(k)) ** B(l)` inside Yacas. In other words, the commutation rule didn't apply because there is no subexpression of the form `A(...)**B(...)` in this expression. It seems that we would need many rules to exhaust all ways in which the adjacent factors `A(k)` and `B(l)` might be divided between subexpressions. We run into this difficulty because Yacas represents all expressions as trees of functions and leaves the semantics to us. To Yacas, the "`*`" operator is fundamentally no different from any other function, so `(a*b)*c` and `a*(b*c)` are two basically different expressions. It would take a considerable amount of work to teach Yacas to recognize all such cases as identical. This is a design choice and it was made by the author of Yacas to achieve greater flexibility and extensibility.

A solution for this problem is not to write rules for all possible cases (there are infinitely many cases) but to systematically reduce expressions to a *canonical form*. "Experience has shown that" (a phrase used when we can't come up with specific arguments) symbolic manipulation of unevaluated trees is not efficient unless these trees are forced to a pattern that reflects their semantics.

We should choose a canonical form for all such expressions in a way that makes our calculations – namely, the function `VEV()` – easier. In our case, our expressions contain two kinds of ingredients: normal, commutative numbers and maybe a number of noncommuting symbols `A(k)` and `B(k)` multiplied together with the "`**`" operator. It will not be possible to divide anything by $A(k)$ or $B(k)$ – such division is undefined.

A possible canonical form for expressions with A's and B's is the following. All commutative numbers are moved to the left of the expression and grouped together as one factor; all non-commutative products are simplified to a single chain, all brackets expanded. A canonical expression should not contain any extra brackets in its non-commutative part. For example, (A(x)+B(x)*x)**B(y)*y**A(z) should be regrouped as a sum of two terms, (y)**(A(x)**(B(y))**A(z)) and (x*y)**(B(x)**(B(y))**A(z)). Here we wrote out all parentheses to show explicitly which operations are grouped. (We have chosen the grouping of non-commutative factors to go from left to right, however this does not seem to be an important choice.) On the screen this will look simply `y ** A(x) ** B(y)` and `x*y** B(x) ** B(y) ** A(z)` because we have defined the precedence of the "**" operator to be the same as that of the normal multiplication, so Yacas won't insert any more parentheses.

This canonical form will allow Yacas to apply all the usual rules on the commutative factor while cleanly separating all non-commutative parts for special treatment. Note that a commutative factor such as `2*x` will be multiplied by a single non-commutative piece with "`**`".

The basic idea behind the "canonical form" is this: we should define our evaluation rules in such a way that any expression containing `A(k)` and `B(k)` will be always automatically reduced to the canonical form after one full evaluation. All functions on our new objects will assume that the object is already in the canonical form and should return objects in the same canonical form.

8.4 Implementing the canonical form

Now that we have a design, let's look at some implementation issues. We would like to write evaluation rules involving the new operator "`**`" as well as the ordinary multiplications and additions involving usual numbers, so that all "classical" numbers and all "quantum" objects are grouped together separately. This should be accomplished with rules that expand brackets, exchange the bracketing order of expressions and move commuting factors to the left. For now, we shall not concern ourselves with divisions and subtractions.

First, we need to distinguish "classical" terms from "quantum" ones. For this, we shall define a predicate `IsQuantum()` recursively, as follows:

```
    /* Predicate IsQuantum(): will return
  True if the expression contains A(k)
  or B(k) and False otherwise */
10 # IsQuantum(A(_x)) <-- True;
10 # IsQuantum(B(_x)) <-- True;
    /* Result of a binary operation may
  be Quantum */
20 # IsQuantum(_x + _y) <-- IsQuantum(x)
  Or IsQuantum(y);
20 # IsQuantum(+ _y) <-- IsQuantum(y);
20 # IsQuantum(_x * _y) <-- IsQuantum(x)
  Or IsQuantum(y);
20 # IsQuantum(_x ** _y) <-- IsQuantum(x)
  Or IsQuantum(y);
    /* If none of the rules apply, the
  object is not Quantum */
30 # IsQuantum(_x) <-- False;
```

Now we shall construct rules that implement reduction to the canonical form. The rules will be given precedences, so that the reduction proceeds by clearly defined steps. All rules at a given precedence benefit from all simplifications at earlier precedences.

```
    /* First, replace * by ** if one of the
    factors is Quantum to guard against
user error */
10 # (_x * _y)_(IsQuantum(x) Or
  IsQuantum(y)) <-- x ** y;
    /* Replace ** by * if neither of the
  factors is Quantum */
10 # (_x ** _y)_(Not(IsQuantum(x) Or
 IsQuantum(y))) <-- x * y;
    /* Now we are guaranteed that ** is
  used between Quantum values */
    /* Expand all brackets involving
  Quantum values */
15 # (_x + _y) ** _z <-- x ** z + y ** z;
15 # _z ** (_x + _y) <-- z ** x + z ** y;
```

```
    /* Now we are guaranteed that there are
  no brackets next to "**" */
    /* Regroup the ** multiplications
  toward the right */
20 # (_x ** _y) ** _z <-- x ** (y ** z);
    /* Move classical factors to the left:
  first, inside brackets */
30 # (x_IsQuantum ** _y)_(Not(IsQuantum(y)))
  <-- y ** x;
    /* Then, move across brackets:
  y and z are already ordered
      by the previous rule */
    /* First, if we have Q ** (C ** Q) */
35 # (x_IsQuantum ** (_y ** _z))
  _(Not(IsQuantum(y))) <-- y ** (x ** z);
    /* Second, if we have C ** (C ** Q) */
35 # (_x ** (_y ** _z))_(Not(IsQuantum(x)
  Or IsQuantum(y))) <-- (x*y) ** z;
```

After we execute this in Yacas, all expressions involving additions and multiplications are automatically reduced to the canonical form. Extending these rules to subtractions and divisions is straightforward.

8.5 Implementing commutation relations

But we still haven't implemented the commutation relations. It is perhaps not necessary to have commutation rules automatically applied at each evaluation. We shall define the function `OrderBA()` that will bring all B's to the left of all A's by using the commutation relation. (In physics, this is called "normal-ordering".) Again, our definition will be recursive. We shall assign it a later precedence than our quantum evaluation rules, so that our objects will always be in canonical form. We need a few more rules to implement the commutation relation and to propagate the ordering operation down the expression tree:

```
    /* Commutation relation */
40 # OrderBA(A(_k) ** B(_l))
  <-- B(l)**A(k) + delta(k-l);
40 # OrderBA(A(_k) ** (B(_l) ** _x))
  <-- OrderBA(OrderBA(A(k)**B(l)) ** x);
    /* Ordering simple terms */
40 # OrderBA(_x)_(Not(IsQuantum(x))) <-- x;
40 # OrderBA(A(_k)) <-- A(k);
40 # OrderBA(B(_k)) <-- B(k);
    /* Sums of terms */
40 # OrderBA(_x + _y) <-- OrderBA(x)
  + OrderBA(y);
    /* Product of a classical and
  a quantum value */
40 # OrderBA(_x ** _y)_(Not(IsQuantum(x)))
  <-- x ** OrderBA(y);
    /* B() ** X : B is already at left,
  no need to order it */
50 # OrderBA(B(_k) ** _x)<-- B(k)
  ** OrderBA(x);
    /* A() ** X : need to order X first */
50 # OrderBA(A(_k) ** _x) <-- OrderBA(A(k)
  ** OrderBA(x));
```

These rules seem to be enough for our purposes. Note that the commutation relation is implemented by the first two rules; the first one is used by the second one which applies when interchanging factors A and B separated by brackets. This inconvenience of having to define several rules for what seems to be "one thing to do" is a consequence of tree-like structure of expressions in Yacas. It is perhaps the price we have to pay for conceptual simplicity of the design.

8.6 Avoiding infinite recursion

However, we quickly discover that our definitions don't work. Actually, we have run into a difficulty typical of rule-based programming:

```
In> OrderBA(A(k)**A(l))
Error on line 1 in file [CommandLine]
Line error occurred on:
>>>
Max evaluation stack depth reached.
Please use MaxEvalDepth to increase the
  stack size as needed.
```

This error message means that we have created an infinite recursion. It is easy to see that the last rule is at fault: it never stops applying itself when it operates on a term containing only A's and no B's. When encountering a term such as `A()**X`, the routine cannot determine whether `X` has already been normal-ordered or not, and it unnecessarily keeps trying to normal-order it again and again. We can circumvent this difficulty by using an auxiliary ordering function that we shall call `OrderBAlate()`. This function will operate only on terms of the form `A()**X` and only after `X` has been ordered. It will not perform any extra simplifications but instead delegate all work to `OrderBA()`.

```
50 # OrderBA(A(_k) ** _x) <-- OrderBAlate(
  A(k) ** OrderBA(x));
55 # OrderBAlate(_x + _y) <-- OrderBAlate(
  x) + OrderBAlate(y);
55 # OrderBAlate(A(_k) ** B(_l)) <--
  OrderBA(A(k)**B(l));
55 # OrderBAlate(A(_k) ** (B(_l) ** _x))
  <-- OrderBA(A(k)**(B(l)**x));
60 # OrderBAlate(A(_k) ** _x) <-- A(k)**x;
65 # OrderBAlate(_x) <-- OrderBA(x);
```

Now `OrderBA()` works as desired.

8.7 Implementing VEV()

Now it is easy to define the function `VEV()`. This function should first execute the normal-ordering operation, so that all B's move to the left of A's. After an expression is normal-ordered, all of its "quantum" terms will either end with an $A(k)$ or begin with a $B(k)$, or both, and `VEV()` of those terms will return 0. The value of `VEV()` of a non-quantum term is just that term. The implementation could look like this:

```
100 # VEV(_x) <-- VEVOrd(OrderBA(x));
    /* Everything is expanded now,
  deal term by term */
100 # VEVOrd(_x + _y) <-- VEVOrd(x)
  + VEVOrd(y);
    /* Now cancel all quantum terms */
110 # VEVOrd(x_IsQuantum) <-- 0;
    /* Classical terms are left */
120 # VEVOrd(_x) <-- x;
```

To avoid infinite recursion in calling `OrderBA()`, we had to introduce an auxiliary function `VEVOrd()` that assumes its argument to be ordered.

Finally, we try some example calculations to test our rules:

```
In> OrderBA(A(x)*B(y))
Out> B(y)**A(x)+delta(x-y);
In> OrderBA(A(x)*B(y)*B(z))
Out> B(y)**B(z)**A(x)+delta(x-z)**B(y)
  +delta(x-y)**B(z);
In> VEV(A(k)*B(l))
Out> delta(k-l);
In> VEV(A(k)*B(l)*A(x)*B(y))
Out> delta(k-l)*delta(x-y);
In> VEV(A(k)*A(l)*B(x)*B(y))
Out> delta(l-y)*delta(k-x)+delta(l-x)
  *delta(k-y);
```

Things now work as expected. Yacas's `Simplify()` facilities can be used on the result of `VEV()` if it needs simplification.

Chapter 9

GNU Free Documentation License

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.

51 Franklin Street, Fifth Floor
Boston, MA 02110-1301
USA

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

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

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

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

Applicability and Definitions

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

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

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

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

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

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

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

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

Verbatim Copying

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

You may also lend copies, under the same conditions stated

above, and you may publicly display copies.

Copying in Quantity

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

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

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

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

Modifications

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

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.
9. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
11. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
13. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
14. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

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

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

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

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

Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

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

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

Collections of Documents

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

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

Aggregation With Independent Works

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

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

Translation

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

Termination

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

Future Revisions of This License

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

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

ADDENDUM: How to use this License for your documents

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

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

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

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