(IN PROGRESS) Lisp as an Implementation Language for Computer Algebra Systems

by the YACAS team [1]

YACAS version: 1.0.57
generated on November 28, 2006

Lisp is a very simple language, one suited for doing computer algebra. Almost all CAS systems today owe a lot to the ideas that came from Lisp. This is a book on the Lisp language, and its connection to Yacas. Yacas is built on top of a Lisp dialect.

Contents

Chapter 1

A brief introduction to Lisp

1.1 Introduction

Yacas owes much of its current form to Lisp. A lot of ideas from Lisp have been taken over into the Yacas interpreter. This book aims to describe this connection, to show how Yacas compares to other `Lisp` dialects (YACAS can be viewed as a dialect of Lisp), and how the YACAS interpreter is related to Lisp.

Viewed from the top level, YACAS is just one function; YACAS is called with one, or possibly two arguments, or from a different point of view, a request is made to the YACAS system. How the function performs the action is implementation-dependent, but at the top-level it can be viewed as one call. From c++ or Java, it might look like

```
yacas.Evaluate("expression");
```

Or from Lisp,

```
(yacas expression environment)
```

The `environment` argument is optional; it can be a global resource that doesn't have to be passed in.

For instance, taking the derivative of a function could look like

```
yacas.evaluate("D(x)Sin(x);");
```

when in c++, or

```
(yacas '(D x (sin x)) )
```

when in Lisp. The quote in Lisp makes it clear that the argument, the parameter to the request, is actually data, and not program. This is an important distinction which has to be made clear in Lisp, but is clearer in other environments, due to the form of data being distinct from the form of program.

To drive home this point, `(* a 1)` would raise an error when given to the Lisp `eval` function if the variable `a` is not bound to a value, but would be perfectly valid data when passed to a CAS. In fact, one would like it to return `a` in this case. The `*` operator is actually a function call when invoked from Lisp (`(* a 1)` is a *program*), but is data when passed to a CAS.

One can conclude that YACAS, an engine for simplifying symbolic expressions and performing calculations, is just one huge function that performs that task, and has one entry point, a function call with one argument as the data to the request.

Of course this is a bit of an oversimplification. We are not just interested in evaluating and simplifying expressions. When Yacas is embedded inside an application that offers a user interface, one would want to have access to all the operations allowed, and their arguments. Furthermore, for a debugger, information about the internal state of the interpreter is needed (values of global variables, etc.).

At the end of the day we want to arrive at a language where we can implement an algorithm in a few lines of code, and have the compiler generate efficient code for it.

Given the strong typed Lisp we are using here, how is it different from using c++/Java directly?

The difference is in the syntax. Algorithms for manipulating symbolic expressions are written down much more naturally in a language that supports these operations natively. The code to perform symbolic manipulation operations can look surprisingly complex when written in c++ or Java. Later on, a compiler for the Lisp dialect developed here that will try to compile to human-readable source code will be described. Although the results are somewhat readable, they are a lot less readable than the original Lisp or Yacas script code they were generated from.

Convenient notation aids the programmer greatly in thinking about a problem. It is not unusual in mathematics to devise a new notation scheme to make life easier. For example, the bra ket notation, where `<a|b>` stands for the inproduct defined as

```
Integrate()a*b
```

has the added advantage of being able to write down projection operators of the form `|b><a|` which would be a lot more cumbersome to deal with without this notation.

Flexibility is not lost in a strong typed system; one can define a type `EXPRESSION`, which (naturally) represents an expression that can be manipulated. This could even be the default type for variables for which types are not specified, to make life easier for the programmer. This is not advisable, though, as the section describing the compiler will make clear.

This section will first introduce the reader to the basic concepts of Lisp, and then proceed to show how the `yacas` function can be implemented in Lisp in general. Clearness of concepts explained will be favored over efficiency and performance. A small subset of Lisp will be used, so it should be easy to take the code along to various different Lisp dialects, or to write a little Lisp implementation to implement the `yacas` function.

The first chapter briefly describes a small Lisp dialect. A small subset of what is usually found in the average Lisp interpreter is described. The Lisp dialect is different in small subtle ways from normal Lisp implementations. Simplicity is put first.

For the Lisp dialect presented, one was chosen that is different from the Lisp dialect that represents YACAS. This separation is done explicitly, on purpose, to facilitate implementation of execution of algorithms written out in YACAS syntax from any other language. All the YACAS-specific details have to be handled by the routine performing these algorithms, independent of language constructs that might support it.

Even though in Lisp program and data look the same, for YACAS they are distinctly different in that data will likely not be evaluated by the default evaluator. It is important to keep the distinction between data and program.

1.2 Why choose Lisp over other languages?

Lisp has some powerful options not available in for instance c++/Java. The syntax is closer to the way a human thinks about transforming expressions, and the resulting routines are thus shorter and easier to read. The same routine, when written out in Lisp or in c++, is much more readable and natural in Lisp than in c++, for instance. c++ syntax is made for other purposes.

Also, the typed Lisp dialect developed in this book is easily translated to other languages. This removes the argument of Lisp being less efficient; the code can be converted to efficient c++/Java code. What is more, the code is easy to parse and transform into these languages, giving a language-independent specification of algorithms.

1.3 A little history of Lisp

Lisp is one of the older computer programming languages in existence. It came from a mathematical background, lambda calculus, and is used even now. Arguably, most programming languages owe a lot to Lisp.

There are various sources on the internet describing the early history of Lisp.

1.4 The Lisp interactive command line interface

When Lisp is started, the user is generally greeted with a prompt. The Lisp interpreter is in its `read-eval-print` loop; it reads in an expression typed in by the user, evaluates it, and prints the result of evaluation to the console.

The definition of an expression can be given such;

1. an expression is either an atom, or a list of expressions.
2. an atom is a word, a list of characters.

So, the words `foo` and `bar` could be called atoms. A list is represented by writing the expressions in the list out with spaces in between them, and brackets around them. So, `(foo bar)` would be a list with two elements, `foo` and `bar`.

In Lisp, each expression can evaluate to a value (if no error occurred). When evaluating an atom, the atom can either be a 'special' atom that evaluates to itself, or otherwise a lexical word that refers to a variable. When it is a variable, the Lisp system looks up its value, and returns the value the variable is bound to, in what is called the 'environment' the Lisp interpreter is evaluating in. When a variable is not bound, an error is raised. The noted difference with the usual Lisp dialects is that a normal Lisp system raises an error when an expression tries to evaluate a variable that is not bound to a value, whereas in a CAS, the variable is a free parameter in some expression.

When a list of expressions is encountered, the first element in the list is a function that should be called, and the rest of the expressions in the list are the arguments. The interpreter handles the list by first evaluating the arguments (the rest of the list), and then calling the function with the evaluated arguments as parameters to the function. There are exceptions to this; the function `quote` returns its argument unevaluated. In stead of writing `(quote <expression>)`, a common short-hand is `'<expression>`, eg. the expression with a quote in front of it. Quotes are used a lot, to tell the system that data is being passed in.

For the following example interaction, the following commands are used:

- `quote` - `quote` takes one argument and returns it unevaluated
- `eval` - `eval` takes one argument, which is evaluated before `eval` is called, and then `eval` evaluates it again. `eval` can be used to oppose `quote`.
- `setf` - `setf` takes two arguments, the first a name of a variable, the second an expression to be evaluated. The result of evaluating the second argument is assigned to the variable.

The following is an interaction with our little lisp dialect. In the usual Lisp dialects, when variables are unbound, an error is raised.

```
In> a
Error, variable has no value : a
```

Lisp dialects are usually case-insensitive (the atom `a` is the same as `A`).

Here the atom `a` evaluated to itself, because it was not assigned.

```
In> (setf a '(this is a list))
Out> (this is a list )
In> a
Out> (this is a list )
```

Here we assigned the value `(this is a list)` to the variable `a`, and `a` now evaluates to that list. The system looked up the value bound to the variable `a` without re-evaluating it.

```
In> 'a
Out> a
In> (eval a)
ERROR: function not defined : this
Out> nil
```

With `quote` we can withhold evaluating a, and with `eval` we can re-evaluate `a`. Since `this` is not a defined function in this case, an error is raised.

Note that evaluation as defined in Lisp dialects is less suitable for CAS systems. A CAS would return expressions unevaluated, which is what you want in a CAS. It then represents some function which has not been specified yet. In a Lisp intepreter, it is nice to have the interpreter raise an error, but in a CAS, where it is data, it should not be simplified further.

For atoms that refer to variables that are not yet bound, the variables should be returned unevaluated by default, and functions that are called but not defined should be returned as the same function but with their arguments evaluated. evaluation as defined in most Lisp dialects is not suited to simplifying expressions. Later in this book an evaluator will be developed in Lisp itself that is closer to a sense of 'simplification' required for CAS.

Sometimes the user wants to refer to a function without having to define it, because transformation rules for the function can be given without explaining explicitly how to go about evaluating the function. Certain functions have properties, and sometimes it is easier to work with the properties than to evaluate them explicitly.

For the rest of this book, it is assumed that the semi-colon, `;`, means that the rest of the input line up to the end of line or end of file is a human readable comment, not to be executed.

1.5 A basic set of Lisp primitive commands

Surprisingly few Lisp functions are needed to be able to implement any algorithm in Lisp. Languages are usually referred to as being *Turing complete* if any algorithm can be implemented in them. Turing showed that a small apparatus, called a Turing machine, was universal in that theoretically any algorithm could be implemented in it. The way to show a language is Turing complete is to implement a Turing machine in that language. if that is done, that language can execute any algorithm, as a Turing machine can be implemented in it, and Turing machines can execute any algorithm.

Note that building a Turing machine is just a theoretical exercise. Turing machines don't concern themselves with efficiency. Algorithms can be run, in finite time, but it will most certainly not be the most efficient way to execute the algorithm on a real computer.

Methods to make execution more efficient break down in two steps:

1. encapsulating a specific algorithm in a function call, and make define an efficient implementation of the function in the environment the Lisp interpreter is executing in
2. writing a compiler that converts algorithms stated in Lisp code to a version that is more efficient, more close to the hardware the program is running on

This section describes a minimum set of Lisp primitives needed for building a Turing machine, theoretically. After that, first numeric operations are introduced, before implementing a Turing machine. Fundamentally, the language does not need to support arithmetic in order to be able to implement any algorithm, as other constructs using lists can be used to represent and work with numbers, but not implementing it does make the effort rather complex.

The following sections will describe the following minimal set of commands;

1. macro-like commands `quote`, `if`, `cond`, `setf`, `while`, `progn`, `eval`, `defun` and `defmacro`.
2. function-like commands `assoc`, `first`, `rest`, `cons`, `atom`, `listp`, `list`, `equal`, `+`, `integerp`, `-` and `<`.

In the next sections, whenever a function call is explained, the arguments to the function will be delimited by `<` and `>`. Arguments are evaluated unless stated otherwise.

Naming conventions

The set of primitives described in this section are meant to be used as building blocks for building a final bigger system.

Controlling evaluation

To control whether evaluation is performed, the following commands are available:

1. `(quote <expression>)` - as described in the introduction above, `quote` takes one argument, and returns it unevaluated.
2. `(eval <expression>)` - `eval` also takes one argument, and evaluates it when `eval` is called. When `eval` is called, its argument was evaluated already.

Assigning variables and functions

In addition, functions and macros can be defined:

1. `(setf <variable> <expression>)` - to assign a variable, `setf` can be used. the `variable` argument is not evaluated before `setf` is called, but `expression` is.
2. `(defun <function-name> <argument-list> <expression>)` - define a function whose name is `function-name`, and a list of arguments `argument-list`. This function can then be called from anywhere. Its arguments are first evaluated, and a set of local variables (listed in the `argument-list` argument) are reserved, and bound to the values passed in.
3. `(defmacro <macro-name> <argument-list> <expression>)` - does the same as `defun`, but with the arguments not evaluated before evaluating the body `expression`. Also, the `expression` is evaluated from within the calling scope, so the `expression` can use variables from its calling environment. A better way to look at macros is as if they were copy-pasted into the place where they were called.

The default set of functions implemented in the core of a Lisp interpreter can be divided between function-like and macro-like functions, in that when a function-like (defun) function is called, its arguments are evaluated first. Examples of this are `cons`, `first` and `rest`.

When calling macro-like (defmacro) functions, however, the arguments are not evaluated first, and the variables defined in the calling environment are visible while expressions using arguments on the command line are evaluated. A macro can be viewed as a bit of code that got pasted in to the place where it got called, by the interpreter (in fact, this is what a compiler can do).

The `defmacro` function allows the programmer to expand the language by adding functions that act as if they were defined in the core part of the interpreter.

Macros are not necessary, strictly speaking. Macros clean up the syntax of a program, in that it allows a programmer to write more clean and brief code. For example, the expression

```
(if (equal a b) () (print a))
```

could be written more concisely as

```
(unless (equal a b) (print a))
```

if `unless` were defined. But it would have to be defined as a macro, as when `unless` is called, the arguments should not be evaluated first. Unless can be defined as:

```
(defmacro unless (predicate body)
  (if predicate nil body) )
```

This makes `unless` work, as the following interaction shows:

```
In> (unless t (print 'SOMETHING))
Out> nil
In>  (unless nil (print 'SOMETHING))
SOMETHING
Out> SOMETHING
```

Especially when code is generated, and does not have to be human-readable, macros are not necessary. But they are a nice feature nonetheless, and have to be implemented in the interpreter, it can not be faked with normal `defun` functionality.

Examples involving `defun`:

```
In> (defun foo (atom) (cons atom (list rest)))
Out> foo
In> (setf rest 'end)
Out> end
In> (foo a)
Out> (a end )
In> (foo b)
Out> (b end )
```

In addition, nameless (pure) functions can be made, which work exactly as their `defun`/`defmacro` counterparts, as `lambda` and `macro`:

1. `(lambda <arguments> <body>)` - define a pure unnamed function.
2. `(macro <arguments> <body>)` - define a pure unnamed macro.

Examples:

```
In> (setf add '(lambda (x y) (+ x y)))
Out> add
In> (add 2 3)
Out> 5
```

Note that pure functions, the `lambda` and `macro` expressions described here, are not strictly necessary, if we have equivalents for `defun` and `defmacro`. But pure unnamed functions do allow for convenient notation, without clobbering up the global name space. They also allow for locally defined functions. If they are required, they have to be part of the interpreter, since it cannot be faked with `defun` and `defmacro`.

An interesting option is to not define the operations `defun` and `defmacro`, but just `setf`, `lambda` and `macro`. This set is just as powerful, and in fact allows you to create `defun` and `defmacro` from them. We choose `defun` and `defmacro` over the route of using `setf`, `lambda` and `macro` to define `defun` and `defmacro` because `lambda` and `macro` are not strictly required to build a CAS. When implementing a more mature Lisp interpreter, it is wise to define `lambda` and `macro`, for the convenience of the programmer.

Creating and disseminating lists of expressions

Lists of expressions are used to build more elaborate structures. The following commands are available for doing so:

To disseminate a list:

1. `(first <list>)` - return the first element of a list
2. `(rest <list>)` - return the list with the first item removed.

Examples, taken from an interaction with `clisp`:

```
In> (first '(a b c))
Out> a
In> (rest '(a b c))
Out> (b c )
```

To reconstruct a list:

1. `(cons <first-expression> <rest-list>)` - `cons` returns a list that has ¡first-expression¿ as its first element, and ¡rest-list¿ as the rest of an expression.
2. `(list <...>)` - return a list with the ¡...¿ arguments evaluated.

Example taken from an interaction with `clisp`:

```
In> (cons 'a '(b c))
Out> (a b c )
In> (list (+ 2 3) 4 3)
Out> (5 4 3 )
```

One important function is `assoc`:

1. `(assoc <key> <assoc-list>)` - given an association list, which is a list of (¡key¿ ¡value¿) pairs, return the first found pair for which the key matches the one given as first argument, or return `nil` if it wasn't found.

Examples:

```
In> (setf list '( ("name" "Peter") ("age" 28) ) )
Out> list
In> (assoc '"name" list)
Out> ("name" "Peter" )
In> (assoc '"age" list)
Out> ("age" 28 )
In> (assoc '"something" list)
Out> nil
```

Predicates

Common Lisp has two special atoms, `t` and `nil`. `t` stands for *true*, whereas `nil` stands for *false*. `nil` is actually usually also the empty list. Typing an empty list `()` is the same as typing `nil`:

```
In> ()
Out> nil
In> nil
Out> nil
```

In Common Lisp, `nil` is considered to mean *false* when considered as a boolean result from a list, and anything different from `nil` is considered to mean *true*. This is then used as an efficient way to find items in lists. The result returned can then either be a found result, or `nil` which means nothing was found.

```
In> (rest '(a b))
Out> (b )
In> (rest '(a))
Out> nil
```

The author feels uneasy with having the `nil` atom overloaded as both an empty list and *false* when used in predicates, because it makes code harder take along to another language. In some cases `nil` means an empty list, and in some cases it means *false*, which complicates converting code, due to the fact that the code has to be examined more closely for this. YACAS has a distinct set of atoms, `True` and `False` precisely for this reason. A test to see if something is an empty list is still possible, by comparing the result of evaluating an expression with an empty list. Indeed, Yacas is not alone in this, Scheme (another dialect of Lisp) has different symbols for *true* and *false*, `#t` and `#f` respectively.

Two predicates needed for operating on lists are then:

1. `(listp <expression>)` - `listp` returns `t` if ¡expression¿ is a list, `nil` otherwise.
2. `atom <expression>` - `atom` returns `T` if the `expression` passed in is an atom (a word).
3. (equal ¡expression-1¿ ¡expression-2¿) - returns `t` if the two expressions are the same, `nil` otherwise. Common Lisp has more equality operators. They are more efficient for specific types of comparisons. `equal` is the most generic form which works on all types of input. As we are composing a minimum subset of functions to work with in this part of this book, we will only mention `equal`.

Note the difference in `atom` and `listp`, which was taken from Common Lisp. In Common Lisp predicates are supposed to be suffixed by a 'p', but apparently not so with `atom`.

Variable scoping

1. `(let <assign-list> <expression>)` - Variables can be declared locally in the function being called, using `Let`.

When `setf` is called, first the local variables are searched for a candidate variable to set, and after that the global variables are searched. When entering a function body, a local frame of local variables for that function are defined, and the parameter variables are bound to the arguments passed in to the function. `let` allows the programmer to declare additional local variables.

The following example sets up variables `n` and `m` and assigns them values locally. `n` is bound to `a`, and `m` bound to `(b c)`.

```
In> (let ( (n 'a) (m '(b c)) ) (cons n m))
Out> (a b c )
In> n
ERROR: variable not defined : n
Out> nil
```

The `expression` part in the `let` form gets evaluated after the local variables have been assigned. Note that evaluation of the expressions in the `let` form are performed in the scope where the `let` form is called, thus expressions in the `let` form can use local variables from its surroundings.

Local variables that are declared in a certain scope where the variable already had a value mask the older declaration of the variable.

Note: the values that will be assigned to the local variables are evaluated *before* being assigned to the local variables. So the expressions in the `assign-list` cannot refer to other variables declared in the same `assign-list`.

Execution control

No language is complete without conditional execution and looping.

1. `(if <predicate> <then-clause> <else-clause>)` - an if statement: if `predicate` doesn't evaluate to `nil` return the result of evaluating the `then-clause` expression, otherwise return the result of evaluating the `else-clause`.
2. `(while <predicate> <body>)` - this function evaluates `body` until `predicate` returns `nil` (`predicate` is called before each iteration), and the result of the last evaluation of `body` is returned afterwards.
3. `(cond (<pred-1> <body-1>) ... (<pred-n> <body-n>))` - similar to `cond` in Lisp: the items are traversed one at a time, from front to end, evaluating the predicates. if one does not evaluate to `nil`, its `body` is evaluated, and the result of evaluating returned.
4. `(progn <body-1> ... <body-n>)` - evaluate multiple expressions, and return the result of evaluating the last expression, `body-n`.

This concludes the a set of functions required for building up a computer algebra system. It is not a minimal set by any means, as a following section will show; `if` can be defined by using `defmacro` and `cond`, for instance.

The next section will first introduce arbitrary number arithmetic support. It is not strictly necessary for the interpreter to work and be able to do anything, but it is the first concession we will make towards efficiency, as it is rather hard to find an algorithm that does not need arithmetic.

The next sections after that will continue by introducing a few building blocks that are needed to implement a CAS like Yacas.

1.6 Arbitrary precision arithmetic

Before proceeding to build up a full system, we first introduce the arithmetic to the system. Arithmetic could theoretically be performed by using what we already have (lists of objects, where the number of objects in the list represents the number), but it would be impractical.

We start with the representation of numbers. This section will only refer to arbitrary precision integers, floating point numbers will come later. The implementation of the Lisp dialect described here is designed to be easily extended that way.

Numbers are represented as atoms, sequences of characters. More specifically, they can be a sequence of digits from `0` to `9` (we will work in the decimal number system here, as that is what most humans are used to). Negative integers have a minus sign in front of them. So, `1234` and `-1234` are atoms representing the integers 1234 and -1234 respectively.

The most basic arithmetic operations are then addition and negating a number:

```
In> (+ 1 2 3)
Out> 6
In> (- 5)
Out> -5
In> (+ 2 (- 3))
Out> -1
```

Together with a predicate to determine that something is an integer (`integerp`), and an ordering operator *less than* denoted `<`, other arithmetic operations like multiplication, division, exponentiation etc. can be built.

Some examples using `<` and `integerp`.

```
In> (integerp 4)
Out> t
In> (integerp -4)
Out> t
In> (integerp 'a)
Out> nil
In> Out> (a b c )
In> (integerp '(a b c))
Out> nil
In> (< 2 3)
Out> t
In> (< 3 2)
Out> nil
```

Note that a computer algebra system requires arbitrary precision arithmetic, as for certain algorithms the intermediate results of a calculation can become big, even if the final result is small. So, a CAS should be able to do the following:

```
In> (+ 12345678901234567890999999999999999999 1)
Out> 12345678901234567891000000000000000000
```

1.7 A few useful macros and functions

Now we are ready to create more powerful building blocks using the primitive set of functions introduced in previous sections.

The routines developed here will be built from the primitive building blocks described before. However, an implementation

is free to implement these hard-coded in the core kernel for speed. This is advisable because the constructs developed here will be used often, and so there is a need for efficiency.

Note that the functions introduced so far are not minimal. Some can be defined in terms of others. They are defined here because they are easy to define in an interpreter without making the interpreter much larger, but are used often, so efficiency is required. For instance, given the `cond` and `defmacro` operations, it is possible to define an `if` operation:

```
In> (defmacro Newif
          (pred then else)
          (cond
             ( pred then)
             ( t    else)
        ) )
Out> Newif
In> (Newif (equal 'a 'b) 'c 'd)
Out> d
In> (Newif (equal 'a 'a) 'c 'd)
Out> c
In> (if (equal 'a 'b) 'c 'd)
Out> d
In> (if (equal 'a 'a) 'c 'd)
Out> c
```

Now let us start by building a set of new primitives using the ones introduced so far. We start by defining `Null`, which returns True if its argument is an empty list:

```
In> (defun null (expr) (equal expr nil) )
Out> null
In> (null 'a)
Out> nil
In> (null ())
Out> t
In> (null nil)
Out> t
```

With this, we can define a `Length` function which returns the length of a list:

```
In> (defun length (list)
  (if (null list)
    0
    (+ 1 (length (rest list)))
) )
Out> length
In> (length '(a b c))
Out> 3
In> (length ())
Out> 0
```

In addition to the already defined function `first`, we can define the functions `second`, `third`, etc. to print code that is easier to read:

```
In> (defun second (list) (first (rest list) ) )
Out> second
In> (defun third  (list) (first (rest (rest list) ) ) )
Out> third
In> (setf lst '(a b c))
Out> lst
In> (first lst)
Out> a
In> (second lst)
Out> b
In> (third lst)
Out> c
```

and with this we can start to print functions for boolean expressions. First, let us define a `not` operator, which returns True if its argument is `nil`:

```
In> (defmacro not (a)  (null a) )
Out> not
In> (not 'a)
Out> nil
In> (not nil)
Out> t
```

Then *lazy and* and *lazy or* operations can be defined:

```
In> (defmacro and (a b) (if a b nil))
Out> and
In> (defmacro or  (a b) (if a t b) )
Out> or
```

These evaluate their first argument first, and decide to evaluate the second only if the first one does not give enough information to decide whether to stop. For instance, if the first argument to `and` returns *false* (eg. `nil`), then the result of evaluating the entire expression should be `nil`, otherwise it returns the result of evaluating the second argument. `or` works the other way round: it evaluates the second argument if the result of evaluating the first argument was `nil`.

Lazy and and or operators can be used to combine predicates into a larger predicate, but also to guard evaluation of an expression, let it only evaluate an expression if a pre-condition is met. The `unless` operation could equally well have been defined by using `or`:

```
In> (defmacro unless (predicate body)
  (or predicate body) )
Out> unless
In> (unless t (print 'SOMETHING))
Out> nil
In> (unless nil (print 'SOMETHING))
SOMETHING
Out> SOMETHING
```

Most of these functions are convenience functions. They make code easier to read. Macros in effect allow you to extend the language.

1.8 A reality check: implementing more functions in the interpreter for efficiency

Even though new functions *can* be created in Lisp code rather than the interpreter, it is not necessarily a good idea to do so. For instance, declaring additional local variables can be done with `defmacro` (and in this case `macro` would also be necessary). However, it is used a lot in Lisp code, so it makes sense to write an efficient version in the interpreter itself. This section describes a few commands that, although not strictly necessary, should be implemented in the interpreter for efficiency of execution.

The reasons for adding these functions fall into the following categories:

1. routines that make for easy debugging (example: `print`).
2. routines that can be implemented a lot more efficiently in the interpreter than using the base Lisp commands (example: `let`).

3. routines that are already implemented in the interpreter any way (examples: `second`, `third`, `null`, `eq`). It is a pity if these were not used while they are already implemented efficiently and available.
4. commands related to user interfacing. An example is `exit`, which is not strictly needed, but facilitates quitting the command line version of the interpreter. These should be implemented in the code that is only used in the console version. `exit` makes less sense in a graphical user interface, where stopping the application is likely to be performed in different ways. `exit` is not strictly a function needed for CAS.

The commands described here can be implemented in the interpreter, but they are not strictly necessary, or they can be implemented in Lisp code using more basic primitives.

1. `(let ( (<var-1> <val-1>) ... (<var-n> <val-n>) ) <body>)` - evaluate the values `val-1` through ¡val-n¿ and declare new local variables and then assign the values to the new local variables. Note that the values are evaluated *before* assigning them to the variables, so the variables can not depend on each other.

Chapter 2

Language design issues

This documentation comes with some implementation for the little Lisp dialect introduced in this document. It is implemented in `Java`. This section discusses some details pertaining to the implementation. It will discuss some features of the language design.

2.1 Enforcing correct code

Programmers are lazy. They will try to get as much as possible done by typing as little as possible. This can be used as an advantage in the design of a programming language. If the programming language makes it easy 'to do it the right way', but hard 'to do it the wrong way', the programmer might take the easy route and do it right. The culmination of this is to make it possible to do it right, but impossible to do wrong. The language `c` has many examples of the opposite, where it is easy to do something wrong, but hard to do it right. The functions `gets` and `fgets` are examples.

From the other side, exceptions in Java for instance, can not be ignored. A routine has to handle it, or promise to pass off the exception to the calling environment. But then all the functions calling the function must handle the exception, resulting in much more typing work.

This concept can be put to great use in for instance a typing system, where it can be made very hard to pass an arbitrary-type object around, like the `void *` of `c` or `c++`, or a `GenericLispObjectHugeTypeDeclarator` in a Lisp system, where type declarations can be enforced. In addition to this, it can be made harder to pass around wrong types by forcing hand-written type-conversions. An object of a generic type can be passed in, but if it results in considerable extra effort on the behalf of the programmer to do type conversions in the code, the programmer might consider declaring the object of the correct type in the first place.

C++ does this the other way round; the language makes it particularly easy to do automatic type conversions between types, sometimes resulting in dangerous code.

2.2 Containers and iterators

In the `c++` standard library there is a powerful notion of *containers* and *iterators*. The idea is to hide how objects are stored in a structure, and offer a generic way to access objects in a store of objects. General algorithms that work on sets of data can then be defined in terms of this unified interface. The object that gives access to an element in a container is called an *iterator*. The typical things needed are:

1. get the first object in a container
2. get the next object in a container
3. check if there are still objects in the container

A lot of algorithms working on data can get away with this sequential processing, performing an operation on an object, and then moving to the next object to perform the same operation on it.

The classic way in Lisp has been to have a list of objects, and have the `first` function return the first of the list, `rest` return the rest (the objects that still need processing), and when the list is empty, all objects have been processed.

The reason to make this abstraction is that for specific cases optimizations can be made in the way the container is implemented internally. For instance, in stead of a list, an array could be used, which gives constant-time access to an element in a container, but slow O(n) insertion time. Many more options are available; objects could be obtained from a compressed block of data, or read in sequentially from disk (for huge sets of objects), or a set of objects could be defined symbolically, like `(range 1 Infinity)`. A routine could then be written that finds the first occurrence of an integer that has certain properties, for instance. One would need to have a function `first` and `rest` and `null` for arrays and for objects of the form `(range <from> <to>)`.

This is an important issue, because computers are good at doing a dull task, but doing it millions of times over. One option is to have a huge data set and process it. An efficient way of walking over a sequence of data objects is then necessary.

2.3 Packaging modules

Modules are essentially groups of functions and 'global' variables that can only be accessed from within that environment. It is an important feature in a language that hopes to stay modular while the software written in it grows to a sizable number of separate files.

Packaged environments are similar to the well-known singleton classes from c++, which are in turn again similar to global functions and variables. The disadvantage of having things global is mainly that you can only have one instance of the global state of that module. This is not a problem for modules for which there is naturally only one instance.

Packaged modules are usually of this form. They can be seen as a name space, where certain symbols are defined to have a meaning locally. Specific symbols from a module can be exported to another module, thus the symbol of another module becomes visible in another name space.

As mentioned above, modules in the Lisp dialect developed here have a very simple structure; they are essentially what is usually referred to as a class, but there will be only one instantiation of each module, and modules are not 'derived' in any form from other modules. Modules can export functions (or

more precisely, modules can import functions from other modules) when appropriate, but this feature should be used with care as this easily results in re-introducing name collisions in the name space of a module.

Modules have their own global variables. When a function from a module is invoked, it can only see its own globals, even if exported to another environment. This has the following advantages:

1. It enforces modularity, by disallowing modules to modify globals from other modules directly.
2. It facilitates compilation to efficient code when in an object-oriented environment; the module just becomes a class with methods and properties, with a one-on-one relation between the functions, global variables defined in the module, and the properties and methods defined in the compiled class.

When well-thought out, packages can greatly aid in making a system modular, built from small sub-modules which can be added or removed.

Functions supporting modularization

The essential functions for modularization are `namespace`, `import`, `makeenv` and `functionp`. These functions will be described in this section.

An environment can be built up with `makeenv`, and defining the functions and global variables contained therein:

```
(makeenv arithmetic
   (progn
      (defun add (n m) ...)
      (setf pi 3.1415926535)
)  )
```

An expression can then be evaluated in this environment through:

```
(namespace <module-name> <function-name> <arguments...>)
```

The 'environment' can be thought of as a static class, a singleton, in object-oriented terms. There is only one of this type of object, and one instance of each property in the class, which is why the compiler can generate a static link to it. A function in an environment is no more than a method in such a singleton class.

The arithmetic operations like `+` are actually defined in the module `INTEGER` for the reference implementation of the Lisp interpreter described in this book. This means that from start up, the `+` operator is not directly accessible. One way to access a function in a module is to pass the name of the module as an argument to the operator `namespace`:

```
In> (+ 2 3)
ERROR: function not defined : +
Out> nil
In> (namespace 'INTEGER '+ 2 3)
Out> 5
```

This is obviously not the most convenient syntax. One way to encapsulate a call to a function in a module is to define a function for it:

```
In> (defun +(a b) (namespace 'INTEGER '+ a b))
Out> +
In> (+ 2 3)
Out> 5
```

This way we have back our easy access to the functionality of adding two integers. However, we are left with cumbersome syntax for importing this function. For this the function `import` was defined, which makes importing a function from another module into a module a little bit easier:

```
In> (import INTEGER + AddInt)
Out> t
In> (AddInt 2 3)
Out> 5
```

Note that for this language we chose the construct of `namespace` because we don't want access to global variables in the module. Another option would have been to allow passing an environment, or module name to `eval`, as in `(eval <expression> <module-name>)`. This would have allowed the programmer to pass in any expression to the environment, effectively making it possible to evaluate variables in that module. We want to actively disallow that, by only allowing function calls to that environment. The arguments to the function call are evaluated before the function defined in the module is called, thus they are evaluated in the calling environment.

To define a module, `makeenv` can be used. `makeenv` takes two arguments, a module name and the body to execute to define globals and functions inside it. For example, we can define a function `bar` in the module `foo` as follows:

```
In>  (makeenv foo (defun bar(n m) (list 'head n m)))
Out> bar
In> (bar 2 3)
ERROR: function not defined : bar
Out> nil
In> (import foo bar bar)
Out> t
In> (bar 2 3)
Out> (head 2 3 )
In> (namespace 'foo 'bar 2 3)
Out> (head 2 3 )
```

The function `functionp` allows one to check that a function is defined in a package. This is useful for checking that a package name that is passed as an argument is a valid package (has the correct functions implemented). The syntax is `(functionp <namespace> <functionname>)`:

```
In> (functionp 'std '+)
Out> nil
In> (functionp 'INTEGER '+)
Out> t
In> (import INTEGER + +)
Out> t
In> (functionp 'std '+)
Out> t
```

The default environment (module) the application starts in, the one the user can access, is the name space `std`. Other name spaces are `INTEGER`,

Note that `INTEGER` is both the name of a name space and of a type. This has powerful consequences: of we define a vector to be a vector of integers, as some structure with `VECTOR INTEGER` in it, the code to add two vectors knows that the components of the vector can be added by using the addition operation from the `INTEGER` module directly, resulting in speedier code than code that would accept the components as being of type `OBJECT`, and figuring out for each component that it was an integer.

The system can even go as far as checking that a parametric type can be made by checking that certain operations are defined for it, through the `functionp` call.

The bigger picture

The programming language does not stands on itself. It offers a way to execute certain algorithms, through the language offered, but also there are interfaces to the environment (usually the operating system). As such, from within the programming environment, there are functions that are supported that allow for operating system dependent operations, like file manipulation and using libraries commonly found on systems.

As such, the programming language also offers an interface to operating system functionality through modules supporting these operations.

Static versus dynamic scoping

Dynamic scoping is not supported implicitly. To access a function or object from an environment, the environment needs to be specified explicitly. This is important because the compiler needs to be able to determine where to find the information at compile time. The reference to the function or variable needs to be hard-coded at compile time. Having to look up the symbol at run-time would simply be an unnecessary performance penalty.

2.4 Default modules defined in the Lisp environment

All modules have access to the following functions and macros by default:

quote, if, cond, setf, while, progn, defun, defmacro, assoc, eval, first, rest, cons, atom, listp, list, equal, namespace, import, makeenv, functionp.

All other commands (including arithmetic commands) can be accessed through other modules.

For modules representing some type, usually there should at least be a `type` function which returns `t` if an object is of that type, `nil` otherwise, and a function `make` which makes such an object given some arguments (if applicable).

The `INTEGER` module

The `INTEGER` module is one of the few that should be implemented in the interpreter, as it is probably most efficiently implemented through some library. It supports the following commands:

1. `(+ n m)` - addition of integers, returns integer
2. `(- n)` - negation of an integer, returns integer
3. `(* n m)` - multiplication of integers, returns integer
4. `(div n m)` - division of integers, returns integer `q` for which $n = qm + r$, $0 \leq r < m$
5. `(mod n m)` - remainder after division of integers, returns integer `r` for which $n = qm + r$, $0 \leq r < m$
6. `(gcd n m)` - greatest common divisor of two integers
7. `(< n m)` - 'less than' operator for integers, returns boolean
8. `(integerp n)` - returns `t` if n is an integer, `nil` otherwise
9. `(type n)` - returns `t` if n is an integer, `nil` otherwise.

2.5 Error handling

Error handling is done by using exceptions. A function `preEval` is the entry point for evaluating expressions, and can be called from the user interface part. While performing the command, functions at any depth can throw an exception, accompanied with information on what went wrong. `preEval` catches the exception, and shows the error.

It is defendable to use exceptions, as the problem is likely to be invalid input to a function, and as a result invalid input passed in by the user, and as such an environment issue that the code cannot be expected to treat correctly.

Exceptions halt execution immediately. The `preEval` catches it, and the user interface can continue where it left off. This is defendable behavior. The system is only expected to do online commands typed in by the user, or fail. The story would be different if YACAS were used to control some system, and it needs to keep running after failure, but it was not designed for that. YACAS is a computer algebra system, not a controlling system for a Space Shuttle.

2.6 Type systems

Even though the Lisp interpreter discussed so far does not support typing, when we discuss compiled Lisp typing is considered, and thus the subject warrants some discussion.

The Lisp interpreter described so far is untyped, meaning that everything is of one type, `OBJECT`. What happens in fact is that the interpreter executes functions, and these functions internally do conversions. When calling an arithmetic operator like `+`, arguments of type `OBJECT` are converted to `NUMBER`, the addition is performed, and the result is converted back from type `NUMBER` to `OBJECT`. Untyped systems are inefficient this way.

The compiled Lisp dialect that will be developed will allow for typed arguments and functions; functions can only accept arguments of a certain type, and return a result of a specific type. The typed Lisp dialect will still allow for untyped (eg. `OBJECT`) arguments, for the lazy programmer (but as we shall see later, the lazy programmer is in for a few nasty surprises: slower code and more typing required, as a lot of manually coded type conversions will need to be made). With the simple typing system we will use, no automatic type conversions will be made. They will have to be done manually.

The system will consist of various packages involving operations on objects. For instance, there will be a package that performs addition on integers and real numbers, and there will be a package that deals with addition of vectors and matrices. The same will hold for for instance complex numbers. The addition operator in the global environment will then be responsible for dispatching an addition operator to the appropriate addition operator in the appropriate environment. For instance, there can be an `AddIntegers` operator, and an `AddComplexNumbers` operator, and the invoking addition operator can first check argument types before invoking the correct operator. This is called data-directed programming.

Complications arise when two objects of different types are added. When adding an integer to a real number, one cannot use the operators for adding integers or real numbers themselves. One solution for this is to first convert the integer to a real number type, and then invoke the addition of real numbers operator. For numeric types, this works well. Numbers can be put in what is called a tower of types: integers are a sub-type of rational numbers, which are a sub-type of real numbers, which are a sub-type of complex numbers. A number can always be moved up in the tower of types. An integer is just a special case of a complex number, for instance. So when adding an integer to a complex number, one only needs to convert from the integer to complex. This can be done in steps, so for `n` types only `n-1` convertors are needed; `IntegerToRational`, `RationalToReal`, etc.

Typing systems are not always this simple. For example, a diagonal matrix is a sub-type of the generic set of all matrices, but some diagonal matrices are also sub-types of the unitary matrices. So a matrix can have more than one super-type. Super-types can have multiple sub-types too. In this case, choosing the right conversion is not trivial. The choice can be based on some heuristic; when doing a specific type of operation, first try to convert the matrix to a type for which there is the most efficient algorithm for performing that action, otherwise try another type.

It gets worse; for polynomials, $x^2 - y$ can be a multivariate polynomial in `(x,y)` with integer coefficients, or a univariate polynomial in `x` with coefficients which are univariate polynomials in `y`, or a univariate polynomial in `y` with coefficients which are univariate polynomials in `x`. There might be cases where the system can not decide which type to take, and the user will have to help the system by telling what type to take.

Types can be down-cast to sub-types too, if possible. This is sometimes necessary. Suppose one has two objects of type `OBJECT`, as entered by the user. The user wants to perform a division operation. The system can try to drop the object to the lowest possible type, by first trying to convert to integer, and if this doesn't work, see if the objects can be converted to objects of type univariate polynomial, and otherwise multivariate polynomial, etc. The trick is to have a routine that accepts univariate polynomials in internal representation, performs the operation, and returns the result of the operation in internal representation. When the operation is called with generic objects as arguments, if they can be converted to internal representation for univariate polynomials first, this can be done, and the result of the operation converted back to generic object type (a form the user could have typed in). The advantage of this is an object is converted once, when passed to a function. The result of consecutive function calls can then use the object in the type it is already in, without having to convert types each time, and the final result is converted back. This way types are only converted twice; at the beginning and at the end.

2.7 Data structures as types in Lisp

A nice way to work with data structures is to hide them behind what are called *constructors* and *selectors*. For example, suppose we need a type for complex numbers. We could make a function `make-complex`, which is a constructor, and `real-part` and `imag-part`. We don't need to discuss the internal representation of the complex number, as all we need to know is how to make one, and how to obtain all the information there is to know about the object.

Interestingly, often when dealing with mathematical objects, there are properties that always need to hold for such objects. In the case of `make-complex`, `real-part` and `imag-part`, for a complex number `z`, the following should always be true:

```
(equal z (make-complex (real-part z) (imag-part z)))
```

Data structures are a powerful way to abstract types away. A data structure can be defined for for instance rational numbers, matrices, etc. Also `cons`, `first` and `rest` are such a combination, where `cons` is the constructor, and `first` and `rest` are the selectors. The result of evaluating `(cons a b)` is an object for which `first` returns `a`, and `rest` returns `b`.

Suppose we decide to represent complex numbers as `(complex real imag)`. Then the following routines are constructors and selectors:

```
(defun make-complex (r i)
  (list 'complex r i)
)
(defun real-part (z) (second z))
(defun imag-part (z) (third z))
```

Then this allows for the following interaction:

```
In> (setf z (make-complex 2 3))
Out> z
In> (real-part z)
Out> 2
In> (imag-part z)
Out> 3
```

We can define a predicate to check if an object is complex as:

```
(defun complexp(z)
  (and (listp z)(equal 'complex (first z)))
)
```

And with that a simple addition operation:

```
(defun add (x y)
  (cond
    ( (integerp x) (+ x y) )
    ( (complexp x)
      (make-complex
        (add (real-part x) (real-part y))
        (add (imag-part x) (imag-part y))
) ) ) )
```

which allows for the interaction:

```
In> (add (make-complex 2 3) (make-complex 4 5))
Out> (complex 6 8 )
```

This is an example of data-driven programming; invoking different routines for different data types. Also note we use the `add` operation for the components of the complex data object. This has powerful consequences: it means that a complex object can have components of arbitrary type, as long as there is an addition operation defined for it in the `add` function. The complex data object is in effect a parametric type: `(complex <type> <type>)`, where `<type>` can be anything, integers, matrices, etc. `<type>` is a type parameter for this complex data type.

Types in general aid in two ways: to check for correctness of code, and to give information to the compiler to generate more efficient code. This also holds for parametric types. Suppose we have a vector object. Pseudo-code for adding the two vectors could look like this:

```
vector add-vectors(v1,v2)
   vector result = zero-vector
   for i from 1 to degree(v1,v2)
      result[i] := add(v1[i],v2[i])
   return result
```

Note we use the `add` operation here for adding the components of a vector. Suppose the `add` operation was then defined as:

```
object add(x,y)
  if (object-type(x) = integer)
    return add-integers(x,y)
  if (object-type(x) = real)
    return add-reals(x,y)
  if (object-type(x) = complex)
    return add-complex(x,y)
  if (object-type(x) = vector)
    return add-vectors(x,y)
```

Now the important thing to notice is that the `add` function is called from within the loop of `add-vectors`, and if the vectors in question contain a billion items, `add` gets called a billion times. However, `add` checks the type of the object to be added each time it is called, and it is called from the inner loop! The `add-vectors` function can be written with the `add` operation inlined into it:

```
vector add-vectors(v1,v2)
   vector result = zero-vector
   for i from 1 to degree(v1,v2)
     if (object-type(v1[i]) = integer)
       result[i] = add-integers(v1[i],v2[i])
     if (object-type(v1[i]) = real)
       result[i] = add-reals(v1[i],v2[i])
     if (object-type(v1[i]) = complex)
       result[i] = add-complex(v1[i],v2[i])
     if (object-type(v1[i]) = vector)
       result[i] = add-vectors(v1[i],v2[i])
   return result
```

However, because all the type of all the vector components can be expected to be the same, this can be improved by pulling the if statements outside of the loop, so the if statements are only performed once in stead of a billion times. This is a common way to optimize code, and is often done by hand. There is no reason, however, why a computer could not do this optimization. When the `add` operation is inlined, if the compiler knows that the if statements will yield the same results every time, it can pull them outside of the loop, as follows:

```
vector add-vectors(v1,v2)
   vector result = zero-vector
   if (object-type(v1[i]) = integer)
     for i from 1 to degree(v1,v2)
       result[i] = add-integers(v1[i],v2[i])
   if (object-type(v1[i]) = real)
     for i from 1 to degree(v1,v2)
       result[i] = add-reals(v1[i],v2[i])
   if (object-type(v1[i]) = complex)
     for i from 1 to degree(v1,v2)
       result[i] = add-complex(v1[i],v2[i])
   if (object-type(v1[i]) = vector)
     for i from 1 to degree(v1,v2)
       result[i] = add-vectors(v1[i],v2[i])
   return result
```

If a compiler knows beforehand that two vectors of type `vector<integer>`, it can devise a special function that will only work on objects of type `vector<integer>`, but this is already a lot less useful. The if statements have already been pulled out of the loop, so they will not be performed a billion times for an addition of two vectors both with a billion components.

Parametric types can be supported naturally. A vector could be represented as `(vector <type> <components>)`, so that `(vector SMALLFLOAT (1 0 0))` would be a base unit vector in a three-dimensional space, with real-valued vector components. Flexibility is not lost, the type can be `EXPRESSION` or `OBJECT` or some other generic type. But it can also be `(UNIPOLY INTEGER x)`, eg. making the vector a vector with univariate polynomials in x over the integers as components.

The above considerably speeds up the operation. The problem is that it is more natural to place the if statements in the `add` function, so it can be used in many places, when adding complex numbers, matrices, etc. But inlining it allows the compiler to optimize the if statements away by pulling them outside of the loop. In most cases, even if not in a loop, the compiler can optimize code away if inlined, by using typing information.

2.8 Typing in Yacas

Type conversions are considered to be a 'policy' which is best implemented in the final Yacas system, through some strategy for conversion. The code that does the final symbolic manipulation can take care of converting types, as appropriate. The implementation of the simplification algorithm that defines YACAS can contain the heuristics of type conversion.

Inlining functions as described above is automatically done on macros. However, macros are slow to execute in the interpreter, so a possible feature of a compiler could be to tell it a function can be inlined. The variable scoping rules can still hold; the body should not be allowed to access variables in its calling environment.

Chapter 3

Compiling Lisp code

3.1 Introduction

This section deals with compiling Lisp code to a form that runs more efficient on a real computer.

The overheads of executing Lisp are:

1. Parsing expressions when loading in a file. This is not a big performance penalty, as it only happens at startup, and might even be slower after compilation if more needs to be loaded from disk to run a compiled version of a program.
2. When evaluating a variable, the interpreter needs to dynamically look in a list for the value of the variable. Compiled code can be written such that it can immediately access the value.
3. When evaluating a function, looking up the function to call can be expensive. The function needs to be found in some container of functions, like a hashtable, which would run in O(log n) time, worst-case (best case O(1), when only a few functions are added).
4. The overhead of building lists to be passed to a function, which then needs to decompose the arguments.
5. Needless operations, like conversions from string representation to internal number representation and back.
6. Inefficient execution of code that could be done more efficiently in native code. An `if` or `while` statement can be expected to run a lot more efficient when not interpreted.

The first concern will be to convert the Lisp code to source code for the target platform; when the Lisp interpreter is written in `Java` we want the Lisp code compiled to `Java`, etc.

The disadvantage is the compiler then needs to be around for compilation to be able to take place. Programs thus get compiled once, and then run. The compiler is not available at run time. As compiled code can interact with interpreted code, the user is still free to write his own functions, but they will have to run interpreted.

The advantage of directly compiling to source code as input for another compiler is that it is less work. However, a little compiler could be written that takes the resulting output source code and compiles it further to some other target, dynamically at run time. Theoretically, even a byte code interpreter could be written, and a compiler for it, and this could be plugged into the system.

3.2 Steps for converting to source code

When compiling to source code, the first step is parsing in the code to be compiled. The compiler will have to look very similar to an interpreter; code needs to be read in, and part of it needs to be executed at compile time (macros will be expanded during compilation), global variables are set, functions defined, and possibly other things done while loading a file. Compilation will be taken to mean creating a class that gets loaded, and a method in this class called to perform what would usually be performed when loading the file.

The fact that compilation is reminiscent of interpretation suggests that the *strategy* pattern could be used, when the compiler is written in the target environment. For this, the default evaluation routine or object needs to be replaced by another, one that compiles in stead of interpreting. Compiling would then reduce to loading a file, but using a different 'interpreter' for it. This abstraction is further warranted by the fact that a debugging environment is needed, one where a programmer can step through code.

Defining a function will reduce to defining a class (in an object-oriented system) with an `eval` or `apply` method. The method gets called with a list of arguments to perform its tasks. In addition, routines can be created that directly perform the task with the arguments given in directly in the target programming language. This is made easier with the Lisp dialect described earlier because of the constraint of a fixed number of arguments.

3.3 Parsing the expressions into an internal format

The first possible step is to skip parsing at runtime, by doing it at compile time. Even though this does not give a much greater speed at runtime, it is a requirement if the code is to be compiled.

A form `(foo a b)` could be compiled to

```
LispObject object =
   cons(recSymbol("foo"),
      cons(recSymbol("a"),
         cons(recSymbol("b"),nil)));
```

This expression could be assigned at construction of the object, and thus built only once like the interpreter would do.

Equivalently, a symbol `bar` would be defined as:

```
LispObject object = recSymbol("bar");
```

If nothing is known about the function `foo`, no further simplifications can be made, as it could be a macro or a function, and thus nothing is known about whether the arguments need to be evaluated or not.

When `foo` is a macro, it can be dealt with easily by directly expanding the code in place. When `foo` is defined as

```
In> (defmacro foo (a) a)
Out> foo
```

Evaluation of `(foo (+ 2 3))`,

```
In> (foo (+ 2 3))
Out> (+ 2 3 )
```

is the same as inlining the code with a let statement:

```
In> (let ((a (quote (+ 2 3)))) a)
Out> (+ 2 3 )
```

Macros have to be expanded before an expression is evaluated, but other than that, the compiler can read in all the code before inlining macros, to be able to inline macros that are defined later on. Alternatively, the actual evaluation could be cast in code that looks at runtime to discover whether a function call is actually a macro expansion, but this is less efficient at run time.

In the following examples, it is assumed that there are objects of type `LispObject`, which can hold a general Lisp expression, `LispNumber` which can hold a number, and functions `eval`, `cons`, and `recSymbol`. The last, `recSymbol`, takes some native argument like a string, and converts it to a `LispObject`. `nil` is taken to be an empty list.

Optimizing an expression can then continue in various steps. Suppose we take the definition of the call `(foo a b)` above, and that we know that `foo` is a function that will be compiled. Then we can rewrite

```
LispObject object =
   eval(
      cons(recSymbol("foo"),
         cons(recSymbol("a"),
            cons(recSymbol("b"),nil))));
return object;
```

as

```
LispObject object =
   Foo.eval(
         cons(eval(recSymbol("a")),
            cons(eval(recSymbol("b")),nil)));
return object;
```

Here the arguments are evaluated explicitly before being passed to `foo`, and `foo` is hard-linked, called directly with the appropriate arguments.

A next step is to optimize the `eval` inside these expressions, eg. optimizing evaluation of `a` and `b`. If `a` and `b` were local variables from within a let statement, or arguments to a function call in its environment, this could be optimized further. Each declaration of a local variable can be made a real local variable in the target source code.

```
(let ((a 2) (b 3)) (foo a b))
```

could become

```
LispObject a = recSymbol("2");
LispObject b = recSymbol("3");
LispObject object =
   Foo.eval(cons(a,cons(b,nil)));
return object;
```

This skips lookup of local variables in an environment, as the compiler will be able to directly take the appropriate value.

In the dialect described here, we only accept a fixed number of arguments, and thus we could add a method to the `Foo` object which accepts two arguments, and the code would change to:

```
LispObject a = recSymbol("2");
LispObject b = recSymbol("3");
LispObject object = Foo.eval(a,b);
return object;
```

If `foo` is guaranteed to only work with numeric values for input, we could add another method to the `foo` evaluator object `Foo`, one which takes two numeric arguments:

```
LispNumber a = 2;
LispNumber b = 3;
LispNumber number = Foo.evalTyped(a,b);
return recSymbol(number);
```

The interesting option for the last optimization is that all the intermediate values are now numbers, not lisp atoms. The advantage to this is that if an expression is more complex than the above example, the input arguments can be converted to numbers at the beginning, and then the arithmetic operations performed, and only at the end are these numbers converted back to atoms (through the `recSymbol` function explained above). So in this case,

```
(let ((a 2) (b 3)) (+ (foo a b) 5))
```

Could be converted to:

```
LispNumber a = 2;
LispNumber b = 3;
return recSymbol(Foo.evalTyped(a,b)+5);
```

A lot of conversions to and from formats can be avoided if the compiler can recognize that it is appropriate. Code of this form would not be much slower than code written natively in the target language (actually not quite true, if you have the full language at your disposal you might be able to do optimizations by hand).

If functions are defined as macros, of course the compiler can inline them and perform additional optimizations, removing the need for the overhead of a call.

A further optimization could be a conversion to native numeric types. If the compiler knows that the result of some computation is an integer, and it fits in a native integer (for instance a 32-bit integer), the above code could become:

```
return recSymbol(Foo.evalTypedNative(2,3)+5);
```

3.4 Some general rules for optimization

Optimization generally entails doing as much work doing compilation, and as little as possible during runtime. Looking up the location where a local variable is stored, looking up a function, and in a macro that gets inlined determining if an `if` statement or some clause to a `cond` statement will always return `true` are things the compiler can find out and 'evaluate' at compile time, so it doesn't have to be done at runtime.

The compiler can optimize code based on assumptions about the input. This is why compiled languages usually have typed variables. This can also be solved, for specific cases, by using macros. Suppose one wants to multiply two matrices, `(mat-mult a b)`. The elements of the matrices `a` and `b` can be anything, univariate polynomials, multivariate polynomials, analytic functions of one variable, complex, real. But when the compiler knows the elements of the matrices are real-valued, some optimizations can be made in the body of a macro. Alternatively, if a function call required for multiplying matrix elements is defined as a function, that function can be treated as a macro,

and a separate function compiled from it and called from within `mat-mult`, to take into account the fact that it is going to be passed real-valued matrix elements. Having more information about the input gives options for optimizing the code more. This allows algorithms to be stated at a high-level, with the compiler taking care of the optimizations for a specific case.

With optimization by hand, often a situation is encountered where a loop performs a task a few million times, and thus every clock cycle that can be saved in the inner loop saves a few million clock cycles. Inlining macros allows the compiler to discover that an `if` or `cond` statement always executes the same clause, and thus the `if` or `cond` statement (which has to be independent of the loop index) can be taken outside of the loop.

3.5 Dealing with function calls and macro expansions when compiling Lisp expressions to syntax from other languages

Since we are using a strongly typed Lisp dialect with no implicit type conversions, Lisp expressions can be readily converted to other syntax quite easily. Most procedural or object-oriented languages require a notation of the form `function(arg1,arg2,...)` where in Lisp that would be `(function arg1 arg2 ...)`.

One slight problem arises due to the fact that every expression in Lisp returns a result, including macro's. Consider the following piece of code:

```
(while (< i n)
  (progn
    (print i)
    (setf i (+ i 1))
) )
```

This is easy to convert to for instance c++ syntax as:

```
while (i<n)
{
  printf("%d\n",i);
  i=i+1;
}
```

Now a slightly more advanced example shows the problem:

```
(while (progn (print i) (< i n))
  (progn
    (setf i (+ i 1))
) )
```

This is valid Lisp code. `progn` returns the result of evaluating the last expression. A naive translation to c++ would yield:

```
while ({printf("%d\n",i); i<n;})
{
  i=i+1;
}
```

which is obviously invalid c++ code. One way around this in this case could be:

```
{
  printf("%d\n",i);
  pred = i<n;
}
while (pred)
{
  i=i+1;
  {
    printf("%d\n",i);
    pred = i<n;
  }
}
```

However, in this case, if the `progn` statement is large, it could result in a doubling of a large bit of code. It would however be efficient.

The other alternative would be to encapsulate the `progn` in a separate function that can be inlined by the compiler, as:

```
inline int pred(int i, int n)
{
  printf("%d\n",i);
  return (i<n);
}

...

while (pred(i,n))
{
  i=i+1;
}
```

One more sophisticated example:

```
(f a (progn b c))
```

would at first erroneously yield:

```
f(a,{b; c;});
```

Also here the choice can be to either call the `progn` statement beforehand:

```
  aarg=a;
  {
    b;
    barg = b;
  }
  f(aarg,barg);
```

This can become expansive when there are nested macro-like calls. Or, encapsulation can be performed, leading (again) to a slightly less efficient version due to function call overhead:

```
inline int ftemp()
{
  b;
  return c;
}
f(a,ftemp());
```

In the compiler, the choice has to be made to either inline the code directly, or to allow the c++/Java compiler to inline the code. The above suggests that when generating code for a function body, the algorithm should first try to expand macro bodies, until it encounters a function call, and then expand the function calls as arguments to function calls. When it cannot expand a function argument because it is actually a macro expansion, it can either inline it itself, or generate an inlined method for it to allow the c++/Java compiler to inline the code at hand.

3.6 The concept of bootstrapping

To support a computer algebra system like Yacas, it suffices to implement the small interpreter discussed in the beginning of this section. The interpreter will however be slow, as a lot of the functions required will have to e written in Lisp code and interpreted.

Bootstrapping can solve this problem. One starts with a small interpreter which can run the code, albeit slowly, and a (possibly poor) compiler is written and used to compile some base components of the system. At the second iteration, in stead of loading the files that were loaded the first time, the compiled versions can be loaded, which run a lot faster. This can be repeated many times, compiling a small sub-system, and loading the compiled version the next time the interpeter starts up, thus making the system faster at each iteration.

All that is then needed for a YACAS system is to have a little Lisp interpreter in place, plus a compiler to compile to the native platform, and 'bootstrap' the rest of the code into the system using the compiler. The system effectively 'ports' itself to the new platform, by generating efficient code for that platform.

For example, one could first start with compiling some utility functions, then compile support for parsing YACAS expressions and pattern matching, and then compile more powerful script code.

As a first step, a simple but fast compiler can be invoked, invoking more powerful compilers each time the system has faster components, resulting in faster and faster code.

3.7 Strong typing as an option for generating optimized code

Until now we have only discussed an untyped Lisp system, one where the system doesn't know the types of arguments passed or variables. An untyped Lisp is in principle enough for implementing a CAS. However, a small addition, declaration of types for parameters and variables, has a lot of advantages.

Take a function `fact`

```
(defun fact (n)
   (if (equal n 0)
      1
      (* n (fact (+ n -1)))
)  )
```

Now, if we write this as

```
(defun (fact INTEGER) ( (n INTEGER) )
   (if (equal n 0)
      1
      (* n (fact (+ n -1)))
)  )
```

Note that this just adds extra information, about types that should be passed in or returned. But the addition of type information greatly aids in translating (compiling) to other targets. For example, in c++ this bit of code could look like

```
BigInteger fact(BigInteger n)
{
   if (n==0)
      return 1;
   else
      return n*fact(n-1);
}
```

Strong typing has the advantages of adding extra information the compiler can use to optimize code further, and strong typing also allows the compiler to give appropriate error messages when a function is called with incompatible arguments.

The step back to untyped Lisp can of course always be made, with a simple translation step removing typing information from expressions. We can keep a simple low-level Lisp untyped interpreter, write code in a typed version of the Lisp dialect, and in a translation phase remove the typing information so the untyped interpreter can understand it.

For the typed Lisp, the following functions are changed:

1. `defun` and `defmacro`
2. `let`

In addition, global variables will have to be declared explicitly, with `global`:

1. `(global <var> <type>)` - declares a global variable.

`declare` is not needed for the untyped interpreter, just for the typed Lisp code that will be used for conversion to c++/Java or other targets.

The typing system is very simple; no automatic type conversion is performed. In stead, the user is expected to call conversion routines. For instance, when a function `sin` expects a float, but an integer `n` is passed in, the calling sequence should ne something like `(sin (int-to-float n))`, and not `(sin n)`. Some functions, like `setf`, will accept different types of arguments (the second argument to `setf` should be of the type of the variable the value is assigned to).

This is a very simple typing system, but this Lisp dialect is not meant to be really used for programming, just a basic minimum set. Overloading of functions and variable number of arguments are also not supported.

The following native types are supported in the basic Lisp dialect:

1. `OBJECT` - any Lisp expression
2. `INTEGER` - Big (arbitrary precision) integer
3. `FLOAT` - Big (arbitrary precision) float
4. `STRING` - text string (should be encapsulated by double quotes when entered by the user)
5. `STRINGBUFFER` - String, but not stored in any caches or hash tables. Stringbuffer is meant to be used to build up strings.
6. `CHAR` - Array element of a string or string buffer. This is made a base type so that parsers and tokenizers can be written in the Lisp dialect itself, and efficiently compiled to native code. Char is represented as a string with one character.
7. `HASHTABLE` - fast association of one object with another.

This list will be extended.

The compiler can also be made to accept non-typed variables, arguments and functions. The type of these objects should then default to `OBJECT`. However, this does not save the user from converting to appropriate type. For instance, if the type of some expression is not explicitly set to `INTEGER`, and it is used in addition, then a call to `object-to-int` is required every time the object is treated as an integer. Next to it being a nuisance for the programmer, it also makes the code less efficient.

The compiler is of course free to ignore types when it comes to translation. All objects should be able to pass as type `OBJECT`. The typing system is there to ensure efficiency of the generated code. The untyped interpreter described in the beginning of

this book disregards all typing information whatsoever. The compiler does however have to check that arguments passed in have a valid type.

By the same token, the interpreter could be modified to handle the typed expressions, and just ignore the typing information. The interpreter could easily be extended to ignore the typing information given:

```
In> (defun foo-untyped (n) (+ n 2))
Out> foo-untyped
In> (defun (foo-typed INTEGER) ( (n INTEGER) ) (+ n 2) )
Out> (foo-typed INTEGER )
In> (foo-untyped 3)
Out> 5
In> (foo-typed 3)
Out> 5
```

The change from `<name>` to `(<name> <type>)` is a trivial one. But this extra information can easily be mapped to strong typed compiled languages, which is why it is a useful step.

Indeed, 100 lines of Yacas (Yacas already had an interpreter at the time of writing of this document, version 1.0.53 had an interpreter written in c++) code can render the following:

```
(defun foo
    (n )
    (IntegerToObject
      (+
        (ObjectToInteger n )2 )))
```

to

```
LispObject foo(LispObject n)
{
   LispObject result ;
   {
      result  = IntegerToObject(ObjectToInteger(n)+2);
   }
   return result;
}
```

and

```
(defun
    (foo INTEGER )
    ((n INTEGER ))
    (+ n 2 ))
```

to

```
BigInteger foo(BigInteger n)
{
   BigInteger result ;
   {
      result  = n+2;
   }
   return result;
}
```

The typing information in effect gives the compiler enough information to directly convert expressions to another syntax. The compiler thus just becomes a simple expression transformer.

Chapter 4

Converting Yacas script code to other languages

4.1 Introduction

This section describes an implementation of a minimal environment in Common Lisp that allows execution of code originally written in YACAS script code.

Small Lisp interpreters can easily be written in any language. Being able to directly compile the interpreter into a program from for instance Common Lisp or Java allows the freedom to first print the code for the reference Yacas interpreter implementation, and then use it in other projects later on.

The main aim for Yacas is to allow for mathematically-oriented algorithms to be written easily. Being able to compile that code to code that can be compiled into other syntax gives flexibility. This effort thus also defines tha absolute bare minimum required for such a system.

Common Lisp was chosen because it is sufficiently close to YACAS, YACAS resembling a Lisp interpreter internally. YACAS already comes with a parser for YACAS script code, and can easily manipulate expressions to bring them into a form suitable for execution in another environment.

Not all of YACAS will be made available, just the functional subset needed for doing CAS. Other systems have their own functionality for interacting with the system (file input/output and such).

4.2 Relevant parts of the system

The minimum set up consists of:

1. A parser. Although not highly necessary, the author feels the Yacas syntax is a comfortable one. One reference implementation can be written in the mini sub-set of Lisp that will be used.
2. A compiler for converting from Yacas script to Common Lisp evaluable expressions and functions.
3. Various small utility functions written in a small sub-set of Lisp, to offer functionality required for executing YACAS script at run time. Pattern matching functionality for instance can be implemented easily in Lisp code itself.
4. if necessary, a custom eval operation to alter the default evaluation behavior. For instance, in Common Lisp, when a function or variable is not bound, the system raises an error, whereas YACAS returns the expression unevaluated.

Thus this project takes a minimum subset of Common Lisp to implement a minimum environment for running code originally written in YACAS script. In practice this system could then also be used in other Lisp dialects like Scheme or elisp, or a minimum Lisp interpreter implementation can be written for other environments.

The following sections describe the implementation of the parts described above.

4.3 Naming conventions

Along with the code that is converted to Lisp code, and the code to do this conversion (written in Yacas code), comes a small body of utility functions written in Lisp, which are required for simulating YACAS in a Lisp environment.

The functions defined that offer specific functionality for YACAS interpretation are prepended with a `yacas-`, in order to be able to easily recognize their origin.

Functions compiled from YACAS script have `yacas-script-` prepended, and `-arity-n` appended (where n is the arity of the function defined).

The lisp code is divided into the following modules:

1. `yacas.lisp` - the main top-level entry code. This module loads all other modules, and implements the top-level read-eval-print loop.
2. `match.lisp` - implementation of dynamic pattern matching.
3. `yacasread.lisp` - implementation of the parser.
4. `yacaseval.lisp` - implementation of the YACAS evaluation scheme.
5. `yacasprint.lisp` - implementation of the infix pretty printer.
6. `yacassupport.lisp` - various other utility routines, defining a small API that can be called from Yacas code or the command line.

4.4 Functions used from the Common Lisp environment

The conversion to Common Lisp uses a small sub-set of what Common Lisp offers, or worded differently, it only needs a small subset in order to be implemented. Better optimized versions can be made specifically for Common Lisp, but the exercise is to define a minimum required set of features.

The following is used from Common Lisp:

1. the T and NIL atoms, where NIL is the empty list but also designates 'false' when used in a predicate. Anything other than NIL is considered to represent 'true'.
2. defun, cond, equal, atom, eval, and, bound, listp, first, rest

4.5 The read-eval-print loop

Yacas can be invoked from within the Common Lisp shell by typing `(yacas)`. This starts the normal read-eval-print loop, parsing YACAS syntax, converting it to internal format, evaluating it, and lastly printing the result on screen in the same infix syntax used for input.

Note that in theory both read and print can be replaced by something else, or not implemented at all (then the standard Lisp `print` and `read` can be used.

The functions to perform these tasks are `yacas-read`, `yacas-eval` and `yacas-print`.

4.6 Making the Yacas parser available

The parser acts as a front-end to YACAS functionality, converting infix notation more commonly used by humans for writing down mathematical expressions into an internal representation. For Lisp the representation is fixed: linked lists of atoms.

4.7 Compiling functions to native Lisp syntax

The pattern matcher

4.8 The custom evaluator

Part of the language lies in how expressions get evaluated, after they have been converted to internal format, taking an input expression and returning some result after performing an operation on the expression commonly referred to as evaluation.

There are slight differences between the way Common Lisp evaluates and the way YACAS expressions should be evaluated, necessitating a bit of code between the two.

In YACAS, when a function is not defined or a variable not bound to a value, the expression in question is returned unevaluated (actually, in the case of a function call, its arguments are evaluated).

Since interpreting a Lisp program is just an algorithm, and any algorithm can be implemented in Lisp, one could write a Lisp interpreter in Lisp itself. In fact, it is very simple to do so. This section describes an example Lisp interpreter written in Lisp itself. It will be used in the end to implement an interpreter for YACAS code.

The advantage of directly compiling to native code (c++/Java) is clear; one stands a chance of writing a Lisp interpreter in Lisp itself, and have it be as efficient as the internal interpreter itself (if the compiler is good enough). The interpreting code just needs to be compiled by a good compiler.

4.9 The system in action

Example output code generated by the compiler

Chapter 5

GNU Free Documentation License

Version 1.1, March 2000

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

Preamble

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