Tcl Programming

en.wikibooks.org

December 29, 2013

Contents

0.1 Tcl: the Tool Command language

0.1.1 Introduction

So what is Tcl?

The name Tcl is derived from "Tool Command Language" and is pronounced "tickle". Tcl is a radically simple open-source interpreted programming language that provides common facilities such as variables, procedures, and control structures as well as many useful features that are not found in any other major language. Tcl runs on almost all modern operating systems such as Unix, Macintosh, and Windows (including Windows Mobile).

While Tcl is flexible enough to be used in almost any application imaginable, it does excel in a few key areas, including: automated interaction with external programs, embedding as a library into application programs, language design, and general scripting.

Tcl was created in 1988 by John Ousterhout and is distributed under a BSD style license[1] (which allows you everything GPL does, plus closing your source code). The current stable version, in February 2008, is 8.5.1 (8.4.18 in the older 8.4 branch).

The first major GUI extension that works with Tcl is Tk, a toolkit that aims to rapid GUI development. That is why Tcl is now more commonly called Tcl/Tk.

The language features far-reaching introspection, and the syntax, while simple[2], is very different from the Fortran/Algol/C++/Java world. Although Tcl is a string based language

1 `http://www.tcl.tk/software/tcltk/license_terms.html`
2 `http://wiki.tcl.tk/10259`

there are quite a few object-oriented extensions for it like Snit[3], incr Tcl[4], and XOTcl[5] to name a few.

Tcl was originally developed as a reusable command language for experimental computer aided design (CAD) tools. The interpreter is implemented as a C library that could be linked into any application. It is very easy to add new functions to the Tcl interpreter, so it is an ideal reusable "macro language" that can be integrated into many applications.

However, Tcl is a programming language in its own right, which can be roughly described as a cross-breed between

- LISP/Scheme (mainly for it's tail-recursion capabilities),
- C (control structure keywords, expr syntax) and
- Unix shells (but with more powerful structuring).

One language, many styles

Although a language where "everything is a command" appears like it must be "imperative" and "procedural", the flexibility of Tcl allows one to use functional or object-oriented styles of programming very easily. See "Tcl examples" below for ideas what one can do.

Also, it is very easy to implement other programming languages (be they (reverse) polish notation, or whatever) in Tcl for experimenting. One might call Tcl a "CS Lab". For instance, here's how to compute the average of a list of numbers in Tcl (implementing a J-like functional language - see *Tacit programming* below):

```
Def mean = fork /. sum llength
```

or, in a RPN language similar to FORTH or Postscript,

```
mean  dup sum swap size double / ;
```

while a more traditional, "procedural" approach would be

```
proc mean list {
   set sum 0.
   foreach element $list {set sum [expr {$sum + $element}]}
   return [expr {$sum / [llength $list]}]
}
```

Here is yet another style (not very fast on long lists, but depends on nothing but Tcl). It works by building up an expression, where the elements of the lists are joined with a plus sign, and then evaluating that:

```
proc mean list {expr double([join $list +])/[llength $list]}
```

3 http://tmml.sourceforge.net/doc/tcllib/snitfaq.html
4 http://incrtcl.sourceforge.net/itcl/
5 http://media.wu-wien.ac.at/

From Tcl 8.5, with math operators exposed as commands, and the expand operator, this style is better:

```
proc mean list {expr {[tcl::mathop::+ {*}$list]/double([llength $list])}}
```

or, if you have imported the tcl::mathop operators, just

```
proc mean list {expr {[+ {*}$list]/double([llength $list])}}
```

Note that all of these are in Tcl, just that the first two require some additional code to implement *Def* resp. ':'

A more practical aspect is that Tcl is very open for "language-oriented programming" - when solving a problem, specify a (little) language which most simply describes and solves that problem - then go implement that language...

Why should I use Tcl?

Good question. The general recommendation is: "Use the best tool for the job". A good craftsman has a good set of tools, and knows how to use them best.

Tcl is a competitor to other scripting languages like awk, Perl, Python, PHP, Visual Basic, Lua, Ruby, and whatever else will come along. Each of these has strengths and weaknesses, and when some are similar in suitability, it finally becomes a matter of taste.

Points in favour of Tcl are:

- simplest syntax (which can be easily extended)
- cross-platform availability: Mac, Unix, Windows, ...
- strong internationalization support: everything is a Unicode string
- robust, well-tested code base
- the Tk GUI toolkit speaks Tcl natively
- BSD license, which allows open-source use like GPL, as well as closed-source
- a very helpful community, reachable via newsgroup, Wiki, or chat :)

Tcl is not the best solution for every problem. It is however a valuable experience to find out what is possible with Tcl.

Example: a tiny web server

Before spoon-feeding the bits and pieces of Tcl, a slightly longer example might be appropriate, just so you get the feeling how it looks. The following is, in 41 lines of code, a complete little web server that serves static content (HTML pages, images), but also provides a subset of CGI functionality: if an URL ends with *.tcl*, a Tcl interpreter is called with it, and the results (a dynamically generated HTML page) served.

Note that no extension package was needed - Tcl can, with the **socket** command, do such tasks already pretty nicely. A socket is a channel that can be written to with **puts**. The **fcopy** copies asynchronously (in the background) from one channel to another, where the source is either a process pipe (the "exec tclsh" part) or an open file.

This server was tested to work pretty well even on 200MHz Windows 95 over a 56k modem, and serving several clients concurrently. Also, because of the brevity of the code, this is an educational example for how (part of) HTTP works.

```
# DustMotePlus - with a subset of CGI support
set root      c:/html
set default   index.htm
set port      80
set encoding  iso8859-1
proc bgerror msg {puts stdout "bgerror: $msg\n$::errorInfo"}
proc answer {sock host2 port2} {
    fileevent $sock readable [list serve $sock]
}
proc serve sock {
    fconfigure $sock -blocking 0
    gets $sock line
    if {[fblocked $sock]} {
        return
    }
    fileevent $sock readable ""
    set tail /
    regexp {(/[^ ?]*)(\?[^ ]*)?} $line -> tail args
    if {[string match */ $tail]} {
        append tail $::default
    }
    set name [string map {%20 " " .. NOTALLOWED} $::root$tail]
    if {[file readable $name]} {
        puts $sock "HTTP/1.0 200 OK"
        if {[file extension $name] eq ".tcl"} {
            set ::env(QUERY_STRING) [string range $args 1 end]
            set name [list |tclsh $name]
        } else {
            puts $sock "Content-Type: text/html;charset=$::encoding\n"
        }
        set inchan [open $name]
        fconfigure $inchan -translation binary
        fconfigure $sock   -translation binary
        fcopy $inchan $sock -command [list done $inchan $sock]
    } else {
        puts $sock "HTTP/1.0 404 Not found\n"
        close $sock
    }
}
proc done {file sock bytes {msg {}}} {
    close $file
    close $sock
}
socket -server answer $port
puts "Server ready..."
vwait forever
```

And here's a little "CGI" script I tested it with (save as time.tcl):

```
# time.tcl - tiny CGI script.
if {![info exists env(QUERY_STRING)]} {
    set env(QUERY_STRING) ""
}
puts "Content-type: text/html\n"
puts "<html><head><title>Tiny CGI time server</title></head>
<body><h1>Time server</h1>
Time now is: [clock format [clock seconds]]
<br>
Query was: $env(QUERY_STRING)
<hr>
```

```
<a href=index.htm>Index</a>
</body></html>"
```

Where to get Tcl/Tk

On most Linux systems, Tcl/Tk is already installed. You can find out by typing *tclsh* at a console prompt (xterm or such). If a "%" prompt appears, you're already set. Just to make sure, type *info pa* at the % prompt to see the patchlevel (e.g. 8.4.9) and *info na* to see where the executable is located in the file system.

Tcl is an open source project. The sources are available from `http://tcl.sourceforge.net/` if you want to build it yourself.

For all major platforms, you can download a binary **ActiveTcl** distribution from ActiveState[6]. Besides Tcl and Tk, this also contains many popular extensions - it's called the canonical "Batteries Included" distribution.

Alternatively, you can get Tclkit[7]: a Tcl/Tk installation wrapped in a single file, which you don't need to unwrap. When run, the file mounts itself as a virtual file system, allowing access to all its parts.

January 2006, saw the release of a new and promising one-file vfs distribution of Tcl; **eTcl**. Free binaries for Linux, Windows, and Windows Mobile 2003 can be downloaded from `http://www.evolane.com/software/etcl/index.html` . Especially on PocketPCs, this provides several features that have so far been missing from other ports: sockets, window "retreat", and can be extended by providing a startup script, and by installing pure-Tcl libraries.

First steps

To see whether your installation works, you might save the following text to a file *hello.tcl* and run it (type *tclsh hello.tcl* at a console on Linux, double-click on Windows):

```
package require Tk
pack [label .l -text "Hello world!"]
```

It should bring up a little grey window with the greeting.

To make a script directly executable (on Unix/Linux, and Cygwin on Windows), use this first line (the # being flush left):

```
#!/usr/bin/env tclsh
```

or (in an older, deprecated tricky style):

```
#! /bin/sh
# the next line restarts using tclsh \
exec tclsh "$0" ${1+"$@"}
```

6 http://www.activestate.com/Products/Download/Download.plex?id=ActiveTcl
7 http://www.equi4.com/tclkit.html

This way, the shell can determine which executable to run the script with.

An even simpler way, and highly recommended for beginners as well as experienced users, is to start up tclsh or wish interactively. You will see a % prompt in the console, can type commands to it, and watch its responses. Even error messages are very helpful here, and don't cause a program abort - don't be afraid to try whatever you like! Example:

```
$ tclsh
```

```
info patchlevel
```

```
8.4.12
```

```
expr 6*7
```

```
42
```

```
expr 42/0
```

```
divide by zero
```

You can even write programs interactively, best as one-liners:

```
proc ! x {expr {$x<=2? $x: $x*[! [incr x -1]]}}
! 5
```

```
120
```

For more examples, see the chapter "A quick tour".

0.1.2 Syntax

Syntax is just the rules how a language is structured. A simple syntax of English could say (ignoring punctuation for the moment):

- A text consists of one or more sentences
- A sentence consists of one or more words

Simple as this is, it also describes Tcl's syntax very well - if you say "script" for "text", and "command" for "sentence". There's also the difference that a Tcl word can again contain a script or a command. So

```
if {$x < 0} {set x 0}
```

is a command consisting of three words: *if*, a condition in braces, a command (also consisting of three words) in braces.

```
Take this for example
```

is a well-formed Tcl command: it calls *Take* (which must have been defined before) with the three arguments "this", "for", and "example". It is up to the command how it interprets its arguments, e.g.

```
puts acos(-1)
```

will write the string "acos(-1)" to the stdout channel, and return the empty string "", while

```
expr acos(-1)
```

will compute the arc cosine of -1 and return 3.14159265359 (an approximation of *Pi*), or

```
string length acos(-1)
```

will invoke the *string* command, which again dispatches to its *length* sub-command, which determines the length of the second argument and returns 8.

Quick summary

A Tcl **script** is a string that is a sequence of commands, separated by newlines or semicolons.

A **command** is a string that is a list of words, separated by blanks. The first word is the name of the command, the other words are passed to it as its arguments. In Tcl, *"everything is a command"* - even what in other languages would be called declaration, definition, or control structure. A command can interpret its arguments in any way it wants - in particular, it can implement a different language, like *expr*.

A **word** is a string that is a simple word, or one that begins with { and ends with the matching } (braces), or one that begins with " and ends with the matching ". Braced words are not evaluated by the parser. In quoted words, substitutions can occur before the command is called:

- $[A-Za-z0-9_]+ substitutes the value of the given variable. Or, if the variable name contains characters outside that regular expression, another layer of bracing helps the parser to get it right:

```
puts "Guten Morgen, ${Schüler}!"
```

If the code would say *$Schüler*, this would be parsed as the value of variable *$Sch*, immediately followed by the constant string *üler*.

- (Part of) a word can be an embedded script: a string in [] brackets whose contents are evaluated as a script (see above) before the current command is called.

In short: Scripts and commands contain words. Words can again contain scripts and commands. (This can lead to words more than a page long...)

Arithmetic and logic expressions are not part of the Tcl language itself, but the language of the **expr** command (also used in some arguments of the **if**, **for**, **while** commands) is basically equivalent to C's expressions, with infix operators and functions. See separate chapter on *expr* below.

The man page: 11 rules

Here is the complete manpage for Tcl (8.4) with the "endekalogue", the 11 rules. (From 8.5 onward there is a twelfth rule regarding the {*} feature).

The following rules define the syntax and semantics of the Tcl language:

(1) Commands A Tcl script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets are command terminators during command substitution (see below) unless quoted.

(2) Evaluation A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.

(3) Words Words of a command are separated by white space (except for newlines, which are command separators).

(4) Double quotes If the first character of a word is double-quote (") then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.

(5) Braces If the first character of a word is an open brace ({) then the word is terminated by the matching close brace (}). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.

(6) Command substitution If a word contains an open bracket ([) then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket (``]*). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.*

(7) Variable substitution If a word contains a dollar-sign ($) then Tcl performs variable substitution: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms:

$name

Name is the name of a scalar variable; the name is a sequence of one or more characters that are a letter, digit, underscore, or namespace separators (two or more colons).

$name(index)

Name gives the name of an array variable and index gives the name of an element within that array. Name must contain only letters, digits, underscores, and namespace separators, and may be an empty string. Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of index.

${name}

Name is the name of a scalar variable. It may contain any characters whatsoever except for close braces. There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces.

(8) Backslash substitution If a backslash (\) appears within a word then backslash substitution occurs. In all cases but those described below the backslash is dropped and the following character is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without triggering special processing. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

a

Audible alert (bell) (0x7).

b

Backspace (0x8).

f

Form feed (0xc).

n

Newline (0xa).

r

Carriage-return (0xd).

t

Tab (0x9).

v

Vertical tab (0xb).

<newline>whiteSpace

A single space character replaces the backslash, newline, and all spaces and tabs after the newline. This backslash sequence is unique in that it is replaced in a separate pre-pass before the command is actually parsed. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.

\\

Literal backslash (\), no special effect.

\ooo

The digits ooo (one, two, or three of them) give an eight-bit octal value for the Unicode character that will be inserted. The upper bits of the Unicode character will be 0.

\xhh

The hexadecimal digits hh give an eight-bit hexadecimal value for the Unicode character that will be inserted. Any number of hexadecimal digits may be present; however, all but the last two are ignored (the result is always a one-byte quantity). The upper bits of the Unicode character will be 0.

\uhhhh

The hexadecimal digits hhhh (one, two, three, or four of them) give a sixteen-bit hexadecimal value for the Unicode character that will be inserted.

Backslash substitution is not performed on words enclosed in braces, except for backslash-newline as described above.

(9) Comments If a hash character (#) appears at a point where Tcl is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears at the beginning of a command.

(10) Order of substitution Each character is processed exactly once by the Tcl interpreter as part of creating the words of a command. For example, if variable substitution occurs then no further substitutions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the Tcl interpreter; no substitutions are performed before making the recursive call and no additional substitutions are performed on the result of the nested script. Substitutions take place from left to right, and each substitution is evaluated completely before attempting to evaluate the next. Thus, a sequence like

```
set y [set x 0][incr x][incr x]
```

will always set the variable y to the value, 012.

(11) Substitution and word boundaries Substitutions do not affect the word boundaries of a command. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

Comments

The first rule for comments is simple: comments start with # where the first word of a command is expected, and continue to the end of line (which can be extended, by a trailing backslash, to the following line):

```
# This is a comment \
```

```
going over three lines \
with backslash continuation
```

One of the problems new users of Tcl meet sooner or later is that comments behave in an unexpected way. For example, if you comment out part of code like this:

```
# if {$condition} {
    puts "condition met!"
# }
```

This happens to work, but any unbalanced braces in comments may lead to unexpected syntax errors. The reason is that Tcl's grouping (determining word boundaries) happens before the # characters are considered.

To add a comment behind a command on the same line, just add a semicolon:

```
puts "this is the command" ;# that is the comment
```

Comments are only taken as such where a command is expected. In data (like the comparison values in **switch**), a # is just a literal character:

```
if $condition {# good place
   switch -- $x {
       #bad_place {because switch tests against it}
       some_value {do something; # good place again}
   }
}
```

To comment out multiple lines of code, it is easiest to use "if 0":

```
if 0 {
    puts "This code will not be executed"
    This block is never parsed, so can contain almost any code
    - except unbalanced braces :)
}
```

0.1.3 Data types

In Tcl, all values are strings, and the phrase "**Everything is a string**" is often used to illustrate this fact. But just as 2 can be used in can be interpreted in English as "the number 2" or "the character representing the number 2", two different functions in Tcl can interpret the same value in two different ways. The command `expr`, for example, interprets "2" as a number, but the command `string length` interprets "2" as a single character. All values in Tcl can be interpreted either as characters or something else that the characters represent. The important thing to remember is that every value in Tcl is a string of characters, and each string of characters might be interpreted as something else, depending on the context. This will become more clear in the examples below. For performance reasons, versions of Tcl since 8.0 keep track of both the string value and how that string value was last interpreted. This section covers the various "types" of things that Tcl values (strings) get interpreted as.

Strings

A string is a sequence of zero or more characters (where all 16-bit Unicodes are accepted in almost all situations, see in more detail below). The size of strings is automatically administered, so you only have to worry about that if the string length exceeds the virtual memory size.

In contrast to many other languages, strings in Tcl don't need quotes for markup. The following is perfectly valid:

```
set greeting Hello!
```

Quotes (or braces) are rather used for grouping:

```
set example "this is one word"
set another {this is another}
```

The difference is that inside quotes, substitutions (like of variables, embedded commands, or backslashes) are performed, while in braces, they are not (similar to single quotes in shells, but nestable):

```
set amount 42
puts "You owe me $amount" ;#--> You owe me 42
puts {You owe me $amount} ;#--> You owe me $amount
```

In source code, quoted or braced strings can span multiple lines, and the physical newlines are part of the string too:

```
set test "hello
world
in three lines"
```

To **reverse a string**, we let an index *i* first point at its end, and decrementing *i* until it's zero, append the indexed character to the end of the result *res*:

```
proc sreverse str {
set res ""
for {set i [string length $str]} {$i > 0} {} {
    append res [string index $str [incr i -1]]
}
set res
}

sreverse "A man, a plan, a canal - Panama"
```

```
 amanaP - lanac a ,nalp a ,nam A
```

Hex-dumping a string:

```
proc hexdump string {
    binary scan $string H* hex
    regexp -all -inline .. $hex
}

hexdump hello
```

```
68 65 6c 6c 6f
```

Finding a substring in a string can be done in various ways:

```
string first  $substr  $str ;# returns the position from 0, or -1 if not found

string match *$substr* $str ;# returns 1 if found, 0 if not

regexp $substr  $str ;# the same
```

The matching is done with exact match in *string first*, with glob-style match in *string match*, and as a regular expression in *regexp*. If there are characters in *substr* that are special to *glob* or regular expressions, using *string first* is recommended.

Lists

Many strings are also well-formed **lists**. Every simple word is a list of length one, and elements of longer lists are separated by whitespace. For instance, a string that corresponds to a list of three elements:

```
set example {foo bar grill}
```

Strings with unbalanced quotes or braces, or non-space characters directly following closing braces, cannot be parsed as lists directly. You can explicitly **split** them to make a list.

The "constructor" for lists is of course called *list*. It's recommended to use when elements come from variable or command substitution (braces won't do that). As Tcl commands are lists anyway, the following is a full substitute for the *list* command:

```
proc list args {set args}
```

Lists can contain lists again, to any depth, which makes modelling of matrixes and trees easy. Here's a string that represents a 4 x 4 unit matrix as a list of lists. The outer braces group the entire thing into one string, which includes the literal inner braces and whitespace, including the literal newlines. The list parser then interprets the inner braces as delimiting nested lists.

```
{{1 0 0 0}
 {0 1 0 0}
 {0 0 1 0}
 {0 0 0 1}}
```

The newlines are valid list element separators, too.

Tcl's list operations are demonstrated in some examples:

```
set      x {foo bar}
llength  $x        ;#--> 2
lappend  x  grill  ;#--> foo bar grill
lindex   $x 1      ;#--> bar (indexing starts at 0)
lsearch  $x grill  ;#--> 2 (the position, counting from 0)
lsort    $x        ;#--> bar foo grill
```

```
linsert  $x 2 and  ;#--> foo bar and grill
lreplace $x 1 1 bar, ;#--> foo bar, and grill
```

To change an element of a list (of a list...) in place, the **lset** command is useful - just give as many indexes as needed:

```
set test {{a b} {c d}}

 {a b} {c d}



lset test 1 1 x

 {a b} {c x}
```

The **lindex** command also takes multiple indexes:

```
lindex $test 1 1

 x
```

Example: To find out whether an element is contained in a list (from Tcl 8.5, there's the **in** operator for that):

```
proc in {list el} {expr {[lsearch -exact $list $el] >= 0}}
in {a b c} b

 1



in {a b c} d

 #ignore this line, which is only here because there is currently a bug in
  wikibooks rendering which makes the 0 on the following line disappear when it is
  alone
 0
```

Example: remove an element from a list variable by value (converse to lappend), if present:

```
proc lremove {_list el} {
  upvar 1 $_list list
  set pos [lsearch -exact $list $el]
  set list [lreplace $list $pos $pos]
}

set t {foo bar grill}

 foo bar grill
```

```
lremove t bar
```

```
 foo grill
```

```
set t
```

```
 foo grill
```

A simpler alternative, which also removes all occurrences of *el*:

```
proc lremove {_list el} {
  upvar 1 $_list list
  set list [lsearch -all -inline -not -exact $list $el]
}
```

Example: To draw a random element from a list L, we first determine its length (using *llength*), multiply that with a random number > 0.0 and < 1.0, truncate that to integer (so it lies between 0 and length-1), and use that for indexing (*lindex*) into the list:

```
proc ldraw L {
   lindex $L [expr {int(rand()*[llength $L])}]
}
```

Example: Transposing a matrix (swapping rows and columns), using integers as generated variable names:

```
proc transpose matrix {
   foreach row $matrix {
       set i 0
       foreach el $row {lappend [incr i] $el}
   }
   set res {}
   set i 0
   foreach e [lindex $matrix 0] {lappend res [set [incr i]]}
   set res
}

transpose {{1 2} {3 4} {5 6}}
```

```
 {1 3 5} {2 4 6}
```

Example: pretty-printing a list of lists which represents a table:

```
proc fmtable table {
   set maxs {}
   foreach item [lindex $table 0] {
       lappend maxs [string length $item]
   }
   foreach row [lrange $table 1 end] {
       set i 0
       foreach item $row max $maxs {
```

```
        if {[string length $item]>$max} {
            lset maxs $i [string length $item]
        }
        incr i
    }
}
set head +
foreach max $maxs {append head -[string repeat - $max]-+}
set res $head\n
foreach row $table {
    append res |
    foreach item $row max $maxs {append res [format " %-${max}s |" $item]}
    append res \n
}
append res $head
}
```

Testing:

```
fmtable {
   {1 short "long field content"}
   {2 "another long one" short}
   {3 "" hello}
}
```

```
+---+------------------+--------------------+
| 1 | short            | long field content |
| 2 | another long one | short              |
| 3 |                  | hello              |
+---+------------------+--------------------+
```

Enumerations: Lists can also be used to implement enumerations (mappings from symbols to non-negative integers). Example for a nice wrapper around lsearch/lindex:

```
proc makeEnum {name values} {
   interp alias {} $name: {} lsearch $values
   interp alias {} $name@ {} lindex $values
}

makeEnum fruit {apple blueberry cherry date elderberry}
```

This assigns "apple" to 0, "blueberry" to 1, etc.

```
fruit: date
```

```
3
```

```
fruit@ 2
```

```
cherry
```

Numbers

Numbers are strings that can be parsed as such. Tcl supports integers (32-bit or even 64-bit wide) and "double" floating-point numbers. From Tcl 8.5 on, bignums (integers of arbitrarily large precision) are supported. Arithmetics is done with the **expr** command, which takes basically the same syntax of operators (including ternary *x?y:z*), parens, and math functions as C. See below for detailed discussion of *expr*.

Control the display format of numbers with the **format** command which does appropriate rounding:

```
expr 2/3.
```

```
0.666666666667
```

```
format %.2f [expr 2/3.]
```

```
0.67
```

Up to the 8.4 version (the present version is 8.5), Tcl honored the C convention that an integer starting with 0 is parsed as octal, so

```
0377 == 0xFF == 255
```

This changes in 8.5, though - too often people stumbled over "08" meant as hour or month, raised a syntax error, because 8 is no valid octal number. In the future you'd have to write 0o377 if you really mean octal. You can do number base conversions with the *format* command, where the format is %x for hex, %d for decimal, %o for octal, and the input number should have the C-like markup to indicate its base:

```
format %x 255
```

```
ff
```

```
format %d 0xff
```

```
255
```

```
format %o 255
```

```
377
```

```
format %d 0377
```

```
255
```

Variables with integer value can be most efficiently modified with the *incr* command:

```
incr i    ;# default increment is 1
incr j 2
incr i -1 ;# decrement with negative value
incr j $j ;# double the value
```

The maximal positive integer can be determined from the hexadecimal form, with a 7 in front, followed by several "F" characters. Tcl 8.4 can use "wide integers" of 64 bits, and the maximum integer there is

```
expr 0x7fffffffffffffff
```

```
9223372036854775807
```

Demonstration: one more, and it turns into the minimum integer:

```
expr 0x8000000000000000
```

```
-9223372036854775808
```

Bignums: from Tcl 8.5, integers can be of arbitrary size, so there is no maximum integer anymore. Say, you want a big factorial:

```
proc tcl::mathfunc::fac x {expr {$x < 2? 1: $x * fac($x-1)}}

expr fac(100)
```

```
93326215443944152681699238856266700490715968264381621468592963895217599993229
9156089414639761565182862536979208272237582511852109168640000000000000000000000
```

IEEE special floating-point values: Also from 8.5, Tcl supports a few special values for floating-point numbers, namely *Inf* (infinity) and *NaN* (Not a Number):

```
set i [expr 1/0.]
```

```
Inf
```

```
expr {$i+$i}
```

```
Inf
```

```
expr {$i+1 == $i}
```

```
 1
```

```
set j NaN ;# special because it isn't equal to itself
```

```
 NaN
```

```
expr {$j == $j}
```

```
 #ignore this line, which is only here because there is currently a bug in
  wikibooks rendering which makes the 0 on the following line disappear when it is
  alone
 0
```

Booleans

Tcl supports booleans as numbers in a similar fashion to C, with 0 being false and any other number being true. It also supports them as the strings "true", "false", "yes" and "no" and few others (see below). The canonical "true" value (as returned by Boolean expressions) is 1.

```
foreach b {0 1 2 13 true false on off no yes n y a} {puts "$b -> [expr
 {$b?1:0}]."}
```

```
 0 -> 0
 1 -> 1
 2 -> 1
 13 -> 1
 true -> 1
 false -> 0
 on -> 1
 off -> 0
 no -> 0
 yes -> 1
 n -> 0
 y -> 1
 expected boolean value but got "a"
```

Characters

Characters are abstractions of writing elements (e.g. letters, digits, punctuation characters, Chinese ideographs, ligatures...). In Tcl since 8.1, characters are internally represented with Unicode, which can be seen as unsigned integers between 0 and 65535 (recent Unicode versions have even crossed that boundary, but the Tcl implementation currently uses a maximum of 16 bits). Any Unicode U+XXXX can be specified as a character constant with an \uXXXX escape. It is recommended to only use ASCII characters (\u0000-\u007f) in Tcl scripts directly, and escape all others.

Convert between numeric Unicode and characters with

```
set char [format %c $int]
set int  [scan $char %c]
```

Watch out that int values above 65535 produce 'decreasing' characters again, while negative int even produces two bogus characters. format does not warn, so better test before calling it.

Sequences of characters are called strings (see above). There is no special data type for a single character, so a single character is just a string on length 1 (everything is a string). In UTF-8, which is used internally by Tcl, the encoding of a single character may take from one to three bytes of space. To determine the bytelength of a single character:

```
string bytelength $c ;# assuming [string length $c]==1
```

String routines can be applied to single characters too, e.g [string toupper] etc. Find out whether a character is in a given set (a character string) with

```
expr {[string first $char $set]>=0}
```

As Unicodes for characters fall in distinct ranges, checking whether a character's code lies within a range allows a more-or-less rough classification of its category:

```
proc inRange {from to char} {
    # generic range checker
    set int [scan $char %c]
    expr {$int>=$from && $int <= $to}
}
interp alias {} isGreek {}    inRange 0x0386 0x03D6
interp alias {} isCyrillic {} inRange 0x0400 0x04F9
interp alias {} isHangul {}   inRange 0xAC00 0xD7A3
```

This is a useful helper to convert all characters beyond the ASCII set to their \u.... escapes (so the resulting string is strict ASCII):

```
proc u2x s {
   set res ""
   foreach c [split $s ""] {
     scan $c %c int
     append res [expr {$int<128? $c :"\\u[format %04.4X $int]"}]
   }
   set res
}
```

Internal representation

In the main Tcl implementation, which is written in C, each value has both a string representation (UTF-8 encoded) and a structured representation. This is an implementation detail which allows for better performance, but has no semantic impact on the language.

Tcl tracks both representations, making sure that if one is changed, the other one is updated to reflect the change the next time it is used. For example, if the string representation of a value is "8", and the value was last used as a number in an [expr] command, the structured representation will be a numeric type like a signed integer or a double-precision floating point number. If the value "one two three" was last used in one of the list commands, the structured representation will be a list structure. There are various other "types" on the C side which may be used as the structured representation. As of Tcl 8.5, only the most recent structured representation of a value is stored, and it is replaced with a different representation when necessary. This "dual-porting" of values helps avoid, repeated parsing or "stringification", which otherwise would happen often because each time a value is encountered in source code, it is interpreted as a string prior to being interpreted in its current context. But to the programmer, the view that "everything is a string" is still maintained.

These values are stored in reference-counted structures termed objects (a term that has many meanings). From the perspective of all code that uses values (as opposed to code implementing a particular representation), they are immutable. In practice, this is implemented using a copy-on-write strategy.

0.1.4 Variables

Variables can be local or global, and scalar or array. Their names can be any string not containing a colon (which is reserved for use in namespace separators) but for the convenience of $-dereference one usually uses names of the pattern [A-Za-z0-9_]+, i.e. one or more letters, digits, or underscores.

Variables need not be declared beforehand. They are created when first assigned a value, if they did not exist before, and can be unset when no longer needed:

```
set foo    42     ;# creates the scalar variable foo
set bar(1) grill  ;# creates the array bar and its element 1
set baz    $foo   ;# assigns to baz the value of foo
set baz [set foo] ;# the same effect
info exists foo   ;# returns 1 if the variable foo exists, else 0
unset foo         ;# deletes the variable foo
```

Retrieving a variable's value with the $foo notation is only syntactic sugar for [set foo]. The latter is more powerful though, as it can be nested, for deeper dereferencing:

```
set foo   42
set bar   foo
set grill bar
puts [set [set [set grill]]] ;# gives 42
```

Some people might expect *$$$grill* to deliver the same result, but it doesn't, because of the Tcl parser. When it encounters the first and second $ sign, it tries to find a variable name (consisting of one or more letters, digits, or underscores) in vain, so these $ signs are left literally as they are. The third $ allows substitution of the variable *grill*, but no backtracking to the previous $'s takes place. So the evaluation result of *$$$grill* is *$$bar*. Nested [set] commands give the user more control.

Local vs. global

A local variable exists only in the procedure where it is defined, and is freed as soon as the procedure finishes. By default, all variables used in a proc are local.

Global variables exist outside of procedures, as long as they are not explicitly unset. They may be needed for long-living data, or implicit communication between different procedures, but in general it's safer and more efficient to use globals as sparingly as possible. Example of a very simple bank with only one account:

```
set balance 0 ;# this creates and initializes a global variable

proc deposit {amount} {
   global balance
   set balance [expr {$balance + $amount}]
}

proc withdraw {amount} {
   set ::balance [expr {$::balance - $amount}]
}
```

This illustrates two ways of referring to global variables - either with the **global** command, or by qualifying the variable name with the :: prefix. The variable *amount* is local in both procedures, and its value is that of the first argument to the respective procedure.

Introspection:

```
info vars ;#-- lists all visible variables
info locals
info globals
```

To make all global variables visible in a procedure (not recommended):

```
eval global [info globals]
```

Scalar vs. array

All of the value types discussed above in *Data types* can be put into a scalar variable, which is the normal kind.

Arrays are collections of variables, indexed by a key that can be any string, and in fact implemented as hash tables. What other languages call "arrays" (vectors of values indexed by an integer), would in Tcl rather be lists. Some illustrations:

```
#-- The key is specified in parens after the array name
set         capital(France) Paris

#-- The key can also be substituted from a variable:
set                 country France
puts       $capital($country)

#-- Setting several elements at once:
array set   capital         {Italy Rome  Germany Berlin}

#-- Retrieve all keys:
array names capital    ;#-- Germany Italy France -- quasi-random order
```

```
#-- Retrieve keys matching a glob pattern:
array names capital F* ;#-- France
```

A fanciful array name is "" (the empty string, therefore we might call this the "anonymous array" :) which makes nice reading:

```
set (example) 1
puts $(example)
```

Note that arrays themselves are not values. They can be passed in and out of procedures not as *$capital* (which would try to retrieve the value), but by reference. The **dict** type (available from Tcl 8.5) might be better suited for these purposes, while otherwise providing hash table functionality, too.

System variables

At startup, tclsh provides the following global variables:

argc

number of arguments on the command line

argv

list of the arguments on the command line

argv0

name of the executable or script (first word on command line)

auto_index

array with instructions from where to load further commands

auto_oldpath

(same as auto_path ?)

auto_path

list of paths to search for packages

env

array, mirrors the environment variables

errorCode

type of the last error, or {}, e.g. ARITH DIVZERO {divide by zero}

errorInfo

last error message, or {}

tcl_interactive

1 if interpreter is interactive, else 0

tcl_libPath

list of library paths

tcl_library

path of the Tcl system library directory

tcl_patchLevel

detailed version number, e.g. 8.4.11

tcl_platform

array with information on the operating system

tcl_rcFileName

name of the initial resource file

tcl_version

brief version number, e.g. 8.4

One can use temporary environment variables to control a Tcl script from the command line, at least in Unixoid systems including Cygwin. Example scriptlet:

```
set foo 42
if [info exists env(DO)] {eval $env(DO)}
puts foo=$foo
```

This script will typically report

```
foo=42
```

To remote-control it without editing, set the DO variable before the call:

```
DO='set foo 4711' tclsh myscript.tcl
```

which will evidently report

```
foo=4711
```

Dereferencing variables

A *reference* is something that refers, or points, to another something (if you pardon the scientific expression). In C, references are done with *pointers* (memory addresses); in Tcl, references are strings (everything is a string), namely names of variables, which via a hash table can be resolved (dereferenced) to the "other something" they point to:

```
puts foo       ;# just the string foo
puts $foo      ;# dereference variable with name of foo
puts [set foo] ;# the same
```

This can be done more than one time with nested set commands. Compare the following C and Tcl programs, that do the same (trivial) job, and exhibit remarkable similarity:

```
#include <stdio.h>
int main(void) {
  int    i =      42;
  int *  ip =     &i;
  int ** ipp =   &ip;
  int ***ippp = &ipp;
  printf("hello, %d\n", ***ippp);
  return 0;
}
```

...and Tcl:

```
set i    42
set ip   i
set ipp  ip
set ippp ipp
puts "hello, [set [set [set [set ippp]]]]"
```

The asterisks in C correspond to calls to `set` in Tcl dereferencing. There is no corresponding operator to the C `&` because, in Tcl, special markup is not needed in declaring references. The correspondence is not perfect; there are four `set` calls and only three asterisks. This is because mentioning a variable in C is an implicit dereference. In this case, the dereference is used to pass its value into printf. Tcl makes all four dereferences explicit (thus, if you only had 3 set calls, you'd see hello, i). A single dereference is used so frequently that it is typically abbreviated with $varname, e.g.

```
puts "hello, [set [set [set $ippp]]]"
```

has set where C uses asterisks, and $ for the last (default) dereference.

The hashtable for variable names is either global, for code evaluated in that scope, or local to a proc. You can still "import" references to variables in scopes that are "higher" in the call stack, with the upvar and global commands. (The latter being automatic in C if the names are unique. If there are identical names in C, the innermost scope wins).

Variable traces

One special feature of Tcl is that you can associate traces with variables (scalars, arrays, or array elements) that are evaluated optionally when the variable is read, written to, or unset.

Debugging is one obvious use for that. But there are more possibilities. For instance, you can introduce constants where any attempt to change their value raises an error:

```
proc const {name value} {
  uplevel 1 [list set $name $value]
  uplevel 1 [list trace var $name w {error constant ;#} ]
}

const x 11
incr x
```

```
can't set "x": constant
```

The trace callback gets three words appended: the name of the variable; the array key (if the variable is an array, else ""), and the mode:

- r - read
- w - write
- u - unset

If the trace is just a single command like above, and you don't want to handle these three, use a comment ";#" to shield them off.

Another possibility is tying local objects (or procs) with a variable - if the variable is unset, the object/proc is destroyed/renamed away.

8

0.2 Commands and Functions

0.2.1 Commands

Commands are basically divided into C-defined ones, procedures, and aliases (and, from Tcl 8.5 onwards, ensembles). You can rename any command with

```
rename oldname newname
```

To delete a command (make it no more reachable), use the empty string as new name:

```
rename oldname {}
```

Introspection: Get a list of all defined commands with

```
info commands
```

C-defined commands

These are implemented in C and registered to be available in the Tcl interpreter. They can come from the Tcl core, or a loaded shared library (DLL) - for instance Tk.

To get a list of built-in commands, subtract the result of *info procs* from that of *info commands*:

8 http://en.wikibooks.org/wiki/Category%3A

```
set builtins {}
set procs [info procs]
foreach cmd [info commands] {
   if {[lsearch -exact $procs $cmd] == -1} {lappend builtins $cmd}
}
```

The following C-defined commands are available in a fresh tclsh. For detailed documentation, see the respective man pages, e.g. at `http://www.tcl.tk/man/tcl8.5/TclCmd/` - I will characterize each only very briefly:

after

group of commands for timed events

after *msec ?script?*

waits, or executes the script after, some time

append *varName arg..*

appends the arguments to a string variable

array

group of commands for arrays

binary

group of commands for binary scanning and formatting

break

terminate current loop

case

deprecated, use **switch**

catch *script ?varName?*

catch possible error in *script*

cd *path*

change working directory

clock

group of commands dealing with date and time

close *handle*

closes a channel (file, socket, etc.)

concat *list..*

make one space-separated list of the arguments

continue

start next turn of current loop

encoding

group of commands dealing with character set encoding

eof *handle*

1 if channel is at end of file, else 0

error *message ?info? ?code?*

raise an error with the given message

eval *arg..*

evaluate the arguments as script

exec *file arg..*

execute a separate process

exit *?int?*

terminate this process, return status 0..127

expr *arg..*

arithmetic and logic engine, using C-like syntax and functions (variables referenced with $name). In addition, from Tcl 8.4 there are *eq* and *ne* operators for string equal or not; from 8.5, also *in* and *ni* operators for list inclusion or not

expr {"foo" in {foo bar grill}} == 1 The argument to expr should in most cases be {braced}. This prevents the Tcl parser from substituting variables in advance, while *expr* itself has to parse the value from the string. In a braced expression *expr* can parse variable references itself, and get their numeric value directly where possible. Much faster, usually. The only exception, where bracing should not be used, is if you want to substitute operators from variables:

```
foreach op {+ - * /} {puts [expr 1 $op 2]}
```

fblocked *handle*

returns 1 if the last input operation exhausted all available input, else 0

fconfigure *handle -option value...*

configure a channel, e.g. its encoding or line-end translation

fcopy *handle1 handle2*

copy data from handle1 to handle2

file

group of commands dealing with files

fileevent

group of commands dealing with events from channels (readable, writable) but not files

flush *handle*

make sure the channel's buffer is written out. Useful after *puts -nonewline*

for *initbody condition stepbody body*

loop, somehow similar to C's *for*

foreach *varlist list ?varlist list...? body*

loop over one or more lists, The *varlist*s can be a single or multiple varNames. Example:

```
% foreach {x y} {1 0  1 2  0 2  0 0} {puts "x:$x, y:$y"}
x:1, y:0
x:1, y:2
x:0, y:2
x:0, y:0
```

format *fstring arg..*

put the arguments %-formatted into fstring, similar to C's sprintf()

gets *handle ?varName?*

read a line from handle. If variable is given, assigns the line to it and returns the number of characters read; else returns the line. Guaranteed to be safe against buffer overflows

glob *?-options? pattern..*

list of files matching the glob pattern (which can contain * and ? wildcards)

global *varName..*

declare the given variable(s) as global

history

list the last interactive commands

if *condition ?then? body1 ?elseif condition body2...? ??else? bodyN?*

conditional

incr *varName ?amount?*

increments the integer variable by given amount (defaults to 1). Use negative amount to decrement

info

group of commands for introspection

interp

group of commands for interpreters

join *list ?separator?*

Turn a list into a string, with separator between elements (defaults to " ")

lappend *varName arg..*

appends the arguments to the list variable. Can also be used to make sure a variable exists:

```
lappend x ;# corresponds to: if {![info exists x]} {set x ""}
```

lindex *list int..*

retrieve an element from the list by integer index(es)

linsert *list int arg..*

inserts the arguments at int position into list

list *?arg..?*

creates a list form the arguments

llength *list*

length of the list

load *filename ?name?*

loads a shared library (DLL)

lrange *list from to*

returns a sublist at integer indexes from-to

lreplace *list from to arg..*

replaces the sublist in list with the arguments

lsearch *?-options? list element*

searches the list for the element, returns its integer index, or -1 if not found. Can be used to select a subset of elements from a list (using the -all option)

lset *varName int.. value*

sets an existing element in the named list variable, indexed by integer(s), to the given value

lsort *?-options? list*

sorts the list

namespace

group of commands dealing with namespaces

open *name ?mode ?permissions??*

opens a file or pipe, returns the handle

package

group of commands dealing with packages

pid *?handle?*

returns the id of the current process. Can also return the list of pids for a pipeline given the pipeline channel

proc *name arglist body*

defines a procedure

puts *?-nonewline? ?channel? string*

outputs a line to the given channel (default stdout) To prevent errors from closed pipe (like *more* or *head*), use

```
proc puts! str {if [catch {puts $str}] exit}
```

pwd

returns the current working directory

read *handle ?int?*

reads int bytes from handle (all if int not given)

regexp *?-options? re string ?varName...?*

regular expression matching of re in string, possibly assigning parenthesized submatches to the given variables

regsub *?-options? re value substring ?varName?*

substitutes occurrences of the regular expression re in value with substring. If varName is given, assigns the new value to it, and returns the number of substitutions; else returns the new value

rename *cmdName1 cmdName2*

renames a command from cmdName1 to cmdName2, or deletes cmdName1 if cmdName2 is {}

return *?value?*

exits from the current proc or sourced script

scan *string format ?varName...?*

extracts values by %-format in string to the given variables. Similar to C's sscanf()

seek *channelId offset ?origin?*

moves pointer in file to the given position

set *varName ?value?*

sets variable to value if given, and returns the variable's value

socket *?-myaddr addr? ?-myport myport? ?-async? host port*

open the client side of a TCP connection as a channel

socket *-server command ?-myaddr addr? port* open the server side of a TCP connection, register a handler callbacvk command for client requests

source *filename*

evaluate the contents of the given file

split *list ?charset?*

splits a string into a list, using any of the characters in charset string as delimiters (defaults to " ")

string

group of commands dealing with strings

subst *?-options? string*

performs command, variable, and/or backslash substitutions in string

switch *?-options? ?--? value alternatives*

performs one of the alternatives if the value matches

tell *handle*

return the byte position inside a file

time *body ?int?*

runs the body for int times (default 1), returns how many microseconds per iteration were used

trace

group of commands to tie actions to variables or commands

unset *varName..*

delete the given variable(s)

update *?idletasks?*

services events

uplevel *?level? body*

evaluates the body up in the call stack

upvar *?level? varName localVarName...*

ties the given variables up in the call stack to the given local variables. Used for calling by reference, e.g. for arrays

variable *varName ?value ?varName value...??*

declare the variables as being non-local in a namespace

vwait *varName*

suspend execution until the given variable has changed. Starts event loop, if not active yet

while *condition body*

performs body as long as condition is not 0

Procedures

Procedures in Tcl cover what other languages call procedures, subroutines, or functions. They always return a result (even if it is the empty string ""), so to call them functions might be most appropriate. But for historical reasons, the Tcl command to create a function is called **proc** and thus people most often call them procedures.

```
proc name argumentlist body
```

Examples:

```
proc sum {a b} {return [expr {$a+$b}]}
```

The *return* is redundant, as the proc returns anyway when reaching its end, and returning its last result: proc sum {a b} {expr {$a+$b}}

The following variant is more flexible, as it takes any number of arguments (the special argument name *args* collects all remaining arguments into a list, which is the value of the parameter *args*):

```
proc sum args {
    set res 0
    foreach arg $args {set res [expr {$res + $arg}]}
    return $res
}
```

An elegant but less efficient alternative builds a string by *join*ing the *args* with plus signs, and feeds that to *expr*:

```
proc sum args {expr [join $args +]}
```

If an argument in a proc definition is a list of two elements, the second is taken as default value if not given in the call ("Sir" in this example): proc greet {time {person Sir}} {return "good $time, $person"}

```
% greet morning John
good morning, John
% greet evening
good evening, Sir
```

Introspection: Get the names of all defined procedures with

```
info procs
```

There are also **info** subcommands to get the argument list, and possible default arguments, and body of a proc. The following example combines them to recreate the textual form of a proc, given its name (*corp* being *proc* in reverse):

```
proc corp name {
   set argl {}
   foreach arg [info args $name] {
      if [info default $name $arg def] {lappend arg $def}
      lappend argl $arg
   }
   list proc $name $argl [info body $name]
}
```

Using **rename**, you can overload any command, including the C-coded ones. First rename the original command to something else, then reimplement it with the same signature, where ultimately the original is called. Here is for instance an overloaded *proc* that reports if a procedure is defined more than once with same name:

```
rename proc _proc
_proc proc {name argl body} {
   if {[info procs $name] eq $name} {
       puts "proc $name redefined in [info script]"
   }
   _proc $name $argl $body
}
```

Named arguments: Arguments to commands are mostly by position. But it's very easy to add the behavior known from Python or Ada, that arguments can be named in function calls, which documents the code a bit better, and allows any order of arguments.

The idea (as found in Welch's book) is to use an array (here called "" - the "anonymous array") keyed by argument names. Initially, you can set some default values, and possibly override them with the args of the proc (which has to be paired, i.e. contain an even number of elements):

```
proc named {args defaults} {
   upvar 1 "" ""
   array set "" $defaults
   foreach {key value} $args {
     if {![info exists ($key)]} {
        set names [lsort [array names ""]]
        error "bad option '$key', should be one of: $names"
     }
     set ($key) $value
   }
}
```

Usage example:

```
proc replace {s args} {
  named $args {-from 0 -to end -with ""}
  string replace $s $(-from) $(-to) $(-with)
}
```

Testing:

```
% replace suchenwirth -from 4 -to 6 -with xx
suchxxirth
% replace suchenwirth -from 4 -to 6 -witha xx
bad option '-witha', should be one of: -from -to -with
```

Argument passing by name or value

Normally, arguments to commands are passed by value (as constants, or with $ prefixed to a variable name). This securely prevents side-effects, as the command will only get a copy of the value, and not be able to change the variable.

However, sometimes just that is wanted. Imagine you want a custom command to set a variable to zero. In this case, at call time specify the name of the variable (without $), and in the proc use **upvar** to link the name (in the scope "1 up", i.e. the caller's) to a local variable. I usually put a "_" before arguments that are variable names (e.g. _var), and *upvar* to the same name without "_" (e.g. var):

```
% proc zero _var {upvar 1 $_var var; set var 0}
```

```
% set try 42
42
% zero try
0
% set try
0
```

If you often use call by reference, you could indicate such arguments with a special pattern (e.g. &arg) and have the following code generate the necessary **upvar**s: proc use_refs { {char &}} {

```
    foreach v [uplevel 1 {info locals}] {
        if [string match $char* $v] {
            uplevel 1 "upvar 1 \${$v} [string range $v 1 end]"
        }
    }
}
```

That's all. This command is preferrably called first inside a proc, and upvars all arguments that begin with a specific character, the default being "&" - it runs code like

```
upvar 1 ${&foo} foo
```

in the caller's scope. Testing:

```
proc test_refs {a &b} {
   use_refs
   puts a=$a,b=$b
   set b new_value
}
% set bar 42
42
% test_refs foo bar
a=foo,b=42
```

So the values of a (by value) and b (by reference) are readable; and the side effect of changing b in the caller did also happen:

```
% set bar
new_value
```

Variable scope

Inside procedures, variables are by default local. They exist only in the proc, and are cleared up on *return*. However, you can tie local variables to others higher in the call stack (e.g. in the caller), up to the topmost global scope. Examples:

```
proc demo arg {
   global g
   set    g 0            ;# will effect a lasting change in g
   set local 1           ;# will disappear soon
   set ::anotherGlobal 2 ;# another way to address a global variable
   upvar 1 $arg myArg    ;# make myArg point at a variable 1-up
   set          myArg 3  ;# changes that variable in the calling scope
}
```

Aliases

One can also define a command as an alias to a sequence of one or more words, which will be substituted for it before execution. (The funny {} arguments are names of the source and target interpreter, which typically is the current one, named by the empty string {} or ""). Examples:

```
interp alias {} strlen {} string length
interp alias {} cp     {} file copy -force
```

Introspection: Get the names of all defined aliases with

```
interp aliases
```

0.2.2 Advanced concepts

Interpreters

Tcl being an interpreted (plus on-the-fly byte-compiled) language, an interpreter is of course a central kind of object. Every time Tcl is running, at least one interpreter is running, who takes scripts and evaluates them.

One can also create further "slave" interpreters to encapsulate data and processes, which again can have their "sub-slaves", etc., ultimately forming a tree hierarchy. Examples:

```
% interp create helper
helper
% helper eval {expr 7*6}
42
% interp delete helper
% helper eval {expr 1+2}
invalid command name "helper"
```

By deleting an interpreter, all its global (and namespaced) variables are freed too, so you can use this for modularisation and encapsulation, if needed.

In particular, **safe interpreters** have intentionally limited capabilities (for instance, access to the file system or the Web) so that possibly malicious code from over the Web cannot create major havoc.

Introspection: The following command lists the sub-interpreters ("slaves") of the current interpreter:

```
% interp slaves
```

Ensembles

Ensembles, (from Tcl 8.5 on), are commands that are composed out of sub-commands according to a standard pattern. Examples include Tcl's built-in **chan** and **clock** commands. Dispatching of subcommands, as well as informative error message for non-existing subcommands, is built-in. Subcommands are in a *dict* structure called "-map", with alternating name and action. Very simple example: namespace ensemble create -name foo -map \ {bar {puts Hello} grill {puts World}} creates a command *foo* that can be called like

```
% foo bar
Hello
% foo grill
```

```
World
% foo help
unknown or ambiguous subcommand "help": must be foo, or bar
```

Obviously, ensembles are also a good base for implementing object orientation, where the command is the name of the objects, and the map contains its methods.

Introspection: Serialize an ensemble's map with

```
namespace ensemble configure $name -map
```

Namespaces

Namespaces are containers for procedures, non-local variables, and other namespaces. They form a tree structure, with the root at the global namespace named "::". Their names are built with :: as separators too, so *::foo::bar* is a child of *::foo*, which is a child of *::* (similar to pathnames on Unix, where / is the separator as well as the root).

In a nutshell, a namespace is a separate area, or *scope*, where procedures and variables are visible and private to that scope.

To create a namespace, just *eval*uate some script (which may be empty) in it:

```
namespace eval ::foo {}
```

Now you can use it to define procedures or variables:

```
proc ::foo::test {} {puts Hello!}
set  ::foo::var 42
```

To get rid of a namespace (and clean up all its variables, procs, and child namespaces):

```
namespace delete ::foo
```

Introspection:

```
namespace children ::
info var namespace::*
info commands namespace::*
```

The following code gives an approximate size in bytes consumed by the variables and children of a Tcl namespace (of which ::, the global namespace, is of particular interest - all the (grand)*children are also added). If you call this proc repeatedly, you can observe whether data are piling up:

```
proc namespace'size ns {
  set sum [expr wide(0)]
  foreach var [info vars ${ns}::*] {
      if {[info exists $var]} {
          upvar #0 $var v
          if {[array exists v]} {
              incr sum [string bytelength [array get v]]
          } else {
              incr sum [string bytelength $v]
          }
      }
  }
  foreach child [namespace children $ns] {
      incr sum [namespace'size $child]
  }
  set sum
}
```

Usage example:

```
% puts [namespace'size ::]
179914
```

Threads

Tcl users have traditionally been skeptical about threads (lightweight concurrent subprocesses) - the event loop model has proved pretty powerful, and simpler to debug. Originally from Tk, the event loop has moved into Tcl and serves

- fileevents (more on channels than real files)
- timed events
- UI events (mouse or keyboard actions by the user)

However, there is a growing tendency to enable threads in Tcl builds. The underlying model is that every thread runs in an interpreter of its own, so it is mostly encapsulated from the rest of the world. Communication between threads must be done with explicit methods.

0.2.3 Packages and extensions

Packages are Tcl's recommended way to modularize software, especially supportive libraries. The user most often uses the command

```
package require name ?version?
```

One can write packages in pure Tcl, or as wrappers for extensions which come with one or more compiled shared libraries (plus optionally one or more Tcl scripts). Popular extensions are:

- BWidget (adds useful widgets to Tk - more below)
- Expect (supports remote execution over networks)

- Img (adds support of additional image file formats to Tk)
- snack (sound input/output)
- Snit (OO extension, with support for "megawidgets" in Tk)
- sqlite (a tiny yet powerful SQL database)
- tcllib (a collection of pure-Tcl packages - see below)
- TclOO (the canonical object-orientation extension from 8.5)
- tcltcc (a built-in C compiler - see below)
- TclX (collection of system-related extensions, like signal handling)
- tdom (XML parser, SAX or DOM, with XPath query support)
- Tk (the cross-platform GUI toolkit, see below)
- tkcon (a vastly extended console)
- XOTcl (advanced dynamic OO extension)

A little example package

The following script creates the trivial but educational package futil, which in a namespace of the same name implements two procs for reading and writing complete text files, and the little introspection helper function, futil::?. The command to register the package (package provide) is executed only after all went well - this way, buggy source code, which raises an error during package require, will not be registered. (Other bugs you'd have to spot and fix for yourself...)

Common Tcl distribution practice has the good habit of profound testing, typically in a separate test directory. On the other hand, including a self-test in the same file with the code makes editing easier, so after the package provide comes a section only executed if this file is sourced as a top- level script, which exercises the commands defined in futil. Whether the string read should be equal to the string written is debatable - the present implementation appends \n to the string if it doesn't end in one, because some tools complain or misbehave if they don't see a final newline.

If the tests do not run into an error either, even the required construction of a package index is fired - assuming the simplified case that the directory contains only one package. Otherwise, you'd better remove these lines, and take care of index creation yourself.

A script that uses this package will only have to contain the two lines

```
lappend ::auto_path <directory of this file>
package require futil
```

You can even omit the first line, if you install (copy) the directory with the source and pkgIndex.tcl below ${tcl_install_directory}/lib. }

```
namespace eval futil {
    set version 0.1
}
```

But now back to the single script that makes up the package (it would make sense to save it as *futil.tcl*). We provide a *read* and a *write* function in the *futil* namespace, plus a little introspection function *?* that returns the names of available functions:

```
proc futil::read {filename} {
   set fp [open $filename]
   set string [::read $fp] ;# prevent name conflict with itself
   close $fp
   return $string
}
proc futil::write {filename string} {
   set fp [open $filename w]
   if {[string index $string end]!="\n"} {append string \n}
   puts -nonewline $fp $string
   close $fp
}
proc futil::? {} {lsort [info procs ::futil::*]}
# If execution comes this far, we have succeeded ;-)
package provide futil $futil::version
```

Self-test code if {[info ex argv0] && [file tail [info script]] == [file tail $argv0]} { puts "package futil contains [futil::?]" set teststring { This is a teststring in several lines...} puts teststring:'$teststring' futil::write test.tmp $teststring set string2 [futil::read test.tmp] puts string2:'$string2' puts "strings are [expr {$teststring==$string2? {}:{not}}] equal"

```
   file delete test.tmp ;# don't leave traces of testing
```

```
   # Simple index generator, if the directory contains only this package
   pkg_mkIndex -verbose [file dirn [info scr]] [file tail [info scr]]
}
```

Tcllib

Tcllib is a collection of packages in pure Tcl. It can be obtained from sourceForge, but is also part of ActiveTcl. The following list of contained packages may not be complete, as Tcllib is steadily growing...

- aes - Advanced Encryption Standard.
- asn - asn.1 BER encoder/decoder
- http/autoproxy - code to automate the use of HTTP proxy servers
- base64 - Base64 encoding and decoding of strings and files.
- bee - BitTorrent serialization encoder/decoder.
- bibtex - Neil Madden's parser for bibtex files. Not fully complete yet, therefore not set for installation.
- calendar - Calendar operations (see also tcllib calendar module).
- cmdline - Various form of command line and option processing.
- comm - Socket based interprocess communication. Emulates the form of Tk's send command.
- control - procedures for tcl flow structures such as assert, do/until, do/while, no-op
- counter - procedures for counters and histograms
- crc - Computation of various CRC checksums for strings and files.
- csv - manipulate comma separated value data

- des - Data Encryption Standard. ::DES::des (not yet installed)
- dns - interact with the Domain Name Service. dns::address, dns::cleanup, dns::cname, dns::configure, dns::name, dns::reset, dns::resolve, dns::status, dns::wait,
- doctools - System for writing manpages/documentation in a simple, yet powerful format.
- exif - exif::analyze exif::fieldnames
- fileutil - Utilities for operating on files, emulating various unix command line applications (cat, find, file(type), touch, ...).
- ftp - Client side implementation of FTP (File Transfer Protocol). In dire need of a rewrite.
- ftpd - Server side implementation of FTP
- grammar_fa - Operations on finite automatons.
- html - generate HTML from a Tcl script. html::author, html::author, html::bodyTag, html::cell, html::checkbox, html::checkSet, html::checkValue, html::closeTag, html::default, html::description, html::description, html::end, html::eval, html::extractParam, html::font, html::for, html::foreach, html::formValue, html::getFormInfo, html::getTitle, html::h, html::h1, html::h2, html::h3, html::h4, html::h5, html::h6, html::hdrRow, html::head, html::head, html::headTag, html::if, html::init, html::init, html::keywords, html::keywords, html::mailto, html::meta, html::meta, html::minorList, html::minorMenu, html::openTag, html::paramRow, html::passwordInput, html::passwordInputRow, html::radioSet, html::radioValue, html::refresh, html::refresh, html::row, html::select, html::selectPlain, html::set, html::submit, html::tableFromArray, html::tableFromList, html::tagParam, html::textarea, html::textInput, html::textInputRow, html::title, html::title, html::urlParent, html::varEmpty, html::while,
- htmldoc - This is not a true module but the place where tcllib 1.3 installed the tcllib documentation in HTML format.
- htmlparse - procedures to permit limited maniulation of strings containing HTML. ::htmlparse::parse, ::htmlparse::debugCallback, ::htmlparse::mapEscapes, ::htmlparse::2tree, ::htmlparse::removeVisualFluff, ::htmlparse::removeFormDefs,
- ident - RFC 1413 ident client protocol implementation
- imap4 - currently undocumented code for interacting with an IMAP server
- inifile - code to manipulate an initialization file. ::ini::open, ::ini::close, ::ini::commit, ::ini::sections, ::ini::keys, ::ini::value
- dns/ip - Manipulation of IP addresses. ::ip::version, ::ip::is, ::ip::normalize, ::ip::equal, ::ip::prefix
- irc - Internet Relay Chat procedures. irc::config, irc::connection,
- javascript - generate Javascript for including in HTML pages. javascript::BeginJS, javascript::EndJS, javascript::MakeMultiSel, javascript::MakeClickProc, javascript::makeSelectorWidget, javascript::makeSubmitButton, javascript::makeProtectedSubmitButton, javascript::makeMasterButton, javascript::makeParentCheckbox, javascript::makeChildCheckbox
- jpeg - edit comment blocks, get image dimensions and information, read exif data of images in the JPG format
- ldap - Client side implementation of LDAP (Lightweight Directory Access Protocol).
- log - general procedures for adding log entries to files ::log::levels, ::log::logMsg, ::log::lv2longform, ::log::lv2color,::log::lv2priority,
- logger - ::logger::walk, ::logger::services, ::logger::enable, ::logger::disable (part of the log module)

- math - general mathematical procedures. ::math::calculus, ::math::combinatorics, ::math::cov, ::math::fibonacci, ::math::integrate, ::math::interpolate, ::math::max, ::math::mean, ::math::min, ::math::optimize, ::math::product, ::math::random, ::math::sigma, ::math::statistics, ::math::stats, ::math::sum
- md4 - ::md4::md4, ::md4::hmac, ::md4::MD4Init, ::md4::MD4Update, ::md4::MD4Final
- md5 - [fill in the description of this module] ::md5::md5, ::md5::hmac, ::md5::test, ::md5::time, ::md5::<<<
- md5crypt - ::md5crypt::md5crypt, ::md5crypt::aprcrypt
- mime - ::mime::initialize, ::mime::parsepart, ::mime::finalize, ::smtp::sendmessage
- multiplexer - [fill in the external interfaces]
- ncgi - procedures for use in a CGI application. ::ncgi::reset, ::ncgi::urlStub, ::ncgi::urlStub
- nntp - routines for interacting with a usenet news server. ::nntp::nntp, ::nntp::NntpProc, ::nntp::NntpProc, ::nntp::okprint, ::nntp::message,
- ntp - network time protocol procedure ::ntp::time
- png - edit comment blocks, get image dimensions and information for Portable Network Graphics format.
- pop3 - Post Office Protocol functions for reading mail from a pop3 server. ::pop3::open, ::pop3::close, ::pop3::status,
- pop3d - Post Office Protocol Server. pop3d::new
- profiler - ::profiler::tZero, ::profiler::tMark, ::profiler::stats, ::profiler::Handler, ::profiler::profProc, ::profiler::init
- rc4 - stream encryption. ::rc4::rc4
- report - format text in various report styles. ::report::report , ::report::defstyle, ::report::rmstyle,
- sha1 - ::sha1::sha1, ::sha1::hmac
- smtpd - ::smtpd::start, ::smtpd::stop, ::smtpd::configure, ::smtpd::cget
- snit - Snit's Not Incr Tcl - OO package. Delegation based. ::snit::type, ::snit::widget, ::snit::widgetadaptor
- soundex::knuth - string matching based on theoretical sound of the letters
- stooop - OO package. stooop::class, stooop::virtual, stooop::new, stooop::delete, stooop::classof
- struct1 - Version 1 of struct (see below), provided for backward compatibility.

  struct::list, ::struct::graph, ::struct::matrix, ::struct::queue, ::struct::stack, ::struct::Tree, ::struct::record, ::struct::skiplist, ::struct::prioqueue, new: ::struct::sets

- tar - untar, list, and stat files in tarballs and create new tarballs
- textutil - Utilities for working with larger bodies of texts. textutil::expand - the core for the expand macro processor.
- tie - Persistence for Tcl arrays.
- treeql - Tree Query Language, inspired by COST.
- uri - Handling of uri/urls (splitting, joining, ...)
- uuid - Creation of unique identifiers.

TclOO

TclOO is a loadable package to provide a foundation for object orientation, designed so that specific OO flavors like Itcl, Snit, or XOTcl can build on it. But it's a usable OO system in itself, offering classes, multiple inheritance, mixins and filters. Here is some example code to give you an impression:

#!/usr/bin/env tclsh85 package require TclOO namespace import oo::* class create Account { constructor { {ownerName undisclosed}} {

```
        my variable total overdrawLimit owner
        set total 0
        set overdrawLimit 10
        set owner $ownerName
    }
    method deposit amount {
        my variable total
        set total [expr {$total + $amount}]
    }
    method withdraw amount {
        my variable {*}[info object vars [self]] ;# "auto-import" all variables
        if {($amount - $total) > $overdrawLimit} {
            error "Can't overdraw - total: $total, limit: $overdrawLimit"
        }
        set total [expr {$total - $amount}]
    }
    method transfer {amount targetAccount} {
        my variable total
        my withdraw $amount
        $targetAccount deposit $amount
        set total
    }
    destructor {
        my variable total
        if {$total} {puts "remaining $total will be given to charity"}
    }
}
```

tcltcc

Tcltcc is a loadable package that wraps the Tiny C compiler (tcc) for use with Tcl. It can be used to

- compile C code directly to memory
- produce dynamic loadable libraries (DLLs) or executable files.

Convenience functions generate wrapper code, so that the user needs only write the really substantial C code.

Examples:

Wrap a C function into a Tcl command "on the fly" (here shown in an interactive session):

```
% package require tcc
0.2
% namespace import tcc::*
% cproc sigmsg {int i} char* {return Tcl_SignalMsg(i);}
% sigmsg 4
illegal instruction
```

Produce a DLL with a fast implementation of Fibonacci numbers:

```
% set d [tcc::dll]
% $d ccode {
     static int fib(int n) {return n <= 2? 1 : fib(n-1) + fib(n-2);}
  }
% $d cproc fiboy {int n} int {return fib(n);}
% $d write -name fiboy
% load fiboy[info sharedlibextension]
% fiboy 20
6765
```

Produce a tclsh with an extra *square* command:

```
% set code [tcc::wrapCmd square {double x} double x_square {return x*x;}]
% append code {
    int AppInit(Tcl_Interp *interp) {
       int rc;
       rc = Tcl_CreateObjCommand(interp,"square",x_square,NULL,NULL);
           return Tcl_Init(interp);
    }
    int main(int argc, char *argv[]) {
        Tcl_Main(argc, argv, AppInit);
        return 0;
    }
}
% tcc $::tcc::dir exe t
% t add_file    $::tcc::dir/c/crt1.c
% t add_library tcl8.5
% t compile     $code
% t output_file mytclsh.exe
% exec mytclsh.exe {<<puts [square 5]}
25.0
```

Tcltcc is open source, LGPL licensed, available at `http://code.google.com/p/tcltcc/` . The full functionality is at the current early stage (October 2007) only available on Windows 95/XP platforms, but in-memory compilation works on Linux too.

tDOM

tDOM is a popular extension for XML/HTML processing, allowing both SAX-style "on-the-fly" parsing and the DOM approach of representing a whole XML element in memory.

Here is an example of a SAX-style application. The **expat** parser that comes with tDOM is instrumented with callbacks for element start, character data, and processing instructions. Elements, attributes, characters and processing instructions are counted, plus a tally for each element type is done.

#!/usr/bin/env tclsh package require tdom #--- Callbacks for certain parser events proc el {name attlist} { global g incr ::nEl incr ::nAtt [llength $attlist] inc g($name) } proc ch data { incr ::nChar [string length $data] } proc pi {target data} { incr ::nPi } proc inc {varName {increment 1}} {

```
    upvar 1 $varName var
    if {![info exists var]} {set var 0}
    incr var $increment
}
#--- "main" loop
if ![llength $argv] {puts "usage: $argv0 file..."}
foreach file $argv {
    foreach i {nEl nAtt nChar nPi} {set $i 0} ;# reset counters
    array unset g
    set p [expat -elementstartcommand el \
           -characterdatacommand         ch \
           -processinginstructioncommand  pi ]
    if [catch {$p parsefile $file} res] {
                puts "error:$res"
    } else {
       puts "$file:\n$nEl elements, $nAtt attributes, $nChar characters,\
           $nPi processing instructions"
       foreach name [lsort [array names g]] {
           puts [format %-20s%7d $name $g($name)]
       }
    }
    $p free
}
```

9

0.3 expr: the arithmetic & logical unit

0.3.1 Overview

Arithmetic and logical operations (plus some string comparisons) are in Tcl concentrated in the **expr** command. It takes one or more arguments, evaluates them as an expression, and returns the result. The language of the **expr** command (also used in condition arguments of the **if**, **for**, **while** commands) is basically equivalent to C's expressions, with infix operators and functions. Unlike C, references to variables have to be done with *$var*. Examples:

```
set a [expr {($b + sin($c))/2.}]
if {$a > $b && $b > $c} {puts "ordered"}
for {set i 10} {$i >= 0} {incr i -1} {puts $i...} ;# countdown
```

The difference between Tcl syntax and *expr* syntax can be contrasted like this:

```
[f $x $y]  ;# Tcl:  embedded command
 f($x,$y)  ;# expr: function call, comma between arguments
```

In another contrast to Tcl syntax, whitespace between "words" is optional (but still recommended for better readability :) And string constants must always be quoted (or

9 http://en.wikibooks.org/wiki/Category%3A

braced if you wish):

```
if {$keyword eq "foo"} ...
```

Then again, Tcl commands can always be embedded into expressions, in square brackets as usual: proc max {x y} {expr {$x>$y? $x: $y}}

```
expr {[max $a $b] + [max $c $d]}
```

In expressions with numbers of mixed types, integers are coerced to doubles:

```
% expr 1+2.
3.0
```

It is important to know that division of two integers is done as integer division:

```
% expr 1/2
0
```

You get the (probably expected) floating-point division if at least one operand is *double*:

```
% expr 1/2.
0.5
```

If you want to evaluate a string input by the user, but always use floating-point division, just transform it, before the call to expr, by replacing "/" with "*1./" (multiply with floating-point 1. before every division):

```
expr [string map {/ *1./} $input]
```

0.3.2 Brace your expressions

In most cases it is safer and more efficient to pass a single braced argument to *expr*. Exceptions are:

- no variables or embedded commands to substitute
- operators or whole expressions to substitute

The reason is that the Tcl parser parses unbraced expressions, while expr parses that result again. This may result in success for malicious code exploits:

```
% set e {[file delete -force *]}
% expr $e   ;# will delete all files and directories
% expr {$e} ;# will just return the string value of e
```

That braced expressions evaluate much faster than unbraced ones, can be easily tested: % proc unbraced x {expr $x*$x} % proc braced x {expr {$x*$x}}

```
% time {unbraced 42} 1000
197 microseconds per iteration
% time {braced 42} 1000
34 microseconds per iteration
```

The precision of the string representation of floating-point numbers is also controlled by the tcl_precision variable. The following example returns nonzero because the second term was clipped to 12 digits in making the string representation:

```
% expr 1./3-[expr 1./3]
3.33288951992e-013
```

while this braced expression works more like expected:

```
% expr {1./3-[expr 1./3]}
0.0
```

0.3.3 Operators

Arithmetic, bitwise and logical operators are like in C, as is the conditional operator found in other languages (notably C):

- c?a:b -- if c is true, evaluate a, else b

The conditional operator can be used for compact functional code (note that the following example requires Tcl 8.5 so that fac() can be called inside its own definition): % proc tcl::mathfunc::fac x {expr {$x<2? 1 : $x*fac($x-1)}}

```
% expr fac(5)
120
```

Arithmetic operators

The arithmetic operators are also like those found in C:

- + addition
- - (binary: subtraction. unary: change sign)
- * multiplication
- / (integer division if both operands are integer
- % (modulo, only on integers)
- ** power (available from Tcl 8.5)

Bitwise operators

The following operators work on integers only:

- & (AND)
- | (OR)
- ^ (XOR)
- ~ (NOT)
- << shift left
- >> shift right

Logical operators

The following operators take integers (where 0 is considered false, everything else true) and return a truth value, 0 or 1:

- && (and)
- || (or)
- ! (not - unary)

Comparison operators

If operands on both side are numeric, these operators compare them as numbers. Otherwise, string comparison is done. They return a truth value, 0 (false) or 1 (true):

- == equal
- != not equal
- > greater than
- >= greater or equal than
- < less than
- <= less or equal than

As truth values are integers, you can use them as such for further computing, as the sign function demonstrates: proc sgn x {expr {($x>0) - ($x<0)}}

```
% sgn 42
1
% sgn -42
-1
% sgn 0
0
```

String operators

The following operators work on the string representation of their operands:

- eq string-equal
- ne not string-equal

Examples how "equal" and "string equal" differ:

```
% expr {1 == 1.0}
1
% expr {1 eq 1.0}
0
```

List operators

From Tcl 8.5, the following operators are also available:

- a **in** b - 1 if a is a member of list b, else 0
- a **ni** b - 1 if a is not a member of list b, else 0

Before 8.5, it's easy to write an equivalent function proc in {list el} {expr {[lsearch -exact $list $el]>=0}} Usage example:

```
if [in $keys $key] ...
```

which you can rewrite, once 8.5 is available wherever your work is to run, with

```
if {$key in $keys} ...
```

0.3.4 Functions

The following functions are built-in:

- abs(x) - absolute value
- acos(x) - arc cosine. acos(-1) = 3.14159265359 (Pi)
- asin(x) - arc sine
- atan(x) - arc tangent
- atan2(y,x)
- ceil(x) - next-highest integral value
- cos(x) - cosine
- cosh(x) - hyperbolic cosine
- double(x) - convert to floating-point number
- exp(x) - e to the x-th power. exp(1) = 2.71828182846 (Euler number, e)
- floor(x) - next-lower integral value
- fmod(x,y) - floating-point modulo
- hypot(y,x) - hypotenuse (sqrt($y*$y+$x*$x), but at higher precision)
- int(x) - convert to integer (32-bit)
- log(x) - logarithm to base e
- log10(x) - logarithm to base 10
- pow(x,y) - x to the y-th power
- rand() - random number > 0.0 and < 1.0
- round(x) - round a number to nearest integral value

- sin(x) - sine
- sinh(x) - hyperbolic sine
- sqrt(x) - square root
- srand(x) - initialize random number generation with seed x
- tan(x) - tangent
- tanh(x) - hyperbolic tangent
- wide(x) - convert to wide (64-bit) integer

Find out which functions are available with *info functions*:

```
% info functions
round wide sqrt sin double log10 atan hypot rand abs acos atan2 srand
sinh floor log int tanh tan asin ceil cos cosh exp pow fmod
```

0.3.5 Exporting expr functionalities

If you don't want to write expr {$x+5}[10] every time you need a little calculation, you can easily export operators as Tcl commands:

```
foreach op {+ - * / %} {proc $op {a b} "expr {\$a $op \$b}"}
```

After that, you can call these operators like in LISP:

```
% + 6 7
13
% * 6 7
42
```

Of course, one can refine this by allowing variable arguments at least for + and *, or the single-argument case for -: proc - {a {b ""}} {expr {$b eq ""? -$a: $a-$b}}

Similarly, expr functions can be exposed:

```
foreach f {sin cos tan sqrt} {proc $f x "expr {$f($x)}"}
```

In Tcl 8.5, the operators can be called as commands in the *::tcl::mathop* namespace:

```
% tcl::mathop::+ 6 7
13
```

You can import them into the current namespace, for shorthand math commands:

10 http://en.wikibooks.org/wiki/expr%20%7B%24x%2B5%7D

```
% namespace import ::tcl::mathop::*
% + 3 4 ;# way shorter than [expr {3 + 4}]
7
% * 6 7
42
```

0.3.6 User-defined functions

From Tcl 8.5, you can provide procs in the *::tcl::mathfunc* namespace, which can then be used inside *expr* expressions: % proc tcl::mathfunc::fac x {expr {$x < 2? 1: $x * fac($x-1)}}

```
 % expr fac(100)
 93326215443944152681699238856266700490715968264381621468592963895217599993229
15608941463976156518286253697920827223758251185210916864000000000000000000000000
```

This is especially useful for recursive functions, or functions whose arguments need some expr calculations:

% proc ::tcl::mathfunc::fib n {expr {$n<2? 1: fib($n-2)+fib($n-1)}}

```
% expr fib(6)
13
```

11

0.4 Interaction and debugging

Tcl itself is quite a good teacher. Don't be afraid to do something wrong - it will most often deliver a helpful error message. When tclsh is called with no arguments, it starts in an interactive mode and displays a "%" prompt. The user types something in and sees what comes out: either the result or an error message.

Trying isolated test cases interactively, and pasting the command into the editor when satisfied, can greatly reduce debugging time (there is no need to restart the application after every little change - just make sure it's the right one, before restarting.)

0.4.1 A quick tour

Here's a commented session transcript:

```
% hello
invalid command name "hello"
```

11 http://en.wikibooks.org/wiki/Category%3A

OK, so we're supposed to type in a command. Although it doesn't look so, here's one:

```
% hi
     1  hello
    2  hi
```

Interactive tclsh tries to guess what we mean, and "hi" is the unambiguous prefix of the "history" command, whose results we see here. Another command worth remembering is "info":

```
% info
wrong # args: should be "info option ?arg arg ...?"
```

The error message tells us there should be at least one option, and optionally more arguments.

```
% info option
bad option "option": must be args, body, cmdcount, commands, complete, default,
exists, functions, globals, hostname, level, library, loaded, locals,
nameofexecutable,
patchlevel, procs, script, sharedlibextension, tclversion, or vars
```

Another helpful error: "option" is not an option, but the valid ones are listed. To get information about commands, it makes sense to type the following:

```
% info commands
tell socket subst lremove open eof tkcon_tcl_gets pwd glob list exec pid echo
dir auto_load_index time unknown eval lrange tcl_unknown fblocked lsearch gets
auto_import case lappend proc break dump variable llength tkcon auto_execok
return
pkg_mkIndex linsert error bgerror catch clock info split thread_load loadvfs
array
if idebug fconfigure concat join lreplace source fcopy global switch which
auto_qualify
update tclPkgUnknown close clear cd for auto_load file append format tkcon_puts
alias
what read package set unalias pkg_compareExtension binary namespace scan edit
trace seek
while flush after more vwait uplevel continue foreach lset rename tkcon_gets
fileevent
regexp tkcon_tcl_puts observe_var tclPkgSetup upvar unset encoding expr load
regsub history
exit interp puts incr lindex lsort tclLog observe ls less string
```

Oh my, quite many... How many?

```
% llength [info commands]
115
```

Now for a more practical task - let's let Tcl compute the value of Pi.

```
% expr acos(-1)
3.14159265359
```

Hm.. can we have that with more precision?

```
% set tcl_precision 17
17
% expr acos(-1)
3.1415926535897931
```

Back to the first try, where "hello" was an invalid command. Let's just create a valid one:

```
% proc hello {} {puts Hi!}
```

Silently acknowledged. Now testing:

```
% hello
Hi!
```

0.4.2 Errors are exceptions

What in Tcl is called **error** is in fact more like an *exception* in other languages - you can deliberately raise an error, and also **catch** errors. Examples:

```
if {$username eq ""} {error "please specify a user name"}
```

```
if [catch {open $filename w} fp] {
   error "$filename is not writable"
}
```

One reason for errors can be an undefined command name. One can use this playfully, together with *catch*, as in the following example of a multi-loop *break*, that terminates the two nested loops when a matrix element is empty:

```
if [catch {
   foreach row $matrix {
      foreach col $row {
          if {$col eq ""} throw
      }
   }
}] {puts "empty matrix element found"}
```

The *throw* command does not exist in normal Tcl, so it throws an error, which is caught by the *catch* around the outer loop.

The errorInfo variable

This global variable provided by Tcl contains the last error message and the traceback of the last error. Silly example:

```
% proc foo {} {bar x}
% proc bar {input} {grill$input}
% foo
invalid command name "grillx"
```

```
% set errorInfo
invalid command name "grillx"
   while executing
"grill$input"
   (procedure "bar" line 1)
   invoked from within
"bar x"
   (procedure "foo" line 1)
   invoked from within
"foo"
```

If no error has occurred yet, *errorInfo* will contain the empty string.

The errorCode variable

In addition, there is the *errorCode* variable that returns a list of up to three elements:

- category (POSIX, ARITH, ...)
- abbreviated code for the last error
- human-readable error text

Examples:

```
% open not_existing
couldn't open "not_existing": no such file or directory
% set errorCode
POSIX ENOENT {no such file or directory}
```

```
% expr 1/0
divide by zero
% set errorCode
ARITH DIVZERO {divide by zero}
```

```
% foo
invalid command name "foo"
% set errorCode
NONE
```

0.4.3 Tracing procedure calls

For a quick overview how some procedures are called, and when, and what do they return, and when, the *trace execution* is a valuable tool. Let's take the following factorial function as example: proc fac x {expr {$x<2? 1 : $x * [fac [incr x -1]]}} We need to supply a handler that will be called with different numbers of arguments (two on enter, four on leave). Here's a very simple one:

```
proc tracer args {puts $args}
```

Now we instruct the interpreter to trace *enter* and *leave* of *fac*:

```
trace add execution fac {enter leave} tracer
```

Let's test it with the factorial of 7:

```
fac 7
```

which gives, on stdout:

```
{fac 7} enter
{fac 6} enter
{fac 5} enter
{fac 4} enter
{fac 3} enter
{fac 2} enter
{fac 1} enter
{fac 1} 0 1 leave
{fac 2} 0 2 leave
{fac 3} 0 6 leave
{fac 4} 0 24 leave
{fac 5} 0 120 leave
{fac 6} 0 720 leave
{fac 7} 0 5040 leave
```

So we can see how recursion goes down to 1, then returns in backward order, stepwise building up the final result. The 0 that comes as second word in "leave" lines is the return status, 0 being TCL_OK.

0.4.4 Stepping through a procedure

To find out how exactly a proc works (and what goes wrong where), you can also register commands to be called before and after a command inside a procedure is called (going down transitively to all called procs). You can use the following *step* and *interact* procedures for this:

proc step {name {yesno 1}} {

```
    set mode [expr {$yesno? "add" : "remove"}]
    trace $mode execution $name {enterstep leavestep} interact
}
```

```
proc interact args {
    if {[lindex $args end] eq "leavestep"} {
        puts ==>[lindex $args 2]
        return
    }
    puts -nonewline "$args --"
    while 1 {
        puts -nonewline "> "
        flush stdout
        gets stdin cmd
        if {$cmd eq "c" || $cmd eq ""} break
        catch {uplevel 1 $cmd} res
        if {[string length $res]} {puts $res}
    }
}
```

```
#-----------------------------Test case, a simple string reverter:
proc sreverse str {
    set res ""
    for {set i [string length $str]} {$i > 0} {} {
        append res [string index $str [incr i -1]]
    }
    set res
}
```

```
#-- Turn on stepping for sreverse[12]:
step sreverse
sreverse hello
```

```
#-- Turn off stepping (you can also type this command from inside interact):
step sreverse 0
puts [sreverse Goodbye]
```

The above code gives the following transcript when sourced into a tclsh:

{set res {}} enterstep --> ==> {for {set i [string length $str]} {$i > 0} {} { append res [string index $str [incr i -1]] }} enterstep -->

```
{string length hello} enterstep -->
==>5
{set i 5} enterstep -->
==>5
{incr i -1} enterstep -->
==>4
{string index hello 4} enterstep -->
==>o
{append res o} enterstep -->
==>o
{incr i -1} enterstep -->
==>3
{string index hello 3} enterstep -->
==>l
{append res l} enterstep -->
==>ol
{incr i -1} enterstep -->
```

```
==>2
{string index hello 2} enterstep -->
==>l
{append res l} enterstep -->
==>oll
{incr i -1} enterstep -->
==>1
{string index hello 1} enterstep -->
==>e
{append res e} enterstep -->
==>olle
{incr i -1} enterstep -->
==>0
{string index hello 0} enterstep -->
==>h
{append res h} enterstep -->
==>olleh
==>
{set res} enterstep -->
==>olleh
eybdooG
```

0.4.5 Debugging

The simplest way to inspect why something goes wrong is inserting a *puts* command before the place where it happens. Say if you want to see the values of variables x and y, just insert

```
puts x:$x,y:$y
```

(if the string argument contains no spaces, it needs not be quoted). The output will go to stdout - the console from where you started the script. On Windows or Mac, you might need to add the command

```
console show
```

to get the substitute console Tcl creates for you, when no real one is present.

If at some time you want to see details of what your program does, and at others not, you can define and redefine a *dputs* command that either calls *puts* or does nothing: proc d+ {} {proc dputs args {puts $args}} proc d- {} {proc dputs args {}}

```
d+ ;# initially, tracing on... turn off with d-
```

For more debugging comfort, add the proc *interact* from above to your code, and put a call to *interact* before the place where the error happens. Some useful things to do at such a debugging prompt:

```
info level 0    ;# shows how the current proc was called
info level      ;# shows how deep you are in the call stack
uplevel 1 ...   ;# execute the ... command one level up, i.e. in the caller of
```

```
the current proc
set ::errorInfo ;# display the last error message in detail
```

0.4.6 Assertions

Checking data for certain conditions is a frequent operation in coding. Absolutely intolerable conditions can just throw an error:

```
  if {$temperature > 100} {error "ouch... too hot!"}
```

Where the error occurred is evident from ::errorInfo, which will look a bit clearer (no mention of the error command) if you code

```
  if {$temperature > 100} {return -code error "ouch... too hot!"}
```

If you don't need hand-crafted error messages, you can factor such checks out to an assert command:

```
proc assert condition {
   set s "{$condition}"
   if {![uplevel 1 expr $s]} {
       return -code error "assertion failed: $condition"
   }
}
```

Use cases look like this:

```
  assert {$temperature <= 100}
```

Note that the condition is reverted - as "assert" means roughly "take for granted", the positive case is specified, and the error is raised if it is not satisfied.

Tests for internal conditions (that do not depend on external data) can be used during development, and when the coder is sure they are bullet-proof to always succeed, (s)he can turn them off centrally in one place by defining

```
proc assert args {}
```

This way, assertions are compiled to no bytecode at all, and can remain in the source code as a kind of documentation.

If assertions are tested, it only happens at the position where they stand in the code. Using a trace, it is also possible to specify a condition once, and have it tested whenever a variable's value changes:

```
proc assertt {varName condition} {
   uplevel 1 [list trace var $varName w "assert $condition ;#"]
}
```

The ";#" at the end of the trace causes the additional arguments name element op, that are appended to the command prefix when a trace fires, to be ignored as a comment.

Testing:

```
% assertt list {[llength $list]<10}
% set list {1 2 3 4 5 6 7 8}
1 2 3 4 5 6 7 8
% lappend list 9 10
can't set "list": assertion failed: 10<10
```

The error message isn't as clear as could be, because the [llength $list] is already substituted in it. But I couldn't find an easy solution to that quirk in this breakfast fun project - backslashing the $condition in the assertt code sure didn't help. Better ideas welcome.

To make the assertion condition more readable, we could quote the condition one more time,i.e

```
% assertt list
UNKNOWN TEMPLATE [llength list] < 10

% set list {1 2 3 4 5 6 7 8}
1 2 3 4 5 6 7 8
% lappend list 9 10
can't set "list": assertion failed: [llength $list]<10
%
```

In this case,when trace trigger fires, the argument for assert is {[llength $list]<10}.

In any case, these few lines of code give us a kind of bounds checking - the size of Tcl's data structures is in principle only bounded by the available virtual memory, but runaway loops may be harder to debug, compared to a few assertt calls for suspicious variables:

```
assertt aString {[string length $aString]<1024}
```

or

```
assertt anArray {[array size anArray] < 1024*1024}
```

Tcllib has a control::assert with more bells and whistles.

0.4.7 A tiny testing framework

Bugs happen. The earlier found, the easier for the coder, so the golden rule "Test early. Test often" should really be applied.

One easy way is adding self-tests to a file of Tcl code. When the file is loaded as part of a library, just the proc definitions are executed. If however you feed this file directly to a tclsh, that fact is detected, and the "e.g." calls are executed. If the result is not the one expected, this is reported on stdout; and in the end, you even get a little statistics.

Here's a file that implements and demonstrates "e.g.":

PROLOG -- self-test: if this file is sourced at top level: if {[info exists argv0]&&[file tail [info script]] eq [file tail $argv0]} { set Ntest 0; set Nfail 0 proc e.g. {cmd -> expected} { incr ::Ntest catch {uplevel 1 $cmd} res if {$res ne $expected} { puts "$cmd -> $res, expected $expected" incr ::Nfail } } } else {proc e.g. args {}} ;# does nothing, compiles to nothing

Your code goes here, with e.g. tests following proc sum {a b} {expr {$a+$b}}

```
e.g. {sum 3 4} -> 7
```

proc mul {a b} {expr {$a*$b}}

```
e.g. {mul 7 6} -> 42
```

```
# testing a deliberate error (this way, it passes):
e.g. {expr 1/0} -> "divide by zero"
```

```
## EPILOG -- show statistics:
e.g. {puts "[info script] : tested $::Ntest, failed $::Nfail"} -> ""
```

0.4.8 Guarded proc

In more complex Tcl software, it may happen that a procedure is defined twice with different body and/or args, causing hard-to-track errors. The Tcl command *proc* itself doesn't complain if it is called with an existing name. Here is one way to add this functionality. Early in your code, you overload the proc command like this:

```
 rename proc _proc
 _proc proc {name args body} {
     set ns [uplevel namespace current]
     if {[info commands $name]!="" || [info commands ${ns}::$name]!=""} {
         puts stderr "warning: [info script] redefines $name in $ns"
     }
     uplevel [list _proc $name $args $body]
 }
```

From the time that is sourced, any attempt to override a proc name will be reported to stderr (on Win-wish, it would show on the console in red). You may make it really strict by adding an "exit" after the "puts stderr ...", or throw an error.

Known feature: proc names with wildcards will run into this trap, e.g.

```
proc * args {expr [join $args *]*1}
```

will always lead to a complaint because "*" fits any proc name. Fix (some regsub magic on 'name') left as an exercise.

0.4.9 Windows wish console

While on Unixes, the standard channels *stdin*, *stdout*, and *stderr* are the same as the terminal you started wish from, a Windows *wish* doesn't typically have these standard channels (and is mostly started with double-click anyway). To help this, a console was added that takes over the standard channels (stderr even coming in red, stdin in blue). The console is normally hidden, but can be brought up with the command

```
console show
```

You can also use the partially documented "console" command. "console eval

1 Contributors

Edits	User
12	Adrignola[1]
1	Albmont[2]
1	Bovineone[3]
12	Darklama[4]
1	Derbeth[5]
1	Hoxel[6]
1	Jfmantis[7]
9	Jguk[8]
5	Sigma 7[9]
7	Snarius[10]
1	Soulaegis[11]
3	Specs112[12]
231	Suchenwi[13]
1	SwiftBot[14]
1	Tegel[15]
11	Ysangkok[16]

1 http://en.wikibooks.org/wiki/User:Adrignola
2 http://en.wikibooks.org/wiki/User:Albmont
3 http://en.wikibooks.org/wiki/User:Bovineone
4 http://en.wikibooks.org/wiki/User:Darklama
5 http://en.wikibooks.org/wiki/User:Derbeth
6 http://en.wikibooks.org/wiki/User:Hoxel
7 http://en.wikibooks.org/wiki/User:Jfmantis
8 http://en.wikibooks.org/wiki/User:Jguk
9 http://en.wikibooks.org/wiki/User:Sigma_7
10 http://en.wikibooks.org/wiki/User:Snarius
11 http://en.wikibooks.org/wiki/User:Soulaegis
12 http://en.wikibooks.org/wiki/User:Specs112
13 http://en.wikibooks.org/wiki/User:Suchenwi
14 http://en.wikibooks.org/wiki/User:SwiftBot
15 http://en.wikibooks.org/wiki/User:Tegel
16 http://en.wikibooks.org/wiki/User:Ysangkok

List of Figures

- GFDL: Gnu Free Documentation License. http://www.gnu.org/licenses/fdl.html
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/deed.en
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- GPL: GNU General Public License. http://www.gnu.org/licenses/gpl-2.0.txt
- LGPL: GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. http://artlibre.org/licence/lal/de
- CFR: Copyright free use.

- EPL: Eclipse Public License. http://www.eclipse.org/org/documents/epl-v10.php

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses[17]. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrower.

17 Chapter 2 on page 69

2 Licenses

2.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

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

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

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

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

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

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices". * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

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

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

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

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

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

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

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

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

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

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

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

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

2.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful 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. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

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. (Thus, 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. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

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 Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document 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, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. 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, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include 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, PostScript or PDF 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.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. 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. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) 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 computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. 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. 4. 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:

* A. 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. * B. 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 fewer than five), unless they release you from this requirement. * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. 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. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve 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. * J. 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. * K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. 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. * M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

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. 5. 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, and that you preserve all their Warranty Disclaimers.

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". 6. 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. 7. 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, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which 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 half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. 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, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. 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. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. 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 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

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.

2.3 GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or * b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. * e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. * b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

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

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.