Tcl/Tk

Measurement guide

written by Balázs Gaál

1.Introduction

1.1.Review

Tcl and Tk are two program packets, that help running and developing X-Windows based graphic user surface in UNIX environment. (There is also an experimental implementation for MS-Windows.) This guide provides a short review about the Tcl language, basis of Tk, and it will demonstrate an expansion as an example, with that we can realize a network communication with the help of Tcp/IP sockets. This description assumes a minimal precognition of UNIX, X-Windows, and TCP/IP.

Tcl (Tool Command Language) is a high-level „script" language based on a simple interpreter, which can be compared mostly to *perl* language, or to the UNIX *C-Shell* script. Variants, arrays, controlling structures, processes can be used in it. One of its most important features, that a *Tcl interpreter* written as a C function can be built in any kind of C or C++ user programs, thus basic functions of Tcl can be expanded. We will take the 7.3. version as a basis in this description.

Tk (ToolKit) is an expansion like this, with that we can generate a *Motif* user surface with the help of the built-in commands. Almost every task can be solved with it during a more shorter time, than it would be programmed merely in C. Further, Tk program is more transparent and can be modified more flexibly. We will demonstrate shortly possibilities of the 3.6. version.

Tcl-DP (Distributed Programming) is an expansion, that can be given to Tcl and Tk, which supports a distributed object-oriented programming, remote process calling, and TCP/IP socket communication. Later we will talk about some commands of the 3.2 version.

There are four main advantages of Tcl/Tk:

- A user C or C++ program may contain a Tcl or Tk interpreter together with its built-in commands, we must write only new commands necessary to the application in C language. We have function directories for this, that can be simply embedded and provide many convenient functions.
- A very fast development cycle. Since it is at higher level, than C language, after a short learning we can generate graphic surfaces of full value. Size of code and developing time are minimum the tenth of, if the program would have been made in C, with the help of *Xt/Motif toolkit*. Since the language is interpreted, code can be modified without retranslation and reset, thus trying the new ideas and improvement of failures can be made very quickly. That is why execution is naturally slower, than it would be translated program, but it cannot be noticed at the performance of the today's workstations. If the deceleration cannot be permitted, then the critical parts can easily be written to C.
- Tcl is an ideal language for realizing a communication among programs, because the interpreter can be embedded into every program. Programs may send Tcl scripts to each other, in that there are not only programs, but also variants, and controlling structures. We have only to implement commands in the program, all the other issues are solved by Tcl.
- Because of transparency of language the basic conception can be learnt quickly, and it is enough to learn only the new command at expansion.

1.2. Signs

Commands, that must be written accurately, would be signed with a `different font type`

(Courier, actually). E.g.:

```
      set a hello
⇒     hello
```

A ⇒ sign means the answer in case of a normal recurrence value. A ∅ sign means the wrong recurrence value. E.g.:

```
      set a hello 23
∅     wrong # args: should be „set varName ?newValue?”
```

At description of Tcl commands *italic letters* sign the formal parameters, optional parameters are between question marks, and **...** signs repeteition. E.g.:

```
set varName ?newValue?
unset varName ?varName varName...?
```

1.3.Usage

We can work with Tcl and Tk interpreters in two ways:

- Interactively - commands written from the terminal are executed immediately then, and result is displayed to the monitor. Thus it is easy to try things, to make experiments. Most of these examples listed in this description can be tried on the simplest and on the most spectacular way.
- On the non-interactive way – interpreter executes then a Tcl script stored in a file

tclsh We can start an intepreter interactively with writing a

```
tclsh
```

command in. (Sets necessary for running the program are contained by the appendix in a UNIX environment.) We will get a % prompt in the next line, which means, that the interpreter has received the commands. The `tclsh` (*Tcl Shell*) is actually a *UNIX Shell* (like `sh` or `csh`), so it is applicable for running external programs as well.

wish Interpreter of Tk is the `wish` (*Windowing Shell*). Operation of the `wish` is the same as the operation of the `tclsh`, but it displays a window by starting, which will be the base of the graphic surface.
In a non-interactive way, for example, we can run a tcl script saved to a `foo.tcl` file with a

```
tclsh foo.tcl
```

command. In case of the `wish` we have to apply the

```
wish -f foo.tcl
```

command.
The Tcl script file can be executed even in itself (in a Unix environment), if we have an execution right to the file (it is available with the `chmod a+x foo.tcl` command), and the

first line of the file is the following:

```
#!/usr/loca/bin/tclsh
```

or in case of a `wish`

```
#!/usr/local/bin/wish -f
```

Both programs contain the Tcl-DP expansion.

man There is a full online description about `tclsh, wish` and every Tcl/Tk command, a so-called *manual page*, which can be seen with the help of the a command called

```
man command.
```

The online manual is distributed into chapters, and in the text `tclsh` command of chapter 1. is signed with, for example, *`tclsh(1)`*.
For sake of an easier orientation, commands are highlighted on the left side of the page, and there are some reference cards, as an enclosement.

2. Tcl language

This chapter summarizes basics of the Tcl language, including syntax and a brief description of the commands, illustrated with examples.

2.1 Syntax of the Tcl language

Tcl script consists of one or more new lines, or *commands* separated with semicolons.

Every command consists of one or more *words*, where the first word is the name of the command, and words following the first word are the arguments of the command. Words are separated from each other with spaces or tabulator signs. A command may consist of any number of words. Signs separating words or commands from each other are not parts of any of the words.

Every command can be evaluated in two steps. As a first step, interpreter disintegrates the command into words, and makes the replacements. This step is the same in case of any commands, and it is made by the *parser*. As a second step, it executes command designated by the first word, transmitting all the other words, as an argument.

There are three replacements possible during the analysis:

1.Variant substitution

Substitutiont of variants can be obtained with application of the $ sign. The $ sign can be everywhere within the world, and a valid variant name has to go after it. The effect of it results the substitution of the value of the variant in the word. E.g.:

```
set inches 12
expr $inches*2.54
⇒ 30.48
```

The first command fills word „12” to the variant named inches. By analysing the second command, the „$inches” is replaced with „12”, thus by the execution the `„expr 12*2.54”` command will be executed, obtaining result of `„30.48”.`

In some cases we can use form of `${varName}` for sake of clarity.

2. Command substitution

A word placed between rectangular brackets is evaluated as a command, and its return value will be substituted. The word within the brackets must be a valid Tcl script. E.g.:

```
    set inches 12
    set mm [expr $inches*2.54]
⇒   30.48
```

The second `set` command can only „see” the `„30.48”` word during the execution.

3. Backslash substitution

This substitution can be used for placing special characters – e.g. „$”, „[”, „ ; ”, spaces and new line signs in a word. We can use all the *escape sequences* recorded in *ANSI C* (\n – new line, \t tab, \\ - \etc.) E.g.:

```
    set msg Clock\type\6342\nPrice:\ \$19.99
⇒   Clock type 6342
⇒   Price: $19.99
```

The backslash – new line sequence can be used for paginating long lines. E.g.:

```
    set thisisaverylongvariablename \
        value
```

Paginated line must begin with a space or a tabulator sign, and pagination is considered to be a division.

We have a possibility for asking in another way the parser for *quoting*. It can be obtained in two ways:

1.With quotation marks

A series of character placed between quotation marks is considered to be a word, regardless if it is a space, a tab, a new line, etc. Substitutions of variants, commands, and backslashes are executed int he same way, as in another case. Quotation mark is not a part of the word. E.g.

```
    set msg Clock\type\6342\nPrice:\ \$19.99
```

and

```
    set msg „Clock type 6342
    Price: \$19.99
```

can obtain the same result, as in the previous case.

2. With curly brackets

Parser does not execute any substitutions among curly brackets. (Except the backslash – new line sequence.)

```
      set msg {Clock\type\6342\nPrice:\ \$19.99}
⇒     Clock type 6342\nPrice:\$19.99
```

The most important usage of curly brackets is the delayed evaluation. For example, calculation of value of 5! is the following:

```
set result 1
set i 5
while {$i > 0} {
     set result [expr $result*$i]
set i [expr $i-1]
}
```

In the `while` loop always the actual value of variants will be substituted because of curly brackets.

We can write notifications after the # sign placed to the beginning of the line – more exactly, to a place, where the first character of the command will be placed. A # sign placed to another place does not mean any notifications, parser considers itt o a common character.

We must note two important rules in connection with substitutions:

1. Analysis must be executed in one pass, from left to right. Each and every character is read exactly once.
2. At most only a onefold substitution happens to the individual characters, and the substituted value will not be used during the further substitutions.

The following example illustrates it:

```
set x 1
set a x
set b $$a
```

Value of `b` variant will be „`$x`”, since substitution is executed only once.

One of the consequences of these rules, that in

```
set city „Los Angeles”
set bigCity $city
```

script the second command is executed correctly, because value of the `city` variant is substituted, as a word.

It is sometimes a not required behaviour. For example, the next command can be wrong:

```
exec rm [glob *.o]
```

```
∅   rm: a.o  b.o c.o nonexistent
```

The `glob` command returns filenames fitting to the sample, and the `exec` command tries to execute the `rm` UNIX command in a returned „`a.o bo. co`." filename, which does not exist. For the right operation it would be necessary to disintegrate to value returned by `glob`. This second analysis can be forced by the `eval` command:

```
eval exec rm [glob *.o]
```

2.2. Variants

Simple variants have one name and one value. Variant names and values can be optional character series in Tcl. There are no types of variants, every value is stored as a string. Assignment of variant is dynamic, it can be prepared and deleted any time.

Variants can be prepared, modified and read with the `set varName ?value?` command. If `value` is given, then `varName` variant takes the value of `value`. Return value is the new value of the variant in any case.

We can use associative arrays besides simple variants. An array is the collection of elements, which are variants, too, with their own names and values. E.g.:

```
    set uid (root) 1000
⇒   1000
    set uid(guest) 200
⇒   200
    set uid (root)
⇒   1000
```

It is not necessary to declare arrays in advance, their element number can be changed optionally. Substitution of variants can be used in case of arrays, too:

```
    set login guest
    set a [expr $uid($login)+1]
⇒   201
```

unset Variants, array elements, or whole arrays can be deleted with the help of the unset command:

```
unset login
unset uid (guest)
unset uid
```

incr Value of variants with a whole value can increase with the `incr varname ?increment?` command, with an `increment` value, or, if it is not given, then with one.

append The `append varName value ?value….?` command places values of `value` after value of the variant.

2.3. Expressions

expr The `expr arg ?arg...?` command evaluates expression given by its arguments and the result will be the return value.

Syntax of numeric values is the same as the ones defined in ANSI DC (decimal, octal, hexadecimal, float). Every ANSI C operator and large amount of numeric functions can be used with their usual features, with a surplus, that the relation oprators can also be applied for strings.

Arguments of different types are converted automatically, but an explicit conversion is also possible with `double`, `int` and `round` functions. See in details in the *expr(n)* online description.

2.4. Lists

We call list an arranged group of elements in Tcl. Elements of the list are divided from each other with a space, or a tabulator position:

```
Apple Orange Strawberry Lemon
```

lindex The `lindex list index` returns the element of the given list. Indexing of elements starts from 0.

```
    lindex (Apple Orange Strawberry Lemon)1
⇒   Orange
```

Lists occurring in commands are usually between curly brackets, since they generate a word, but brackets are not parts of the list:

```
    set fruits {Apple Orange Strawberry Lemon}
⇒   Apple Orange Strawberry Lemon
```

Elements of lists can also be lists:

```
    lindex {a b {c de e} f) 2
⇒   c d e
```

There are two commands, with that we can make lists. `concat` and `list.`

concat The `concat list ?list...?` concatenates list given in its arguments to a single list.

```
    concat {a b c} {d e} f {g h i}
⇒   {a b c} {d e} f {g h i}
```

list The `list value ?value...?` considers its arguments to list elements one by one:

```
    list {a b c} d e} f {g h i}
⇒   {a b c} {d e} f {g h i}
```

The `list` command always returns a list with a righ format independently from its arguments, assigning backslashes, or curly brackets, if it is necessary. In case of `concat` it is not guaranteed.

llength The `llength` *`list`* command returns the number of the elements of the list:

```
    llength {{a b c } {d e} f {g h i}}
⇒   4
    llength a
⇒   1
    llength {}
⇒   0
```

linsert The `linsert` *`list index value?value..?`* command returns the list, to that it inserts the given values in front of the element with a given index (ha the index is larger, than the number of elements, or equal with them, then it will place it to the end of the list):

```
    linsert {{a b c } {d e} f {g h i}} 1 A B C
⇒   {a b c} A B C {d e } f {g h i}
```

lreplace The `lreplace` *`list first last ?value value..?`* command returns the list, from it deleted elements with `first` and `last` indexes, and – if there are any – substituted them with `value` parameters:

```
    lreplace {{a b c} {d e} f {g h i}} 2 2
⇒   {a b c} {d e} f {g h i}} 0 1 X {A B} Y
⇒   X {A B} Y f {g h i}
```

lrange The `lrange` *`list first last`* command returns a part of the list from the element with `first` index till the element with `last` index (if `last` index is `end`, the to the end).

```
    lrange {{a b c} {d e} f {g h i}} 1 2
⇒   {d e} f
    lrange { A B C D E F} 2 end
⇒   C D E F
```

lappend The `lappend` *`varName value ?value…?`* command fits the given list values to the variant named `varName`:

```
    set 1 {A B C D E F}

⇒   A B C D E F
    lappend 1 X {Y Z}
⇒   A B C D E F X {Y Z}
    set 1
⇒   A B C D E F X {Y Z}
```

The `lappend`, as well as `append`, is not a definitely necessary command, since it can be built up from another commands, but it is very efficient for long lists.

split The `split` *`string ?splitChars?`* disintegrates the given string, and returns, as a list.

Divinding character(s) can be given in the *splitChars* argument:

```
set f /usr/local/lib/libtcl.a
split $f/

⇒  {} usr local lib libtcl.a
```

join The join list ?joinString? does its inverse:

```
join {{} usr local lib libtcla.} /
⇒  /usr/local/lib/libtcl.a
```

2.5.Control structures

There are two kinds of conditional commands: `if` and `switch.`

if The if `test1 ?then? body1 ?elseif test2 ?then? body2 elseif...? ?else? ?bodyn?` command evaluates the `test1` expression, and if it has a non-zero value, then it executes the `body1` script, and returns its value. Otherwise it evaluates the `test2` expression, and if it has a non-zero value, then it executes the `body2` script, and returns its value, and so on. If none of the test were successful, then it executes the `bodyn` script, and returns its value. E.g.:

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

Expressions used at `if` and the other controlling structures are worth to place between curly brackets, so the evaluation could happen at the right time. Every starting curly bracket must be in the same line, as the previous word. So the following script is wrong:

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

switch The `switch` *?options? string pattern body ?pattern body…?* command fits the `string` to every `pattern`, until it will not find a matching, and then it executes the belonging `body` script, and returns. If the last `pattern` is a default, then it fits to everything. `Options` can be `–exact, -glob, -regexp`, accordingly, if we want an exact, glob-style or regular fitting (the default is the `glob` – see it later). The

```
switch $xs a {incr t1} b {incr t2} c {incr t3}
```

form can also be written like this:

```
switch $xs {
     a {incr t1}
     b {incr t2}
```

```
        c {incr t3}
}
```

If the `body` command is "- " then it executes the command of the next pattern, thus we can assign the same command to several patterns. E.g.:

```
switch $x {
        a –
        b –
        c {incr t1}
        d {incr t2}
        default {incr t3}
}
```

If value of the x variant is "a", "b" or "c", then it increases the value of `t1` variant, and if it is "d", then it increases the value of `t2`, and in case of oher values the value of `t3` variant.

There are three cycle commands. `while, for` and `foreach`.

while The `while test body` command evaluates the *`test`* expression, and if it is a non-zero one, it executes the `body` script, and then it evaluates the expression again. It is repeated again and again, until the evaluation will not result zero, and then it returns with an empty string. The next script copies elements of list `a` to list `b`:

```
set b " "
set i [expr [llength $a] -1]
while {$i>=0} {
        lappend b [lindex b [lindex $a $i]
        incr i -1
}
```

for The `for init test reinit body` command executes the init scipt, and then evaluates the `test` expressiion, and if it is a non-zero one, then executes the `body` script, and then the `reinit` script, and it evaluates again the expression. It is repeated again and again, until the evaluation will not result zero, and then it returns with an empty string. The previous example realized with `for` will be the following:

```
set b " "
for {set i expr [llength $a]-1]} {$i>=0} \
        {incr i -1} {lappend b [lindex $a $i]}
```

foreach The `foreach varName list body` command sets the variant named `varName` to every element of the given list in sequence, and executes the `body` script. It makes easy the process of lists. Realization of the previous example with the help of the `foreach` will be the following:

```
set b " "
foreach i $a{
        set b[linsert $b 0 $i]
}
```

We can also use `break` and `continue` commands, that are usual in C.

break The `break` interrupts execution of the innermost cycle.
continue The `continue` leaps immediately to the next iteration of the innermost cycle.

eval The `eval` *`arg ?arg..?`* can generally be used for generation and execution of command scripts.
One of its applications is the execution of commands stored in variants:

```
set cmd "set x 0"
eval $ cmd
```

Another important application is the second analysis, which we have already seen earlier.

source The `source` *`fileName`* command reads and executes the given script file.

2.6. Processes

There are processes in Tcl, that can be parametered in many ways, and can be defined with a return value.

proc The `proc` *`name arglist body`* command generates a process named as `name`, with a list of parameters, listed in `arglist`, with a `body`. E.g.:

```
proc add {a b} {expr $a+$b}
add 12 4
⇒ 16
add 3
∅ no value given for parameter ”b” to „plus”
```

return Return value of process is the return value of the last command. Using the `return` command we can return immediately with the given value.

global By evaluating the body of the process, the emergent variants are automatically local. We can obtain global variants with the `global name1` *`?name2..?`* command.

There is a possibility for the default of the parameter. If the argument is not given by calling, the process uses the default. If there is a default set to a parameter, then all the following parameters must be like this. E.g.:

```
proc inc {value {increment 1}} {
     expr $value+$increment
}
inc 23 4
⇒ 27
inc 62
```

```
⇒   63
```

If the last element of the argument is `args`, then parameter with alternate numbers can be transmitted by calling. The `args` list will contain the parameters (which can even be an empty list). E.g.:

```
proc sum args {
     set s 0
     foreach I $args {
          incr s $I
     }
     return $s
}
sum 1 2 3 4 5
⇒   15
    sum
⇒   0
```

Within the process variants of the calling process can be available in another way beyond the `global` command. Variants of the optional calling levels are available with `upvar` and `uplevel` commands. See the online description for further information.

2.7 Error handling

If an error occurs during the execution of a command, then running of the program is interrupted, and an error message is displayed. We can get further information about the error in a global variant named `errorInfo.`

catch It would be necessary to handle errors within the program. We can use the `catch` *`command ?varName?`* command, that executes the given command, and places its result (or an error message in case of an error) to the variant named `varName`. Return value of the `catch` command will be the return error code of the command (it will be 0 in case of a successfully executed command).

error We can return from this process with the `error` *`message ?info? ?code?`* command, where the `message` is an error message, the `info` is the value of the `errorInfo` variant, and the `code` is the return error code (its value is usually 1).

2.8. Maintenance of strings

string There are many string operation programs in Tcl. Many of them are realized by the `string` command. This command has numerous subcommands. For example, `string index` and `string range` commands solve the same tasks on strings, as on the `lindex` and `lrange` lists. The `string first string1 string2`, and the `string last string1 string2` commands search for `string1` in `string2` from the left and from the right. In function `of string compare` it corresponds to the strcmp () C function.

format Formatted strings can be prepared with the help of the `format` *`formatString`*

?value value…? command, which can be used on the same way, as ANSI C sprintf () function.

scan We can fit a given string to a given format with the scan *string format varName ?varName varName…?* command, and fill in the variants according to this.

string We can use the string match pattern string command to pattern matching, with that we can match a *glob*-styled pattern. The glob pattern may contain "* " and "? " characters, that can be defined, as usual. Any of the given characters placed between rectangular brackets can be matched to the character, e.g. the "[ch] " pattern to the "c", or to the "h", while the "[a-z] " pattern matches to every small letters. Special definition of the mentioned characters can be resolved with the "\ " sign.

We can match with UNIX regular expressions with the regex and regsub commands. See further details in the online description.

2.9. Maintenance of files and processes

File handling operations used in C can also be found in Tcl.

open A file can be opened with the open *name ?access?* command. Value of the access can be r, r+, w, w+, or a+, it gives the opening method of the file. The open returns with a *file identifier*, with that we can refer to the file during the further operations.

gets We can read an opened file line by line with the help of the gets *fileId ?varName?* command. If the varName parameter is given, then the read string is placed to this variant (without a new line sign), and number of the read characters is returned (it is -1 in case of the end of the file). If the varName is not given, then the string itself is returned.

puts We can write line by line with the help of the puts *?-nonewline? ?fileId? string* command. If there is no file identifier, the it writes to the stdout, and the automatc displaying of the new line character can be put down by giving the –nonewline option.

seek Positioning within a file can be executed with the seek *fileId offset ?origin?* command. Compared to the origin (which can be start, current or end – the default is the start) the next reading will start at the byte given by the offset.

tell, eof Actual position is provided by the tell *fileId* command. In case of the end of the file the command returns with 1.

glob, file We can obtain information from the files of the actual diretory with the help of glob and file commands.

cd, pwd Usage of cd and pwd commands is equal to the ones corresponding to UNIX.

flush, close The output buffer can be cleared out with the flush fileId command. We must close the opened files with the close *fileId* command (it will result also a flush operation).

exec UNIX processes can be started from Tcl program, and there is also a possibility for using *pipelines*. They are processes, that can be started with the exec command, where it is possible to make standard 1/O redirections (<, <<, >, |) and background runnings (&). For opening a pipeline we can

use the `open` command, and the filename must begin with the | sign in cases like this.

pid We can query the *UNIX process identifier* of the actual process or the opened pipeline with the `pid` command. UNIX environmental variants can be found in the `env` built-in array.

exit The `exit` command can be used for exiting from the process with giving the exit status.

2.10. Other Tcl commands

Tcl interpreter provides a possibility for the query and modification of its own inner state.

array, info The `array` command returns size of the associative arrays, identifiers of their elements, etc. We can query names of the existing global and local variants, names of processes, their parametering, bodies, names of commands, version number of the interpreter with the help of the `info` command. The `info` and `array` commands contain several subcommands – see also the online description.

trace Usage of Tcl variants can also be traced from the program, with the help of the `trace` command. We can assign a process to every variants, which is called, when the given event happens. This event could be reading, modification or elimination. See also *trace(n)*.

rename Commands can be renamed optionally, or deleted with the help of the `rename` command.

unknown Usage of the `unknown` command is a special possibility. This command is executed, when the interpreter cannot find a command. We can modify the original command with the definition of a new `unknown` process. The following example allows the abbreviated usage of the commands, until the unambiguity allows it.

```
proc unknown {name args} {
     set cmds [info commands $name*]
     if {[llength $cmds] ! = 1} {
          error ”unknown command \”$name\” ”
     }
     uplevel $cmds $args
}
```

6. Appendix

In order to run `tclsh` and `wish`, or to reach on-line manual the following environmental variables must be set:

```
PATH /usr/local/bin:$PATH
MANPATH /usr/local/man
TCL_LIBRARY /usr/local/lib/tcl
TK_LIBRARY /usr/local/lib/tk
```

The environmental variables can set in `csh` and `tcsh` shells with

```
setenv variable value
```

command, in `sh`, `bash`, `ksh`, `zsh` shells with

```
variable=value; export variable
```

command.

In order to include Tcl interpreter in C program codes we have `libtcl.a` function library in the `/usr/local/lib` directory, which contains the whole interpreter, and `tcl.h` header file in the `/usr/local/includes` directory, which contains the necessary definitions.
Instead of futher comments let we look the following C program code which execute a Tcl program given in the command line.
(As a matter of curiosity we should remark, that even the `tclsh` and the `wish` programs are not much longer than this.)

```
#include <stdio.h>
#include <tcl.h>
main(int argc, char *argv[]) {
     Tcl_Interp *interp;
     int code;
     if (argc != 2) {
          fprintf(stderr, "Wrong # arguments: ");
          fprintf(stderr, "should be \"%s fileName\"\n",
                         argv[0]);
          exit(1);
     }
     interp = Tcl_CreateInterp();
     code = Tcl_EvalFile(interp, argv[1]);
     if (*interp->result != 0) {
          printf("%s\n", interp->result);
     }
     if (code != TCL_OK) {
          exit(1);
     }
     exit(0);
}
```