Useful Tcl and Tcllib functions

HIDDEN TREASURES

The Tcl/Tk distribution contains a lot of useful functions that many programmers don't know about. These can help to solve many common problems, such as parsing the command line, very quickly. Carsten Zerbst takes a closer look

When writing applications you often encounter problems that countless developers have had before you. Besides the normal system functions, Tcl provides solutions for many problems, so that programmers don't need to go to the trouble of sorting them out for themselves. These solutions come in the form of packages, which are easily loaded into the interpreter and are usually part of a normal system installation. An earlier instalment of Tcl/Tk has already introduced the msgcat internationalisation package, this time we are going to look at three more packages.

Split

Don't you just love the thousands of options on the command line that control an application? Before the program can make use of these parameters there is an arduous task to be performed. The string entered by the user must be divided up and examined for valid and invalid options. You could, of course, write this yourself, but why bother when a solution already exists in the shape of the opt package?

This package contains the command *tcl::OptProc*, which is used instead of the normal *proc* command. The command has three parameters: *tcl::OptProc* name, parameter description and body. The name of the procedure to be created is followed by the description of its parameters. This description is actually a list of the valid passing parameters. Each individual parameter is in turn described in another list that contains the following elements:

- Parameter name, with a hyphen for optional arguments.
- Type. OptProc recognises *-boolean*, *-int*, *-real*, *-list* and *-choice*, the latter including the choices available.
- Default value. Used if this parameter has not been set when the procedure was called.
- Description.

It is not necessary for all four elements to be defined. For example, OptProc can automatically determine the type from the default value. *Listing 1* shows some different examples of definitions as well as nested lists in the second parameter of *tcl::OptProc*. This is followed by the actual function.

When calling a command defined in this way the parameters are passed as individual strings, which the procedure automatically parses according to the definition. This is also the reason for the *eval* construction in line 23: *$argv* contains the command line parameters in form of a list, but *main* is expecting them as individual options.

The parameters to be passed are available as variables within the procedure. Another interesting feature of the example is the *foreach* loop: at each iteration of the loop the variable *v* receives the name of a variable containing the value of the respective option. *$v* in line 19 therefore only outputs the name of a variable, *[set $v]* is required to show its content.

If the parser encounters an error, either because the type of a variable is incorrect or because a

Listing 1: The *opt* package

```
01 #!/bin/sh
02 # Example for the opt package
03 # \
04 exec tclsh $0 $@
05
06 package require opt
07
08 tcl::OptProc main {
09     {require -string "file name"}
10     {-flag}
11     {-int 2 }
12     {-real 1.0 "flag, default 1.0"}
13     {-bool -boolean false "boolflag, default false"}
14     {-choice -choice {1 2 3} "selection, 1, 2 or 3"}
15     {-list -list {} "list, default {}"}
16     {?more? -string "" "unparsed remainder"}
17 } {
18     foreach v [list required flag int real bool choice list more] {
19         puts stdout [format "%14s : %s" $v [set $v]]
20   }
21 }
22
23 if {[catch {eval main $argv} err]} {
24     puts stderr $err
25     exit
26 }
```

Latest news

No matter how insecure domain management by email at InterNIC may be, even without security gaps things can go badly amiss with name server entries. In this particular case the cause is something that is often red, always small and has two cute little round eyes. Others see it as an expensive toy with which its owners are trying to recapture their lost youth. We are, of course, talking about the new Mini.

What exactly is the connection between BMW and the Internet? At first glance nothing, apart from the fact that BMW also uses Tcl. Since 1996 an enormous font of Tcl knowledge has been available at the "Tcl'ers Wiki". Its URL *http://www.mini.net* recently started to rather unexpectedly link to BMW. The entry at *registers.com* had been transferred to BMW without anyone bothering to consult the domain's owner, Jean-Claude Wippler. The company had bought a number of other addresses featuring the Mini, but not mini.net. Wiki users were quick to notice the error, but it took some time before the Wiki was back in business. Accidents like this just go to show once again how easy it is to cause major disruptions on the Internet.

Combat: CORBA scripting with Tcl

CORBA is the solid foundation of many an application. Frank Pilhofer's Combat has probably been the best Tcl binding available for this middleware for quite some time. It allows you to write CORBA clients as well as servers. Until now it required MICO as its basic ORB – but with Combat 0.7 there now exists a pure Tcl implementation so that complicated libraries are no longer required.

Patchlevel Tcl 8.3.4

The latest patchlevel for Tcl 8.3 is now available in Tcl 8.3.4. The improvements primarily concern 64-bit platforms and are therefore (not yet) of much general interest. While Tcl can generally do more with each new release, naturally growing ever bigger at the same time, CISCO is currently financing a project by ActiveState to develop a modular Tcl. The aim is to only load those modules into the interpreter at startup that are absolutely necessary. What can be gained by this is demonstrated by NASA's Marsrover or currently by the game Wiggles in which each figure runs on a pared-down interpreter which takes up all of 17Kb.

New Tcl database

A close symbiosis has existed between databases and Tcl for quite a while. There is hardly an SQL database that doesn't come with a Tcl extension, if Tcl isn't used for system tools or code tests anyway, as in the case of Adabas or Oracle. Sometimes you need just a little bit more than a simple text file without really requiring a full-blown database. For these occasions Richard Hipp offers SQLite, a small database engine that runs directly in the application and understands a sufficient subset of SQL. This means that small applications for interactive customer catalogues or CD collections can be implemented without elaborate server processes.

Even without security gaps things can go badly amiss with name server entries

required parameter is missing, it returns an error. This can return the section of the program called to the user (*catch* in line 23 and *puts* in line 24). Another popular option is *-help*, which outputs the definition of the arguments with their description. In Figure 1 you can see the script from Listing 1 in action.

The OptProc package allows you to provide Tcl programs with a useful command line interface very quickly. Even individual procedures can benefit from this flexibility.

Let's have it

Of course Tcl has much more to offer, including an implementation of HTTP, the Hyper Text Transfer Protocol. The HTTP package is part of a standard Tcl installation and allows access to Web pages. At the heart of the package is the command *http::geturl*, it can load files with all the trimmings, fill in forms or simply retrieve information about a page.

The return value of the command is a token. This token is the name of an array, which in turn contains information about the page and, depending on the request, possibly even the file itself. After each use of *http::geturl* it is therefore necessary to delete this array with the command *http::cleanup*, otherwise this is a great way for the interpreter to become bloated with more and more data.

First of all it makes sense to have a look at the environment of a Web page. The flag *-validate* restricts *http::geturl* to only loading information like size and MIME type of the file or Web server type instead of the entire file. This is what happens in the first few lines of Listing 2, while line 14 outputs this meta-information. However, not all Web servers reveal their meta-data without pages being actually requested. The CUPS server, for instance, is very unforthcoming in this respect and only supplies meta-data with a file.

Listing 2: Files from the WWW

```
01 #!/bin/sh
02 # Example for the http package
03 # \
04 exec tclsh $0 $@
05
06 package require http
07
08 set url http://tcl.activestate.com
09 #set url http://127.0.0.1:631
10
11 # meta-information
12 set token [http::geturl $url -validate 1 ]
13 foreach {name value} [set $token\(meta)] {
14     puts stderr [format "%-20s = %-20s" $name $value]
15 }
16
17 # file
18 set fd [open as.html w]
19 puts stderr "get $url"
20
21 proc progress {handle max size } {
22     puts -nonewline stderr [format " %.0f%% " [expr 100.0*$size/$max]]
23 }
24
25 set token [http::geturl $url \
26     -channel $fd \
27     -blocksize 2048 \
28     -progress progress
29 ]
30
31 puts stderr "finished"
32 puts stderr [http::code $token]
33 http::cleanup $token
34 close $fd
35 exit
```

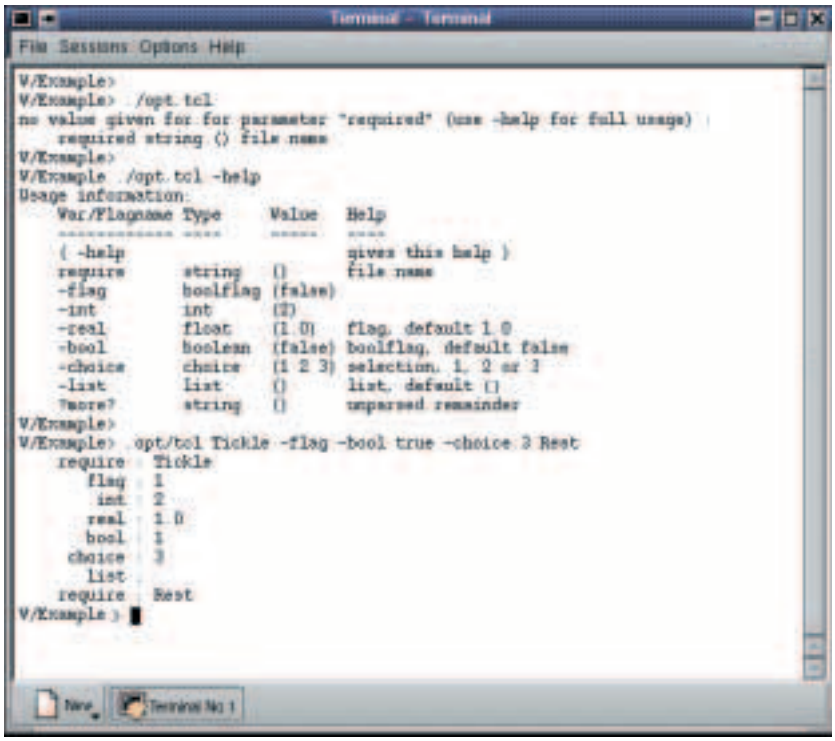

Figure 1: Entry using the *opt*package

Once you have the information about a page you might want the whole thing. The next lines of Listing 2 contain a simple example. As described above, *geturl* normally returns a token that describes an array in which the file itself eventually ends up. In Listing 2 the file is instead written directly into an open file due to the option *-channel*.

The command *http::geturl* blocks the interpreter until it is finished or an error occurs. So that the user will get some sort of feedback during loading, *geturl* invokes the callback procedure *progress* every 2048 bytes. *progress* simply outputs the percentage of the file that has already been loaded.

As soon as the file has been transferred completely the information held in the token becomes available. Various functions from the HTTP package access these data, *http::code*, for example, requests the transfer status code. The output of the script in Listing 2 can be seen in Figure 2.

The Web has much more to offer than simple data transfer from server to client. The *form* tag allows you to design simple user interfaces in HTML with input fields, radio buttons and the like. A popular service requiring user input is AltaVista's Babelfish, which can translate single words, sentences or entire HTML pages between various Western and Eastern languages.

The values of the buttons and entry fields must be suitably packaged for transfer to the server. This is done using the command *http::formatQuery*, which expects a list of variable names and values as input. The variable names can be found in the HTML code as attributes of the tags *input*, *select* or *textarea*. For radio buttons and the *select* tag the code also contains the valid parameters. The request's target URL is contained in the *form* tag as the attribute *action*; the formatted request will need to be appended to this.

A simple example can be seen in Listing 3 where a single word is translated using Babelfish. The opt package is used to parse the command line. Line 14 assembles the required URL for translation of the word in the desired language combination. Unlike

Info

Tcl'ers Wiki	*http://www.mini.net*
Getleft	*http://personal1.iddeo.es/andresgarci/getleft/english/*
GNOCL	*http://www.dr-baum.net/gnocl*
CORBA	*http://www.omg.org*
COMBAT	*http://www.fpx.de/Combat*
Wiggles	*http://www.wiggles.de*
SQLite	*http://www.hwaci.com/sw/sqlite/*
Wrong registration	*http://www.mini.net/tcl/2355.html*

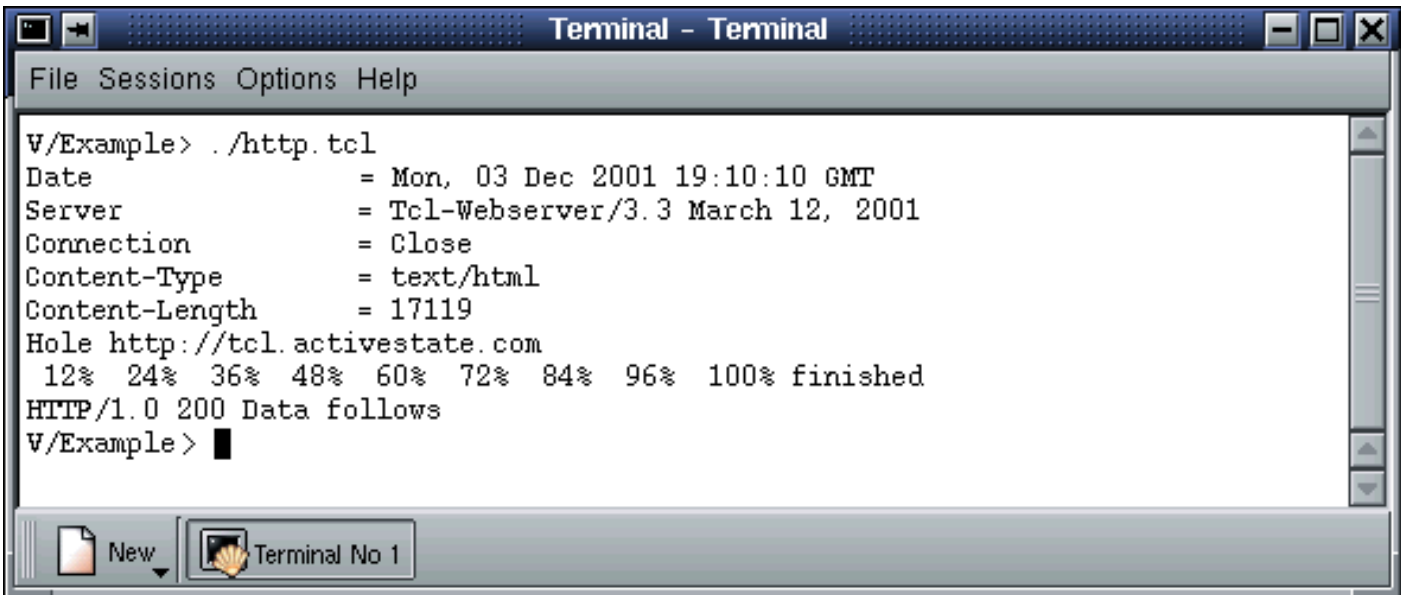

Figure 2: Trawling through the WWW with Tcl: the script first shows information about http://tcl.activestate.com, then it loads the page and gives a running update of its progress

most of our other examples this one also contains error handling, otherwise it wouldn't be much use. Babelfish is often busy, so our program cuts its losses after 15 seconds. A check in line 19 whether the transfer was successful is followed by the code for processing the received data.

Worldwide

Internally Tcl works with Unicode and is therefore able to handle Chinese or Arabic characters. For strings that either originate externally or are intended for external use, Tcl assumes Western European ISO-8859-1-encoding. However, AltaVista pages are created in UTF-8 so the text needs to be converted to the internal Unicode format before processing. This is done using the *encoding* command in line 23.

Next, we need to extract the translated word from the page. The most elegant way of doing this is with the help of the W3C's DOM model, which we are going to have a closer look at in a future instalment. Until then we are going to extract the old-fashioned way: the HTML code including the result is split into individual lines. The translated word is located between the start and end tag of *textarea*, the script simply combines any relevant lines and discards any unwanted tags with *regsub*. Not pretty, but effective.

The author

Carsten Zerbst works for Atlantec on a specialised PDM-System for the ship-building industry. Apart from that he devotes his time to the general application of Tcl/Tk.

Listing 3: Client for interactive HTML pages

```
01 #!/bin/sh
02 #
03 # Translation using Altavista's Babelfish \
04 exec tclsh $0 $@
05
06 package require opt
07 package require http
08
09 tcl::OptProc main {
10     {text -string "text"}
11     {-langs -choice {en_de en_fr en_it fr_en fr_de de_en de_fr it_en}
"languages, default en_de"}
12 } {
13     set url  http://world.altavista.com/tr
14     append url "?[http::formatQuery tt urltext urltext "$text" lp
$langs]"
15
16     if {[catch {http::geturl $url -timeout 30000} token]} {
17         error "Problem with network: $token"
18     }
19     if {[http::ncode $token] != 200} {
20         error "Problem with server, $token"
21     }
22     # "Brutal" data extraction method
23     set htmllist [split [encoding convertfrom UTF-8 [http::data
$token]] \n]
24     http::cleanup $token
25     set index0 [lsearch -regexp $htmllist "<textarea"]
26     set index1 [lsearch $htmllist "</textarea>"]
27     if {($index0 < 0) ||($index1< 0)} {
28         error "Problems with parsing"
29     }
30     set result [join [lrange $htmllist $index0 [expr $index1 -1 ]]]
31     regsub {<textarea[^>*]>} $result "" result
32     puts stdout $result
33     exit
34 }
35
36 if {[catch {eval main $argv} err]} {
37     puts stderr $err
38     exit
39 }
```