A practical explanation of the Vim scripting language

Powerful Solution

Scripting facilities are not only available for Emacs; Vim, the modern extension of the venerable *vi* editor, includes its own scripting language. It may not offer quite as many features as Emacs' Lisp, but then again, it does not need as many brackets either. We will be investigating its potential based on an editor plug-in for word for word translations. BY SUSANNE SCHMIDT

Unilever

The Vim [1] editor is the modern variant of the heritage editor *vi*. With its GUI interface, syntax highlighting for a wide range of programming languages and multiple undo facilities, Vim has a lot more to offer than the original. Many users are aware that Perl, Python, Ruby, or even Tcl can be used to call various Vim functions. However, the editor also had its own embedded language that provides while loops, if-else constructs, variables, predefined functions, and a handful of standard arithmetical operators. Although one would hesitate to call Vim fully-featured, it does offer enough for most tasks.

One major advantage that the Vim scripting language has to offer is the fact that the complete collection of keyboard shortcuts and ex-commands (that is, colon commands) are available for scripting, and the GUI version of Vim even adds menu items. The scripting language can be used to author tools that enhance Vim's syntax highlighting facilities or change the editor's approach depending on the current file type. The language is developing quite quickly; to date new functions have become available in each new release. This can also prove to be a pitfall as some functions have been renamed in each Vim version; thus, backward compatibility can be an issue.

The Vim helpfile *usr_41.txt* provides a list of functions; you can type *:help usr_41* to access the list. The functions in our sample script are based on Vim 6.1. If you have an older version of Vim, you may need to refer to the helpfile before performing a few modifications.

Quick Access Translations

Scripting languages are best explained by reference to an example. The short script we will be looking at contains all the vital elements of the Vim language and actually does something useful: It parses the English word highlighted by the cursor, performs a lookup in a translation dictionary and displays the translation for the word. The user can then choose to insert the word or store in an external file.

```
@L:It makes sense to use the <I>map<I> command to define
ccess to the required translation function in <I>dict.vi
ly when launched. However, Vim maps most keys by default
ned their own mappings. So, how do you locate an unused
t keyboard assignments, and <I>:help map-which-key<I> pr
<I> provides a list of keyboard shortcuts for commands.
le to prevent inadvertent re-mapping of important keys w
 a user-definable keyboard assignment protection mechani
eing overwritten. This is indispensable for plug-ins tha
provides more details. Our sample script <I>dict.vim<I>
cally unused.

@L:To assign the word currently highlighted by the curso
unctions <I>Eng2Ger()<I> and <I>Ger2Eng()<I> need to be
 function to use the highlighted word and the function f
ut (start of Listing 1).

:call Eng2Ger(expand("<cword>"))
Sinn
(I)nsert, (S)ave to File, [C]ancel:
```

Figure 1: The Vim menu for translations in the basic console version

Conversely, you can enter a word in a dialog box and display the English equivalent. Doing so requires interaction with the user, access to external files and commands, search pattern recognition functionality and conditional execution of an option selected by the user. Access to the script is provided by a special keyboard mapping. Some user dialog functions work with both the GUI and console versions – Vim will automatically display the correct variant.

Scripts as Plug-ins

There are several ways to enable your own script in Vim. You can insert short

code segments in *.vimrc*, or create a file such as *myscript.vim* in the Vim plug-in directory. When launched, Vim will automatically load any scripts residing in the script path as if they had been added to *.vimrc*. The global path for Vim plug-ins typically points to */usr/ share/vim/ vimVersion/plugin* (or */usr/local/share*). Plug-ins will then be available for any system users.

You should store your own plug-ins in your home directory below *~/.vim/ plugin*. You can use environment variables or the global Vim configuration file to change the path while compiling. The *FileType* configuration option allows you to load a new plug-in based on the file type. Our sample vimrc leverages this feature for gzipped files or C source code. A similar technique would allow you to open an English or French dictionary by parsing the names *file.gb* or *file.fr*, in fact you could even load a Chinese language plug-in for a Unicode file.

From Keyboard to Script

It makes sense to use the *map* command to define a keyboard shortcut that provides instant access to the required translation function in *dict.vim*. Vim will load this mapping automatically when launched. However, Vim maps most keys by default, and many users will have additionally defined their own mappings.

So, how do you locate an unused key? The *:map* command shows the current keyboard assignments, and *:help map-which-key* provides a few additional tips. *:help index* provides a list of keyboard shortcuts for commands.

Vim also provides the *mapleader* variable to prevent inadvertent re-mapping of important keys with your own assignments. *mapleader* is a user-definable keyboard assignment protection mechanism that prevents existing assignments from being overwritten. This is indispensable for plug-ins that you intend publish. *:help mapleader* provides more details. Our sample script *dict.vim* uses the [F9] and [F8] keys as they are typically unused.

To assign the word currently highlighted by the cursor to a function when a key is pressed, the functions *Eng2Ger()* and *Ger2Eng()* need to be mapped to a key. We want the *Eng2Ger()* function to use the highlighted word and the function for the opposite direction to accept user input (start of Listing 1).

The most important aspect here is the call to *expand* with a special parameter. When [F9] is pressed, the *:call* command is performed for the *MyDict* function and a special Vim parameter called *<cword>*. *<cword>* designates the word at the current cursor position. Vim resolves special variables of this type, including environment variables, by calling *expand()*. The *Eng2Ger()* function, which we wrote ourselves, contains *<cword>* as a parameter (Listing 1, lines 7 through 15). Functions and loops are not enclosed in brackets but use the *while – endwhile* or *function – endfunction* directives.

Listing 1: Vim script *dict.vim*

```
01 map <F9> :call
   Eng2Ger(expand("<cword>"))<CR>
02 map <F8> :call Ger2Eng()<CR>
03
04 let dictpath = '~/stuff/
   eng2ger.vok'
05 let vocpath  = '~/stuff/
   vimvoc.txt'
06
07 function Ger2Eng()
08     let word = inputdialog
  ("Translate de -> eng: ")
09     call Extract(word,
   "want_eng")
10 endfunction
11
12 function Eng2Ger(word)
13     let word = a:word
14     call Extract(word,
   "want_ger")
15 endfunction
16
17 function Extract(word, lang)
18     let word = a:word
19     let lang = a:lang
20     let wordpair = system
   ('grep "^' . word . '\b \-\|\-
   \b'. word . '$" ' . g:
   dictpath)
21     if(v:shell_error)
22         echohl ErrorMsg
23         echo "Error occurred!"
24         echohl None
25     else
26         let match = matchstr
   (wordpair, "[^\n]*")
27         call Translate(match,
   word, lang)
28     endif
29 endfunction
30
31 function Translate(match,
   word, lang)
32     let match = a:match
33     let word = a:word
34     let lang = a:lang
35     if(lang == "want_eng")
36         let result =
   substitute(match, "\\
   (" . word . "\\) -- ",
   submatch(1), "")
37         call ConfirmDialog
   (result, match)
38     elseif(lang == "want_ger")
39         let result =
   substitute(match,
   " -- \\(" .  word .  "\\)",
   submatch(1), "")
40         call ConfirmDialog
   (result, match)
41     endif
42 endfunction
43
44 function ConfirmDialog(result,
   match)
45     let result = a:result
46     let match = a:match
47     let choice = confirm
   (result,"&Insert\n&Save to
   File\n&Cancel",3, "Question")
48     if(choice == 1)
49         execute "normal ea " .
   result
50     elseif(choice == 2)
51         execute "e " . g:
   vocpath
52         execute "normal o" .
   match
53         execute "w" . g:
   vocpath . "|bd"
54     elseif(choice == 3)
55         execute "normal e"
56     endif
57 endfunction
```

The keyword *let* assigns values to variables, paths for the script in our case. The Vim *a:word* reference provides access to the variable arguments for a function. *a* refers to an argument, *s:* a local variable of the called script, *g:* a global variable that is referenced within a local block; *b:* is local to a buffer, *w:* local to a window, and finally, *v:* represents a Vim-specific variable. Window not only refers to GUI windows, but to Vim console elements.

The Vim *input()* and *inputdialog()* functions allow Vim to accept user input. The dialog text is passed to the function as an argument, and the function returns the user input. The appropriate terms are stored in a local variable of the function where the script can access them. The *Extract()* function (see Listing 1, lines 17 through 29) then locates the required term in the external file (*dictpath*) and returns a pair of terms from the file. The term pair is then passed to the *Translate()* function (see Listing 1, lines 31 through 42) along with the target language indicator.

Performing Lookups

There are several ways to look up the word at the current cursor position in a dictionary file: the *system()* function allows you to call arbitrary system commands, such as *grep*. An extremely basic English dictionary file is already available for lookups (Listing 1, line 20). Alternatively, you might like to use a *system* call to access a dictionary, such as *dict.leo.org* or a *dictd* server (dictd is a daemon that provides various dictionary files on the network). The advantage of the grep version is that it does not necessitate network access.

Just like Perl, the grep call needs to use . to create a string and resolve the variable. The system call will pass *grep TheWord ~/stuff/eng2ger.vok* to the shell. The return value for grep, that is any lines that match the regular expression, ends up in the *wordpair* variable. The terminology file provides term pairs in *Wort – word* format. If grep does not find a match, the script parses the Vim variable *v:shell_error*. It adds the standard highlighting for *ErrorMsg* to the error message and removes the highlighting again after outputting the error (see Listing 1, lines 20 through 25).

As grep can return multiple lines including newlines, whereas Vim tends to store the whole return value as a long string in a variable, *matchstr()* is used to extract the relevant response (Listing 1, line 26). If you use a different dictionary file, you might need to look into sanitizing the return value. In other words, vocabulary extraction will always depend on the format of your dictionary file. This issue may well be resolved within the next couple of years, by using an XML format that can handle natural languages and inflexions, thus recognizing that "fell" comes from "fall" [3], for example.

Translating

The sanitized word pair and the required language are then passed to the *Translate()* function. Translate calls *substitute()* to remove the unwanted part of the word pair and keep the target word. The target language is then passed to the *ConfirmDialog()* function along with the word pair. This function displays the translation and takes care of saving or inserting the translation.

Vim basically uses a kind of Perl regex dialect, or at least reflects many of its capabilities: lookahead and lookbehind, various quantifiers, and lots more besides.

The syntax for some elements may be different from what you are used to with Perl, and calling them via the *substitute()* function is also different from the command-line. The *submatch (number)* parameter corresponds to *\1* in the regex in the Vim command-line. The unwanted portion of the word pair is simply removed (see Listing 1, lines 31 through 42).

Display, Insert, or Store?

Vim provides functions for user interaction such as *input()* or *inputdialog()*. The *Confirm()* function displays a dialog box with various buttons in the GUI interface and uses keyboard shortcuts if you are not using the GUI. The second variable *vocpath* is used to point to where a terminology file needs to be created when inserting a word pair (Listing 1, lines 44 through 57). The Confirm dialog box is easy to use:

```
let choice = confirm(translation, "&Insert\n&Save to File\n&Cancel",1, "Question").
```

The first parameter is the text to display – this is the translation for the word in *dict.vim*. The next parameter is a list of options. The letter following the ampersand *&* is displayed in the console menu as *(C)ancel* with a shortcut; the GUI displays a button labeled *Cancel*. The third argument is a default. *confirm* simply returns a number that depends on the number of options in the second parameter. The default value is the number to select if the user simply pressers [Enter] – this is option *3*, *Cancel*, in *dict.vim*.

The last parameter is important for the GUI: Vim will display the appropriate icon for the parameter *Generic*, *Question*, *Error*, *Info* or *Warning*. The return values *1*, *2*, and *3* are evaluated in the If-Elseif-Endif loop (Listing 1, lines 48 through 56).

Simple Insertion

It is extremely easy to insert a word in a vocabulary file. In Vim, we use the *execute()* command in a script to call commands for normal mode, this is the mode available when [Esc] is pressed. We now want Vim to execute *e* for "End of Word" and then *a* for "Append", and insert the translated word directly after the keyword. Finished. Option *3* allows you to quit the translation menu without performing any task. The script simply places the cursor at the end of the word, but you could just as easily jump to the end of a line or perform some other task.

The second option is more complicated and adds both the source word and the translation to a terminology list

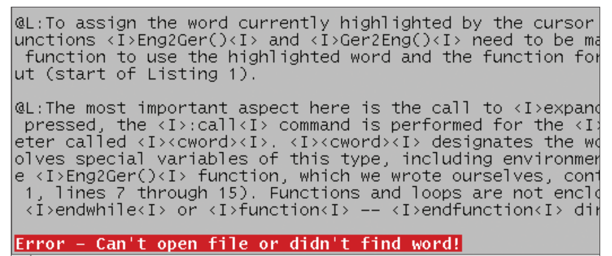

Figure 2: If something goes wrong, an error message is displayed in the current color scheme

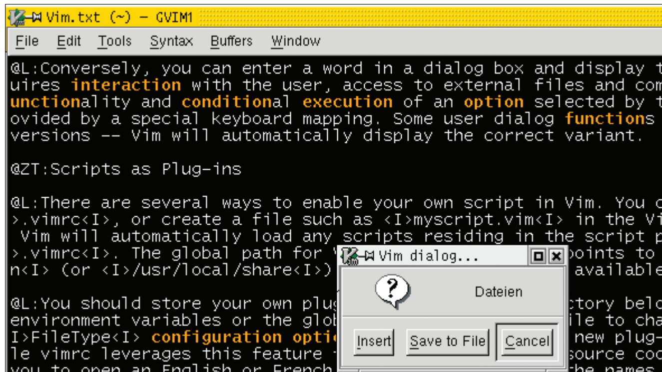

Figure 3: Now choose what you want to do

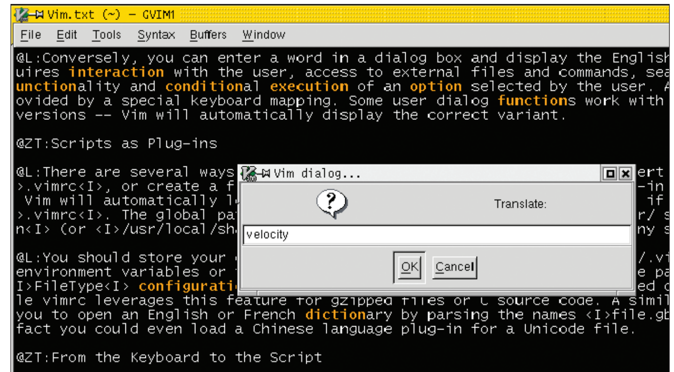

Figure 4: In the GUI version a window appears, accepting input for translation

that might come in useful if you need to revise some vocabulary. The dialog box (see Listing 1, line 53) also takes care of saving and inserting the translation. *execute* in combination with *normal* only simulate the user typing to or navigating a file. This allows for arbitrary sequences of Vim commands – just like in the modes enabled by pressing [Esc] or *:*.

Revising Vocabulary

A call to *:e file* opens a new file to export a pair of terms to it. *execute* simulates the *:* command in the Vim script. In contrast to the normal command, you do not need a keyword like *colon* or something similar. Instead just pass the *execute* command directly. The new file will be the current buffer for a short time. The [O] command inserts the new pair of terms in front of any existing content.

To change the order – that is insert the current terms at the end of the file – simply use [Shift] + [O]. In order to be able to apply the same procedures to the vocabulary file as to the dictionary file at a later date, *dict.vim* uses the same syntax to write the word pair. After completing this step, you can press [W] to store the current buffer, that is the vocabulary file, and after saving, call *|bd* (buffer delete) to flush the buffer. The pipe character | in Vim's *:* mode means: only move on to the next command, if the previous command completed ok (Listing 1, line 53).

Additionally, unknown terms can be formatted and stored in an external file to create vocabulary cards at a later date.

Plugins & Icons

For those heavily addicted to mouse and nifty little icons, a vim-plugin could be added to the icon-bar of gvim. Create a little icon with the exact size of 20x20 pixels, save it as *mybutton.bmp* (a bitmap file).

Create in your *~/.vim* directory a subdirectory named "bitmaps". Copy the new bitmap into this directoy. Now a plugin or a vim command could be "mapped" to this new icon. For experimental use, try this:

```
menu ToolBar.mybutton :call Eng2Ger(expand("<cword>"))<CR>
```

This command works as follows: The command *:menu* adds a new icon in the icon bar, according to the parameter *ToolBar.mybutton*. *Mybutton.bmp* has to be found in the bitmaps-directory of vim, if not, an empty icon is shown. With the menu-commands, mapping a function call is quite easy. In addition to mapping the translate function to a key, it's mapped to an icon. The new *mybutton* entry in the icon bar raises immediately after activating the *:menu* command.

Clicking the new icon OR pressing F9 acts in the same way: The translation function *Eng2Ger* which translates the word under the cursor, is called. For the vice-versa direction work with the same command, but map a different icon to the plugin-function *Ger2Eng*. Of course, the menu mapping command call could be added into the *.vimrc* or better to the *.gvimrc* in the home directory.

More Featurezzzz!

If you require more than just simple translation, you can modify the script correspondingly – more options are available if you use the Vim interface for Perl, Python, Ruby, or Tcl, such as looking up Unicode encoded terms, and thus compiling far more exotic dictionaries. Many publishing companies offer digital versions of their dictionaries; if you figure out the format you could even design an interface to the "Micro Robert" (French dictionary).

Ispell allows for flexible search patterns, as it understands the kind of inflected forms that occur in German or French. Thus, a plug-in would recognize "lis", "lu" or "lirai" as different forms of the French verb "lire" (to read). Building on Ispell's capability to recognize the stem and ending of a word, programmers are only a few steps away from elegant syntax highlighting for typos, grammatical or even stylistic issues.

This would allow Vim to check a ruleset for magazine articles and indicate any misdemeanors. Shortly after those Star Trek style pads hit the stores, you can just point at a word to boldly display the translation. ■

INFO

[1] Vim homepage: *http://vim.sf.net*

[2] English/German dictionary file: *http://ftp.leo.org/download/pub/comp/doc/dict/*

[3] Natural Language Processing with XML: *www.w3c.org/TR/2000/WD-nl-spec-20001120/*

THE AUTHOR

Susanne Schmidt is a political scientist, and has been using Linux since 1994. Open Source has helped her pay the rent so far thanks to various publications and freelancing for various Linux enterprises in Berlin, Germany.