The Answer Girl ALL IN THE TRANSLATION

PATRICIA JUNG

The world of everyday computing, even under Linux, is often good for surprises. Time and again things don't work, or not as they are supposed to. The Linux Magazine's Answer Girl demonstrates how to deal elegantly with such little problems.

__Read rights__: For the content of a file to be made accessible with the aid of a pager such as less or an editor to the eyes of a user, it must carry, from the point of view of this user, the r- ("read"-) flag. This can be set with the command chmod for the owner of the file (chmod u+r filename), the owner group (g+r) and all others (o+r). In the case of directories the read right allows the content of the directory to be displayed with ls. Other rights include write (w) and execution rights (x). These can be shown using ls -l ("long listing").

Little graphical helpers such as Qtrans, (presented in Linux Magazine Issue 8 May 2001 on p96) also offer offline help, but sadly the dictionary formats used there make no allowance for simple browsing with *less* & co. on the command line. For the DICT-protocol there is certainly also the command line tool *dict*, although DICT, despite its open format, does have one major drawback: without the *dictd* server nothing whatsoever will happen.

All in all, not exactly ideal for users who like to use their dictionaries for browsing, or would like to continue to use vocabulary lists created by the sweat of their own brow. Pure ASCII files are unbeaten, so long as we stick to the Latin alphabet: Browsed through with *less*, you can use the *less* command */search term* to look specifically for certain *search terms*.

Wanted: ASCII glossaries

You can find collections of ASCII dictionaries on the Internet. Anyone hunting for collected English-German glossaries, for instance, will strike lucky at *http://www.wh9.tu-dresden.de/~heinrich/dict/dict_leo_ftp/leo_ftp/*.

Once downloaded and copied into a joint directory (anyone having *root* rights will create */usr/dict/eng_deu*) the browsing can commence (assuming you have **read rights**):

```
[trish@lillegroenn ~]$ cd /usr/dict/eng_deu
[trish@lillegroenn eng_deu]$ less *
```

When it comes to the letter *z*, *less* asks for leave to speak in the last line:

```
(END) - Next: EXERCISE.VOC
```

How on earth can we get to the next file *EXERCISE.VOC*? Pressing *h* brings up a Help page, from which we can read:

```
      CHANGING FILES
[...]
:n   *  Examine the (N-th) next file from the➥
 command line.
:p   *  Examine the (N-th) previous file from➥
 the command line.
```

The *less*-command *:n* thus brings us to the *n*ext file, while with *:p* we can jump back a file at a time. Unfortunately, forward searches are always limited with */searchterm* and backwards searches with

?searchterm to the currently displayed file. But here again the *h* (or the man page) comes to our assistance:

```
       SEARCHING
[...]
       Search patterns may be modified by one➘
 or more of:
[...]
       ^E or *  Search multiple files (pass th➘
ru END OF FILE).
```

To try it out we close the Help mode with *q*, go back, using *:x* to the first file and once in there, with *1G* ("Goto line *1*") to the first line. If we now enter */*yesterday* instead of */yesterday*, and with *n* jump to the next occurrence of *yesterday*, the end of a file is no longer the end of the search. We also search through all the files stated on the command line. After entering the asterisk *less* reports with *EOF-ignore* in the last status line, that it is remembering, for this search, to ignore the end of a file ("*End of file*").

Not browsing, but searching

Browsing was an important argument in favour of ASCII vocabulary lists, but we don't want to do without a targeted search. For this purpose, *grep* is our friend:

```
[trish@lillegroenn eng_deu]$ grep yesterday *
BOOK.VOC:yesterday gestern
EXERCISE.VOC:gestern - yesterday
[...]
eng2ger.voc:gestern – yesterday
[...]
```

No matter how delighted we may otherwise be that *grep* tells us where it was found - for our reference purposes we are not exactly dying to know in which file *grep* found it. Luckily *man grep* declares...

```
 -h, --no-filename
   Suppress the prefixing of filenames on out➘
put
   when multiple files are searched.
```

that it is possible to turn off the mention of the filename with the flag *-h*:

```
[trish@lillegroenn eng_deu]$ grep -h yesterday *
yesterday gestern
gestern - yesterday
[...]
gestern – yesterday
[...]
```

But this brings the disadvantage that the vocabulary is distributed throughout several, sometimes thematic, ASCII files with the filename endings *.voc* or *.VOC*, even more to the fore: The various files use different conventions, to separate phrase and translation from each other. In order to filter out duplicates, there is only one thing left to do: We must tailor all the files to a single convention.

Egalitarianism

eng2ger.voc separates the German vocabularies with two hyphens and a space before and after its respective English translations:

```
erst gestern --only yesterday
```

Since this is by far the largest file, it is advisable to transfer its convention to the other files.

In the case of *EXERCISE.VOC* this is not so hard: This file retains it with a hyphen (-) between the spaces, which we quickly replace with *sed*:

```
[trish@lillegroenn eng_deu]$ sed -e „s/ - / ➘
-- /" EXERCISE.VOC > EXERCISE.VOC_
```

The *sed* command *s* quickly and simply substitutes the first occurrence of *SpaceMinusSpace* in each line with *SpaceMinusMinusSpace*. We actually receive the result of this command, applied to *EXERCISE.VOC*, as the standard output. But since we would rather see it in a file, we use > to divert the output into the file *EXERCISE.VOC_*. After we have checked that the new file looks reasonable, a

```
[trish@lillegroenn eng_deu]$ mv EXERCISE.VOC➘
_ EXERCISE.VOC
```

is sufficient to overwrite the old with the new file.

The file *BOOK.VOC* imposes higher demands. Here a simple space serves as the dividing symbol:

```
yesterday gestern
```

So that there can be no confusion with spaces between words in phrases, these are marked by an underscore, which fortunately does not occur as part of a word:

```
yearn sich_sehnen
```

So here we have to replace twice: the first space in each case by *SpaceMinusSpace* (*s/ / — /*) and globally, every occurrence of _ by space (*s/_/ /g*). Combined, this looks like so:

```
[trish@lillegroenn eng_deu]$ sed -e "s/ / -- ↵
/" -e "s/_/ /g" BOOK.VOC > BOOK.VOC_
```

Compare and contrast

Before we overwrite *BOOK.VOC* with *BOOK.VOC_*, we would like to check the new file, thus compare it with the original. But *diff* is not suitable for this, as it outputs all *lines* which are different, and that's all of them... What we need is a *word*-based *diff*: *wdiff*. If this does not come with the distribution, it is available from *http://rpmfind.net/linux/rpm2html/ search.php?query=wdiff* or *http://packages.debian. org/stable/text/wdiff.html*.

```
[trish@lillegroenn eng_deu]$ wdiff --help
Usage: wdiff [OPTION]... FILE1 FILE2
[...]
 -3, --no-common     inhibit output of common↵
 words
```

With the option *-3* you can thus avoid having *wdiff* output words which have stayed the same. If we send the entire output through *less* again, we are also preventing something slipping by us when we look through:

```
[trish@lillegroenn eng_deu]$ wdiff -3 BOOK.↵
VOC BOOK.VOC_ | less
[...]
========================================
{+--+}
========================================
[-you can-] {+--you can+}
[...]
```

wdiff's output does, admittedly, take some getting used to: The = line functions merely as a dividing line. In *[- -]*there are strings from *BOOK.VOC*, which have been replaced in *BOOK.VOC_* by the character string in the *{+* - brackets. The *{+--+}* means that in *BOOK.VOC_* simply two minus symbols have been added – spaces are easier to ignore for the word-based *diff*.

The output is more readable in the so-called *less* mode, which does not really have very much to do with *less*. But nevertheless,

```
[trish@lillegroenn eng_deu]$ wdiff -3l BOOK.↵
VOC BOOK.VOC_ | less
[...]
========================================
--
========================================
you_can --you can
[...]
```

waives the unwanted bracketing and thus makes the output easier to read.

But we have no desire to go through the entire *less* output, and we ponder the following: If we have done everything right, *wdiff -3* will throw out exactly the same number of -- lines as *BOOK.VOC* (and *BOOK.VOC_*) has lines (lines: *-l*):

```
[trish@lillegroenn eng_deu]$ wc -l BOOK.VOC↵
BOOK.VOC_
 29018 BOOK.VOC
 29018 BOOK.VOC_
 58036 total
```

If we filter all the distracting dividing lines out of the *wdiff* output, we should in fact again end up with 29018 lines (*grep -v* seeks out all the lines with no ==):

```
[trish@lillegroenn eng_deu]$ wdiff -3l BOOK.↵
VOC BOOK.VOC_ |grep -v "=="| wc -l
 29023
```

So that did not go quite as planned - where do the five extra lines come from? Clever as we are, we will simply display all the lines which contain *no* double minus:

```
[trish@lillegroenn eng_deu]$ wdiff -3l BOOK.↵
VOC BOOK.VOC_ | grep -v "==" | grep -v "--"| wc -l
Usage: grep [OPTION]... PATTERN [FILE]...
Try `grep —help' for more information.
     0
```

But we've certainly done something wrong here... Of course – minus signs serve as symbols for the shell that there is an option to follow, and the strength of the double quotes is not enough to hide our minus search pattern from the shell.

Luckily we can remember the old Bash trick, of telling a command with a – that there are no further options to follow:

```
[trish@lillegroenn eng_deu]$ man bash
[...]
OPTIONS
[...]
  -  A single - signals the end of options and
disables further  option processing.  Any
arguments after the - are treated as filenames
and arguments.  An argument of  -- is
equivalent to an argument of -.
[...]
```

With a

```
[trish@lillegroenn eng_deu]$ wdiff -3l BOOK.↵
VOC BOOK.VOC_ | grep -v "==" | grep -v – "--"↵
wc -l
     5
```

thus we come to the missing five lines. But where did they come from?

```
[trish@lillegroenn eng_deu]$ wdiff -3l BOOK.↵
VOC BOOK.VOC_ | grep -v "==" | grep -v – "--"
```

```
arrow_keys

sensing_mark
```

Three empty lines, which *wdiff* has somehow included, but what is going on with *arrow_keys* and *sensing_mark*? The same command without the *l* option for *wdiff* provides information, and

```
[trish@lillegroenn eng_deu]$ wdiff -3l BOOK.↵
VOC BOOK.VOC_ | less
```

lets us track down the corresponding point by comparison with the *less* command */arrow_keys*. Look at this:

```
[-arrow_keys-]
{+arrow keys –+}
```

The fault (for the empty lines, too) lies clearly with *wdiff*.

With all this toing and froing we had almost forgotten why we dragged out *wdiff* in the first place: We wanted to check if, in the lines where we replaced underscores, everything had gone smoothly. Here we would prefer to take *wdiff* without the *l* option, because then we could exclude all lines in which a *{+--+}* occurs:

```
[trish@lillegroenn eng_deu]$ wdiff -3 BOOK.V↵
OC BOOK.VOC_ | grep -v "==" | grep -v „{+--+}"↵
 | less
```

Everything in order? Then we simply overwrite the old *BOOK.VOC* with the converted content from *BOOK.VOC_*:

```
[trish@lillegroenn eng_deu]$ mv BOOK.VOC_ BO↵
OK.VOC
```

Grep and paste

As if we hadn't already gone to enough trouble, *technic.voc* presents us with a disproportionately difficult task: Here stand the original and the translation, each on their own line, and the pair is separated from the rest of the vocabularies by an empty line in each case:

```
Ab-; Abfall
waste

abfuehren
discharge

[...]
```

With *sed* on a command line, nothing more will come of this, because here we must replace line breaks with " -- " and additionally eliminate empty lines. It also becomes difficult to construct a halfway comprehensible one-liner with Perl for this. But luckily the file is so regularly constructed that – once we have removed the empty lines – an odd, and the following even line, always go together.

We can get rid of the empty lines by using *grep* to seek out all those lines in which at least one letter *a-z* and/or *A-Z* occurs:

```
[trish@lillegroenn eng_deu]$ grep [a-zA-Z] t↵
echnic.voc
```

Now it gets a bit more difficult. But then we remember the *cut* command, with which columns can be extracted from text files. Where's there's a *cut*, there must also be a *paste*, which combines several columns into a file. In fact, we find it with *man paste*.

With *-d* we can specify a column delimiter – unfortunately only a single letter, but we can still replace that later with *sed*. What matters is only that the Delimiter does not occur in *technic.voc*. How would # work? Let's count:

```
[trish@lillegroenn eng_deu]$ grep -c "#" tec↵
hnic.voc
0
```

The hash symbol ("#") occurs precisely 0 times in this dictionary file and is therefore ideally suited as a temporary column delimiter for *paste*.

The rest is perfectly simple: *paste* wants to have as argument just the two files which serve as first and additional column(s). Now we have no files at all, but the manpage tells us that *paste* is also satisfied with the standard input (from the **Pipe** of *grep*), if instead of a filename we insert a -.

In fact we can settle quite happily for the standard input *STDIN* ("*standard input*"); this has in particular the nice property that a line disappears from STDIN as soon as it has been read out once. If we twice replace *paste* in an admittedly dastardly move for STDIN, we obtain precisely the effect we want: In the first column are the odd lines, in the second column the even lines:

```
[trish@lillegroenn eng_deu]$ grep [a-zA-Z] t↵
echnic.voc | paste -d "#" - -
Ab-; Abfall#waste
abfuehren#discharge
[...]
```

To remove the hash symbol from this is one of our easiest exercises, and we immediately divert the result into the *technic.voc_* file:

```
[trish@lillegroenn eng_deu]$ grep [a-zA-Z] t↵
echnic.voc | paste -d "#" - - | sed -e "s/#/ ↵
-- /" > technic.voc_
```

The result *technic.voc_* ...

```
Ab-; Abfall -- waste
abfuehren -- discharge
[...]
```

... can thus at the same time be renamed in *technic.voc*.

This means we have a sufficient selection of dictionary files (*BOOK.VOC*, *EXERCISE.VOC*, *eng2ger.voc* and *technic.voc*) in place – I will leave

Pipe: *written on the command line as |, takes the standard output of the commands standing to the left of it and feeds the command to the right-hand side.*

■

converting the rest to your inventive powers – and can finally turn to a small script, which takes over the translation of words entered on the command line.

Look me up

Like (almost) every shell script it begins by specifying which **Shell** we are using. Naturally the one with which we are most familiar, and that will usually be the Linux standard shell *bash*:

```
#!/bin/bash -vx
```

Turn around once

Of the four vocabulary files used here, BOOK.VOC displays one major difference from the others: The English term is on the left, the German match on the right. Since the wb script from Listing 1 does not recognise that for example gestern – yesterday from eng2ger.voc and yesterday – gestern from BOOK.VOC is a duplicate for our purposes, it is presumably simplest just to swap the columns in BOOK.VOC.

Like all the text modification exercises covered in this Answer Girl, there are several ways to achieve this goal. A few of them will be listed at this point by way of example.

Cut and paste

With cut columns can be extracted from a text file, which, with paste can be – and in reverse order, too - added back again. We explicitly specify the column delimiters with the option -d ("delimiter"). Unfortunately this can only be one character, not a character string, and that makes the whole thing somewhat fiddly:

```
[trish@lillegroenn eng_deu]$ sed -e "s/ -- /%/" BOOK.VOC | cut -d "%" -f 1 > /tmp/BOOK.VOC.1
[trish@lillegroenn eng_deu]$ sed -e "s/ -- /%/" BOOK.VOC | cut -d "%" -f 2 > /tmp/BOOK.VOC.2
[trish@lillegroenn eng_deu]$ paste -d "%" /tmp/BOOK.VOC.2 /tmp/BOOK.VOC.1 | sed -e "s/%/ -- /" > /tmp/BOOK.VOC.paste
```

In the first two lines we replace the true column delimiters in each case "--" with the working delimiter %. Line one, with cut -f 1, then fetches out everything on the left of the delimiting symbol, and writes it into the temporary file /tmp/BOOK.VOC.1. The same thing happens with the second column (-f 2) on the right of the delimiting symbol in line two – the output of this cutting-out action with cut lands in /tmp/BOOK.VOC.2. If we give paste as first argument in the third line the second and as second argument the first temporary file, we have swapped the columns from BOOK.VOC. Now just replace the percentage sign again with "--" and save the result of the swap action in /tmp/BOOK.VOC.paste. If everything has gone smoothly, the original file can be overwritten with this.

Pearls and expressions

It is of course less fiddly, too - but then we step into the area of independent script languages such as Perl. Perl can be used very well with the -p option as a more powerful sed substitute. As with sed the -e option ("execute") introduces a Perl command to be executed on the command line.

```
[trish@lillegroenn eng_deu]$ perl -pe 's/(^.*)( -- )(.*$)/$3$2$1/' BOOK.VOC > /tmp/BOOK.VOC.perl
```

Everything (.) from the start (^) of a line to the end ($) is to be replaced by a revised version. So that the content of the lines does not get lost, we save it in round brackets: the start of the line before the delimiter string "--" in the first buffer, "--" in the second and the rest up to the end of the line in the third buffer. The whole thing is now replaced by the content of the third buffer ($3), followed by the delimiter string from the second ($2) and the former line start from the first buffer ($1).*

Make sure that you set the Perl Substitute command in single quotes ('). Double quotes cause the shell to assume that $3$2$1 means the contents of shell, not Perl, variables.

As if it *were (k)*not a problem

The most elegant way is via awk. Contrary to paste, this tool can also manage with multi-character column dividers. All the same, the delimiter here is specified with the option -F ("Field separator").

```
[trish@lillegroenn eng_deu]$ awk -F "--" '{print $2 "--" $1}' BOOK.VOC > BOOK.VOC.awk
```

The awk-"program" in single quotes normally consists of a pattern, on which a command block is applied in braces. Since we mean the entire file, we need not specify any explicit pattern and settle for the bracket block.

In this we instruct awk to output the content of the second column ($2), then the delimiter string "--" and finally the content of the first column.

Errors often occur when developing a script, which is why we will first switch on the debug options *-vx*.

Provided */usr/dict/eng_deu* contains only converted dictionary files, we shall hold this dictionary directory in the variable *WBDIR*:

```
WBDIR=/usr/dict/eng_deu
```

As with any script intended for more than one person, we begin with a call up test: If the user enters more or less than *one* search term as argument (thus "*n*ot *e*qual") to one...

```
if [ $# -ne 1 ]; then
```

... we simply spit out how our script ought to be used:

```
     echo "Usage: $0 string"
```

Nicely enough, in the variable # a shell script recalls the number of arguments with which it was called up. In the variable *0* (null) can be found the reset argument, thus the command name itself (if applicable, with specified path).

On the other hand...

```
else
```

...we search in the vocabulary lists in the directory *$WBDIR* for the first command line argument (*$1*):

```
       grep -hw "$1" $WBDIR/*
```

With the "*W*ord option" *-w* we ensure that *grep* only outputs something when the search word pops up as such (and not for example as part of another word) in the vocabulary lists.

In order to exclude typing errors in upper and lower case, we can also force *grep* to *i*gnore differences between upper and lower case letters:

```
       grep -hwi "$1" $WBDIR/*
```

which means we really are finished and can close the *if* construction:

```
fi
```

Issue execution rights to our *wb* script

```
[trish@lillegroenn /tmp]$ chmod ugo+x wb
```

and test:

```
[trish@lillegroenn /tmp]$ ./wb
#!/bin/bash -vx

WBDIR=/home/trish/dict
+ WBDIR=/home/trish/dict

if [ $# -ne 1 ]; then
     echo "Usage: $0 string"
else
     grep -hwi "$1" $WBDIR/*
fi
+ [ 0 -ne 1 ]
+ echo Usage: ./wb string"
Usage: ./wb string
```

Thanks to the verbosity option *-v* ("*v*erbose") the Bash displays every single line, which it remembers to execute. The lines with the initial plus are something for which we can thank the *-x* ("e*x*tensive") option, which also states each time what the shell really sees internally, if it has performed all the replacements (read out the contents of variables). Last of all - and unfortunately not especially marked out - we also find the output with which we would have been faced without the debug options in the wilderness - in this instance: *Usage: ./wb string*.

And the variant with one search word functions:

```
[trish@lillegroenn /tmp]$ ./wb yesterday
[...]
yesterday -- gestern
only yesterday -- erst gestern
yesterday -- gestern
gestern -- yesterday
vorgestern -- the day before yesterday
[...]
```

No doppelgangers

This output clearly shows that we still have some plans for the script: We want to get rid of the duplicates. This is really quite a simple matter: sort the output with *sort* (thanks to *-f* — "*f*old" - with equal value for upper and lower case letters) and use *uniq* to throw out the doppelgangers:

```
       grep -hwi "$1" $WBDIR/* | sort -f | uniq
```

Unfortunately, there is something wrong with this, because the test run produces

```
[trish@lillegroenn /tmp]$ ./wb yesterday
[...]
gestern -- yesterday
only yesterday -- erst gestern
vorgestern -- the day before yesterday
yesterday -- gestern
yesterday -- gestern
[...]
```

which may be sorted, but it is still not free from duplicates. An investigation using *./wb yesterday > /tmp/test* of the output diverted into the file */tmp/test* with an editor comes up with: The sole difference between the two "*yesterday -- gestern*" lines is the **Whitespace** characters.

OK, then we'll standardise all these ('*[:blank:]*')s first into spaces (' ') and simplify all space sequences with the *tr* option *-s* ("*s*queeze") into a single one in each case:

```
       grep -hwi "$1" $WBDIR/* | tr -s '[:➘
blank:]' ' ' | sort -f | uniq
```

***Shell**: The command line interface between users and their input devices and the operating system. Most UNIX shells have a more or less powerful programming language built in.*

***$:** Shells such as the Bourne shell (sh), Korn shell (ksh) or the Bourne-Again-Shell (bash, under Linux also known as sh) reveal the content of a variable, if one places a dollar symbol before their name.*

***Whitespace**: Collective term for characters which mislead the eye into believing "There is no character here". These include space and tab characters.*

■

And yet the double line is still proving problematic: Of course, because now we have, in one output *none* and in the other precisely *one* space at the end of the line, which is bothering *uniq*.

So, with a sigh, we again pull out *sed* and replace a single space at the end of the line (*$*) with nothing

```
        grep -hwi „$1“ $WBDIR/* | tr -s ‘[:U
blank:]’ ‘ ‘ | sed -e „s/ $//“ | sort -f | uniq
```

Et voila – at last the *wb* script (Listing 1) is ready to go to work. Now the debug options can go, and *root* can copy it to */usr/local/bin* to be used by anyone. Since this directory is usually included in the *PATH* variable, it is now also sufficient to call up *wb* without specifying the path. ■

Listing 1: The dictionary script wb

```
#!/bin/bash

WBDIR=/home/trish/dict

if [ $# -ne 1 ]; then
      echo “Usage: $0 \“string string ...\“”
      echo “      $0 string“
      echo “      $0 regexp“
else
      grep -hwi “$1“ $WBDIR/* | tr -s ‘[:blank:]’ ‘ ‘ | sed -e “s/ $//“ | sort -f | uniq
fi
```

Value added

Sharp-eyed readers may be wondering how, in Listing 1, one proposed line suddenly turned into three echo lines. Anyone who has experimented a little with the script (or grep) will know that one can suggest to the shell, by including several strings in quotes, that despite everything, there is only one argument involved.

As soon as users want to search for an expression consisting of several words, they simply have to place it in double quotes:

```
[trish@lillegroenn /tmp]$ wb “sich erinnern“
recollect -- sich erinnern an
remember -- sich erinnern
sich erinnern --remember
[...]
```

We should of course document this type of use:

```
echo “Usage: $0 \“string string ...\“”
```

So that echo does not wrongly interpret the quotes to be output as the delimitation of its own argument, they have to be escapt (stripped of their special position in the shell) with \.

The last echo line

```
echo “      $0 regexp“
```

on the other hand intends that grep, right from the start, looks not only for character strings, but also for regular expressions (“regexps“). This means for example that the user can elegantly skate over any uncertainties in terms of spelling:

```
[trish@lillegroenn /tmp]$ wb “ye.*y“
erst gestern -- only yesterday
Freibauern -- yeomanry
gelbliche -- yellowly
gestern -- yesterday
hefig -- yeasty
[...]
```

*searches for the translation of words beginning with ye and ending in y. The dot here stands for any character, and the following * signals that any number of (at least none) should pop up from this.*

The only thing to watch here is that with regular expressions, too, the saying applies: “Some are more equal than others.“ Although the ground rules are the same, such as not all perl regexps can also be used with grep. It’s therefore often worth taking a look at the grep man page...