TEXniques

Toolbox: the toolbox?

Maarten Gelderman

abstract

This MAPS is about the TEX Toolbox, about other programs than TEX itself. So this MAPS's toolbox should probably deal with this kind of material. As a consequence this toolbox is even more eclectic than earlier ones. First I will show you how I make mailings to NTG-members, by combining Excel and LATEX. Next I will present the most ugly regular expression I know of, and finally I will say something about using makefiles.

keywords

mail-merge, regular expressions, emacs, Excel, makefiles

Using Excel and LATEX to do a mail-merge

A recurring question on forums like TEXNL is whether it is possible to use LATEX to do mail merging. Of course the answer is yes, and numerous ways are available. One might create a single large TEX file by using the reporting facilities of a database program like Access. However, as 99% of the resulting document will consist of mere repetitions of the body text, this approach is rather inefficient. Table 1 shows a solution that avoids repetitions. The trick is to define a new command (`\myletter`) which contains the text of the letter, the `letter`-environment and the `\opening` and `\closing` statements. In the body of our document, we now repeatedly use the `\myletter` macro with appropriate arguments to produce the individual letters.[1]

The `\myletter` macros can be generated by a database application, by a Perl script, of even by using a simple spreadsheet-formula. Of course it is more elegant to produce the file directly with Perl or the database manager, however, for most applications using a spreadsheet suffices. Elementary functions for sorting the database and making selections are available. The only thing we have to do is to find a way to put the required fields of the database in a single cell for each row. This also is easily accomplished. In a single cell, we enter a formula similar to that presented underneath:

```
=CONCATENATE("\myletter{";+IF(K2<>"";\
+CONCATENATE("Dear ";K2;",");\
"Dear member,");"}{";J2;"}{";I2;"}\
{";L2;"}{";M2;"}{";+IF(O2<>"";\
O2;N2);"}{";P2;" ";Q2;"}{";R2;"}")
```

Table 1. Mail merge using the standard LATEX letter-class

```
\documentclass{letter}
\newcommand{\myletter}[8]{%
  \begin{letter}{#2 #3\\#4 #5\\#6\\#7\\#8}
    \opening{#1}

    This is a letter. A rather short one, but a
    letter nevertheless.

    \closing{With kind regards}
  \end{letter}}

\begin{document}
\myletter{Dear Karel,}{J.F.}{Krammers}
      {University of Nowhere}{%
         Department of Improbable research}
      {Piet Heinstraat 10}{1399 EW Muiderberg}{%
         Nederland}
\myletter{Dear Jan,}{J.H.}{Drupnats}
      {Ministry of Silly Walks}{}
      {Binnenhof 30}{2222 KH Den Haag}{Nederland}
\end{document}
```

This formula (the backslashes at the end of the lines just indicate that it should be put in a single cell) concatenates a number of text fields. Everything between quotation marks is put into the cell verbatim. The letter number-combinations point to the cells containing the data. The K-column, for instance contains the first name of the addressee. The if-clause checks whether a first name is present in the database. If this is the case it puts 'Dear firstname,' in the first parameter field of the `\myletter`-command, if it is not present, 'Dear member,' will be used instead. The remainder of the formula refers to the other fields of the database in a similar way.

After selecting the appropriate records, we just use copy-and-paste to put the results of the concatenation formula into the TEX-file and are ready to generate our document.

An unreadable regular expression

If a produce-the-least-readable-regular-expression-contest would exist, the next one probably would have a good

1. In terms of computation time required this approach is still rather inefficient: the bodytext has to be typeset by TEX for each individual letter. It is possible to solve this problem, but that would probably cost more time than we would ever save by this improvement.

chance of winning it:
`\\\([’\|\^\|‘\|"]\)\([^\{]\)` → `\\\1{\2}`
Perhaps the most surprising thing about this regular expression is that it is actually useful. Although it admittedly is prone to typing errors, it is even easy to understand.

What does this regular expression accomplish? It replaces every occurrence of an accent directly followed by a character, with the same accent, followed by the same character, but after replacing the character is placed between curly braces. A small example: `\’a` becomes `\’{a}`, `\^e` becomes `\^{e}`, and so on.[2] How do we accomplish this replacement? First we have to decide which program we want to use. I decided to use emacs, as I do all my editing in this program, but one might decide to use any other program that is able to deal with regular expressions. Perl would be another likely candidate. Now we have to build the regular expression we want to use for searching. This also is fairly easy. The first character we are looking for is the backslash. Unfortunately this is a character with a special meaning, hence we have to escape it. To indicate that we are looking for a single backslash, we start the regular expression with `\\`. The next character we will be looking for is the accent. We want to treat this accent slightly different from the backslash: it has to be stored as we will need it in the replacement operation. In order to store it, we start a new group. This is done with the next two characters in the regular expression: `\(`. A few characters later, the group will be closed with `\)`. The sequence between the opening and the closing of the group will be used to find the accent and store it. As this is our first group, it will be saved in `\1`.

The group itself contains the different accents. The accents are separated by the ‘or’-operator: `\|` and all alternatives are places between bracketed braces (`[]`), the order in which the accents themselves appear is irrelevant. The `’` is found by the first character in the group, which is identical to `’` itself, `^` has a special meaning in regular expressions, so it is escaped and will be found by `\^`, `‘` and `"`, don’t have special meanings and can be used directly.

After matching the accent we open a new group with `\(`, which is closed by the `\)` at the end of the regular expression and the contents of which will be stored in `\2`. This group is used to find the character to be put between braces. The character itself is defined in a somewhat peculiar way. Instead of listing all possible characters, we define this group as every thing that is *not* a curly brace (if we wouldn’t do this `\’{a}` would be replaced by `\’{{}a}`, which it not wrong, but not particularly desirable either). The operator `^` is the not-operator, and the curly brace itself is escaped by putting a backslash in front of it.

The search regular expression has been constructed by now. The replacement expression if fairly short. First we put the backslash in place again (`\\`), next we put the accent (which has been stored in register 1) in the replacement text by the command `\1`, next we place the opening curly brace (which this time does not have to be escaped: {), the character itself (which is stored in `\2`) and the closing curly brace (}) and we are done.

Makefiles

If you already know something about makefiles, you can skip this section. I know hardly anything about this topic. Until recently my conviction was that makefiles were made by other people who had written a program I wanted to install, and the only thing I ever did with a makefile was to change some site specific settings in the first few lines of it. As those latter modifications lead to errors every now and then, I decided it might be worthwhile to read the manpages. It turned out they didn’t exist. A manpage for the make-program did exist, but did not contain any information about makefiles themselves. Fortunately, it did refer to a file in my doc-directory. As this file did not exist either, I decided to follow a somewhat unusual approach: I consulted a real manual. Here I learned that makefiles, although you can probably do very complicated things with them, are in fact really simple. The only thing you have to remember is that you have to use tabs for indenting and not spaces.

Let’s examine a very simple makefile:

```
doc.dvi: doc.tex
      latex doc
      bibtex doc
      latex doc
      latex doc
```

Make sure that on line 2–5 tabs instead of spaces are used for indenting. We save this file and give it the name Makefile (with a capital M, if you choose another name, you have to use `make -f file` instead of just `make`). After saving the makefile, we can compile our document by simply entering the command `make` from the command line. Make checks whether doc.tex is newer than doc.dvi and if this is the case, it runs the four commands on line 2–5.[3]

The example given above, although useful, is not very inspiring.[4] A slightly more complex example is given below:

```
all: dea.dvi dea.1
        dvips  -f dea.dvi -o print.ps
```

2. The only reason why this operation is useful, is that latex2rtf cannot deal with accented characters without the curly braces.

3. If you want make to run those four commands any time you enter make, just replace `doc.dvi` by some other name, e.g., `dvi`. The file `dvi` does not exist and will never exist. Consequently make will always run the four commands.

4. It is not very efficient either, each line is executed in a separate shell, it would be more efficient to put the four commands on one line separated by semicolons.

```
dea.dvi: dea.tex dea.1
        latex dea
        latex dea

dea.1: dea.mp
        export TEX=latex; mpost dea

clean:
        rm *~ *.dvi *.log *.?lg
```

If we just type `make` (or `make all`), make will first check whether the files mentioned after `all:` are up-to-date. If not, that is for instance if `dea.tex` or `dea.1` has been modified more recently than `dea.dvi`, first the commands for those files are run and next the command for all is run. We could also run those commands separately, by entering `make dea.1`, make would only check whether `dea.1` is up-to-date and run the required commands if necessary. Another interesting featue is the line `export TEX=latex; mpost dea`. First we tell our shell that LaTeX should be invoked to process TeX-commands issued by MP, and next we run MP. Because each line has its own shell, this setting does not carry over to other commands (and consequently, we cannot put them on separate lines). Makefiles can be made more complex if desired. It is, for instance, possible to use parameters to make a Makefile more generic (in this example I also added two comment lines):

```
# Begin definitions
FILE=dea

# Begin dependencies
all: $(FILE).dvi $(FILE).1
        dvips  -f $(FILE).dvi -o print.ps

$(FILE).dvi: $(FILE).tex $(FILE).mp
        latex $(FILE)
        latex $(FILE)

$(FILE).1: $(FILE).mp
        export TEX=latex; mpost $(FILE)
```

This example should be clear withour further explanation. One final remark, which will also end this toolbox, is in order however. The variables declared can be used by prepending them with a dollar sign. In order to use the dollar sign itself, it has to be escaped by another dollar sign. The next excerpt from a Makefile demonstrates this latter feature.

```
fonts:
        tex pad

tfm:
        for f in *.pl; do pltotf $$f; done
        for f in *.vpl; do vptovf $$f; done
```