The mkpattern program

Javier A. Múgica de Rivera

Version 1.1, April 23, 2007

Index

1. Introduction

There have been in the past people who wrote scripts for the generation of hyphenation patterns. They either used a language different from TEX, wrote a program specific for the needs of their language, or they didn't distribute it (there was none at Ctan at the time of writing this paper).

The mkpattern program is just the file mkpatter.tex. It is a general purpose pattern generating program. It allows the user to define letter sets and to write the patterns in a template-like manner, with the several options described below.

2. How to use it

You should create a generating file to be processed with TEX that at some point inputs mkpatter.tex. The program was created with occasion of the need to write the Galician patterns, so the file mkpattern-exmpl.tex, (almost) the source for version 2.1 of those patterns, is a quite complete example of a generating file.

The order `\input mkpatter.tex` need not be at the very first line, since you may define macros before that point. The general structure of the generating file is:

```
\input mkpatter.tex
⟨optional change of the output file name⟩
\begin{code}
⟨definition of user macros⟩
\end{code}
⟨definition of letter sets⟩
⟨other definitions⟩
\begin{pseudopatterns}
⟨pseudopatterns⟩
\end{pseudopatterns}
\end{}
```

Once this file is written it has to be processed under INITEX —e.g., by typing `initex` ⟨*filename*⟩ on the command line— (it will also work under PlainTEX or LATEX). If no other name is given (see the next section), a file with the same name but extension `.pat` will be created. This is the patterns file, and has the following structure:

```
⟨catcode settings⟩
{
⟨setting of active characters⟩
\patterns{
⟨patterns⟩
}
}
```

⟨*patterns*⟩ consist only of patterns and comments, provided the generating file is correct.

3. Changing the name of the output file

The file to be written can be given a name different form the default one. Just write

```
\filename{⟨filename⟩}
```

before any write action is performed —simply write it just after `\input mkpatter`.

4. Defining letter sets

The ⟨*definition of letter sets*⟩ is a set of instructions with this syntax:

`&`⟨*set name*⟩`={`⟨*list of characters*⟩`}`

For example, `&V={a e i o u}`. The ⟨*set name*⟩ may contain, excluded numbers, braces, ampersands and equal signs, almost any character. The ⟨*list of characters*⟩ is a space separated list of usually, but not necessarily, single characters, as well as previously defined sets of characters, as in the following example:

```
&V={a e i o u}
&V+={&V h}
&Va={á é í ó ú}
&Vall={&V &Va}
&Vall+={&Vall h}
```

5. Other definitions

⟨*other definitions*⟩ is three kinds of definitions, two of which are character related. One is the command

`\encodingsubstitutions`

It takes two arguments, each one being a space separated list of characters. The first includes the characters to be replaced, and the second the corresponding replacements. For example, the file mkpattern-exmpl.tex declares the following encoding substitutions:

```
\encodingsubstitutions{á é í ó ú ñ ü ï}{^^e1 ^^e9 ^^ed ^^f3 ^^fa ^^f1 ^^fc ^^ef}
```

The patterns are specific for a particular font encoding, but the text editor you are using may follow a different encoding. For example, the character á is in position `E1` in the T1 encoding, but it may be represented by a different number by your text editor. Thus, whenever you write a pattern that contains á, and you see an á in your text editor, you are not writing the pattern you intended for the T1 encoding. Suppose á is stored as `E7` by your text editor. That position is ç in the T1 encoding, so, even if you see an á, you are writing a pattern containing ç instead.

In order to achieve the correct results, all the á's that you wrote have to be replaced by T1's á, i.e., character `E1`, which may be whatever in your text

editor encoding. This can be done either at the time of writing the patterns, or by means of active characters at the time INITEX shall read them. The program mkpattern takes this latter approach. Suppose that the slot `E1` (á in T1) is interpreted by your text editor as ø, then the above declaration will result in a set of definitions written in the patterns file, before the patterns, the first of which will look like

```
\catcode`\á=13 \edefá{\string ø}
```

This will cause INITEX to replace each occurrence of your text editor's á with your text editor's ø, i.e., T1's á. These definitions are written inside a group, so that they remain local.

If the character to be replaced is the same than the replacement character, then making it active is superfluous, and mkpattern will not write the definitions to the file. The machine where mkpattern-exmpl.tex was written happened to represent all the needed letters above position 128 in the same places as the T1 encoding, so if you process that file no definition will be generated.

You may wonder why mkpattern writes `\edefá{\string ø}` and not simply `\defá{ø}`. The reason is that the set of characters to be replaced and the set of replacement characters may have nonempty intersection, which renders the later approach invalid.

The other character related command of ⟨*other definitions*⟩ is `\letters`, that takes as argument a space separated list of characters. The catcode of these characters will be made equal to 11 in the patterns file. You write the characters normally with your text editor encoding; mkpattern will replace them if necessary.

Successive occurrences of `\encodingreplacements` and `\letters` are added to previous ones.

The relative order of encoding replacement declarations, letter declarations and letter set definitions is free, because mkpattern simply stores the information necessary when processing those commands, and it does not write anything to the patterns file till it sees the beginning of the patterns.

Finally, the other predefined command is `\templatedef`, and example of the use of which is `\templatedef{prefixos}{#1&{V}2}`, and later in the pseudopatterns you could write

```
\begin{prefixos}
⟨#-parts⟩
\end{syntax}
```

It is selfexplanatory once you know about templates, which is explained below.

6. The pseudopatterns

The pseudopatterns are the bulk of the generating file. There you may write bare patterns or "patterns for the generation of patterns", thus the name pseudopatterns. These are patterns where one or more letters have been replaced by letter sets, and the program writes a pattern for each letter of the set. Thus, for example, the pseudopattern

`&{Vall}1`,

with the set `Vall` defined as above, would be replaced by the five patterns `a1`, ..., `ú1`; and the input `&{Vall}2&{Vall+}` would write to the patterns file a pattern matrix starting at `a2a` and ending at `ú2h`. If the name of the letter set consists of a single character you may omit the braces.

If a single pattern is written it will be transparently passed to the patterns file. Indeed, any text that does not deserve any special treatment is directly output to the patterns file. Any such text as well as pseudopatterns are *words*. Words must be preceded and followed by one or more spaces or end of lines. Remember this when you write control sequences or comments (do not append them directly to a word).

7. Templates

Some times you may find that the synthesis provided by letter sets is not enough, as you may need to write several pseudopatterns which share a common structure. You may then write a template, in a way similar to the alignments of TEX. The syntax is:

`\template{`⟨*pre-text*⟩`#`⟨*post-text*⟩`}`
⟨*#-parts*⟩
`\end{template}`

For example, the file mkpattern-exmpl.tex defines the following template for prefixes:

```
\template{#1&{V}2}  %Prefixos
⟨#-parts⟩
\end{template}
```

In this case the ⟨*pre-part*⟩ is empty. The ⟨*#-parts*⟩ is any text that may also appear outside the template scope. The program prepends and appends the ⟨*pre-part*⟩ and ⟨*post-part*⟩ to any word, and then it reads the resulting string as if it had been there from the beginning.

Some lines below in the same file the template `\template{co2# \newline}` is defined. This example illustrates the fact that the ⟨*pre-text*⟩ and the ⟨*post-text*⟩ may contain spaces as well as control sequences. Spaces will be respected whenever they don't follow a control sequence.

Templates cannot be nested. (They actually can, but the effect is not to apply both, but to apply the innermost).

8. Exceptions

It is very likely that a pseudopattern or template applies to all but one or a few patterns. Such exceptions can be specified prior to their generation with the command `exceptions`:

`\exceptions{`⟨*exceptions*⟩`}{`⟨*replacements*⟩`}`

⟨*exceptions*⟩ is a space separated list of exceptions. ⟨*replacements*⟩ may be either empty or a comma separated list of replacements. This command may appear more than once; it simply accumulates exceptions.

If ⟨*replacements*⟩ is empty mkpattern will generate for each pattern in ⟨*exceptions*⟩ a "hole" of equal size than the removed pattern, so that the output remains nicely formatted and the exception can be spotted at the first sight. Note that empty means exactly that, so a space is not considered empty. If it is not empty it must include exactly $n-1$ commas (not hidden by braces), where n is the number of patterns in ⟨*exceptions*⟩. The replacement text for each exception is copied verbatim to the patterns file, without further processing. This has some limitations, as is for instance that you cannot replace a particular pattern with a pseudopattern. Note, however, that you may still replace one pattern with two or more patterns, and that one of those patterns may be the original pattern (there is a case of this in mkpattern-exmpl.tex).

Each pattern written to the patterns file is followed by a space, and it is so for the replacement text, so even if ⟨*replacements*⟩ includes only commas at least a space will be written in the place of the missing pattern. If there is just one pattern to be replaced the least you can get is *two* spaces, for in that case the replacement `{}` is interpreted as explained above. If you *really* want the replacement to be as short as possible, then write a faked pattern:

```
\exceptions{a2a fff}{,}
```

9. Comments, blanks and arbitrary output

mkpattern has a command that allows you to place any text (any balanced text) in the output file:

`\put{`⟨*balanced text*⟩`}`

The most frequent elements that need to be written, and that cannot be written by simply placing them in the generating file, are comments and empty lines. mkpattern treats the end of a line and empty lines as spaces, so

the only way to insert an end of line in the patterns file is with the command `\newline` (you can include end(s) of line in the argument to `\put`, but it will output the character 13, which may or may not be the end of line for your OS). You may occasionally need to write two `\newline` instead of one. As it has already been shown, this command may appear in the definition of a template.

Comments may be output with `\put`, but there are other possibilities. mkpattern has three states regarding the treatment of comments:

```
\nocomments          \halfcomments          \keepcomments
```

`\nocomments` makes mkpattern ignore all comments (by simply doing nothig) except those inside `\put`. `\halfcomments` is necessary if you want comments to appear in the arguments to functions other than `\put`. There are actually two cases. The one is inside the definition of a template, and the other in the replacements of `\exception`. While the former is unlikely to be ever needed, the later can turn to be useful. Moreover, in this latter case `\put` does not work, since the replacement is copied verbatim. One reason you may need it is that the exception is the last pattern of its line, in which case a hole will not be perceptible. You may then write

```
\exceptions{z1z}{%%%}
```

to point out the absence of the pattern. You may also want it in order to emphasise an exception. Thus, a line of mkpattern-exmpl.tex could have been written in this manner:

```
\exceptions{.de3s2esper}{.de3s2esper de4s3esperanz %Note! Exception}
```

The comments inside the replacement of an exception can only contain commas if they are hidden by braces, but the braces will also appear in the output.

`\keepcomments`, in addition to the behaviour of `\halfcomments`, writes to the patterns file any comment that is read in the generating file. Each comment is written on a line by its own, which is usually the desired behaviour. If you want the comment to appear after some text in the line use `\put`.

Before the pseudopatterns start, comments can only be placed in the patterns file with `\put`. You may also write `\newline`'s there.

10. Input files

In order to input a file you need to write `\mkinput{`⟨*filename*⟩`}`. The file must include the `\endinput` command.

Currently, outside the pseudopatterns a common `\input` will work, but you'd better not rely on that, for it may change in the future while `\mkinput` will always work.

11. User defined macros

It is obvious that mkpattern makes substantial changes to the catcodes. In this respect the program is quite radical, which makes sense if you think that the main task of TEX is usually to output pages or read macro definitions, while the mkpattern program processes a file with the purpose of writing another text file. It has to catch every character of the input file, so that instead of being added to a box to be eventually typeset it is added to the material to be eventually written to the patterns file, after prior processing if necessary.

All these is done by means a control sequence named `\lee`. It is triggered by `\begin{pseudopatterns}` and does not stop to act till the end of the input, which is signalled by `\end{}`. When the command `\lee` sees another command in front of it, it executes that command, thus yielding temporarily the control to that command. Every command designed in mkpatter.tex and liable to appear inside pseudopatterns, places a `\lee` after it has completely expanded, restoring the control to the main reading flow of the program. In the like manner, every control sequence designed by the user should place the control sequence `\lee`, or any other that does that, before any text, lest the following text goes to the dvi file. Thus, the following definitions are valid:

```
\def\foo{\newline c2c}  \def\twolines{\newline\newline}
\def\foo{\lee c2c}      \def\foo{\exceptions{1h}{}}
\def\foo{\show\a\lee}
```

But the following ones are not

```
\def\foo{c2c \newline}    \def\foo{\show\a}
```

Before pseudopatterns, `\lee` is equal to `\relax`, so it makes no harm.

Since the program "destroys" the catcode settings, macros have to be defined in the following manner:

```
\begin{code}
⟨user definitions⟩
\end{code}
```

or before the program is read. The latter is not possible if the file is processed with INITEX. The code environment does not open a new group level, so the scope of macros defined there is the enclosing group.

In addition, some usual conventions regarding registers are changed. Among the allocation routines only `\newcount`, `\newtoks`, `\newread` and `\newwrite` are available. Also count registers 0 to 9, usually reserved for page numbering, are available for scratch use. The allocations have to take place after the program file is read. This doesn't disable the use of these to be allocated registers when defining macros prior to the reading of the program, provided that they never get expanded at that time.

There is another environment for defining macros:

```
\begin{patternscode}
⟨user definitions⟩
\end{patternscode}
```

It sets the catcode equal to those that will be in force inside the pseudopatterns. This concerns in particular `#` and `&`. The macro parameter character inside that environment is `$`. The above definition of prefixes, `\templatedef{prefixos}{#1&{V}2}`, can be performed by hand as

```
\begin{patternscode}
    \def\prefixos{\template{#1&{V}2}}
    \endprefixos{\end{\template}}
\end{patternscode}
```

You may even define a set of macros that completely replaces the task of mkpattern of generating the patterns. If, after the complete expansion, your macros result in the control sequence `\lee` followed by a (possibly long) list of patterns, those patterns will be placed in the patterns file. Remember to insert a `\newline` from time to time. If you add some `\exceptions` before generating the patterns you may use mkpattern as a formatter for your patterns.

When material to be output is seen by the program it is not usually directly output, but it instead gets placed in a buffer called `\linesofar`. This is also what `\put` does. If you want something to be output immediately, before the current line is completed, write

`\iwrite{`⟨*material*⟩`}`,

but recall that each write goes to a separate line.

12. Tracing and the log file

Mkpattern displays in the log file each defined letter set, as can be seen by running the example. If `\tracingexceptions` is positive it will also write the exceptions found and the subsequent replacement, as well as the exceptions that were written but did not arise.