makedoc—Preprocessing documentation by TEX

Uwe Lück — http://contact-ednotes.sty.de.vu

April 16, 2009

Abstract

makedoc provides commands for generating LATEX input from a package file in order to typeset documentation of the latter (somewhat similar and opposite to docstrip). Certain comment marks are removed, and listing commands are inserted. This continues the policy of niceverb to minimize documentation markup in package files. makedoc extends and exemplifies the parsing package fifinddo. After an edit (and test) of your package, you get the new documentation in one run (or two or three runs—for labels and TOC, as usual) of the documentation driver file.—This is an *alternative* to the standard doc package and its `\DocInput`. The present approach provides *less* than doc does, rather deliberately. It may be helpful at least for the development of small packages, or at least at early stages.

Contents

1 Introduction

The abstract will not be repeated in this section. Let me add instead that I was in dire need of such a package, I got stuck with my packages because I lost orientation in them, and I was unhappy with the forms of documentations of my other packages, and documenting them with the doc system was not attractive for me (neither considered helpful). I also worked on *Windows* until September 2008, and I find a system like the present one still more attractive then using (learning!) other filtering utilities (see below on awk). And I may work on *Windows* once again and don't want to depend on installing some ... there.

1.1 Prior work and what is new

It is, of course, not a new idea to get around comment marks % to typeset the documentation. doc's `\DocInput` does this by making % an "ignored" character. This way you cannot use % for commenting comments (so doc offers a "new comment mark" `^^A`). You also cannot use % for commenting out code (that you are pondering or use for debugging only).

Moreover, doc requires enclosing package code explicitly by environment commands (behind comment marks). Stephan I. Böttcher with his lineno.sty and Grzegorz Murzynowski in gmdoc aimed at doing away with this requirement. lineno.sty contains awk scripts to remove starting comment marks and to insert listing commands. A file lineno.tex is generated that typesets the documentation. By the way, lineno.sty is full of discussions, but it is not docstripped—the maintainers never have received a complaint that inputting lineno.sty were too slow.

gmdoc seems to get around comment marks and insert listing commands *while typesetting* by a refined version of `\DocInput`, through some careful detecting and analysing comment marks, the approach resembles detection of lists in wiki.sty.[1] And this is a matter of principles—comparing the approaches of *preprocessing* (lineno.sty) and *"smart typesetting"* (gmdoc, wiki). Sometimes preprocessing seems to be simpler, sometimes detecting while typesetting. (Another example is the preprocessor easylatex of which wiki.sty is a much reduced "while typesetting" variant.) "While typesetting" may be easier when single characters or sequences of two or three encode markup information—but such detection can badly interfere with other packages etc. "Preprocessing" may be

[1] See gmdoc.pdf on `\DocInput`. You can learn a lot from this 220 pages document! I also find pauldoc and xdoc inspiring.

easier when entire "strings" of characters decide, which may be anywhere in a file line.

makedoc chooses *preprocessing*, as lineno.sty, but by *TEX*. There is a general discussion of this choice in the documentation of fifinddo. Preprocessing here can be done in the same LaTeX run as typesetting, though you can avoid incompatabilities with packages needed for typesetting (by inputting them only *after* preprocessing).

lineno.sty exemplifies why preprocessing with *TEX* may be preferable to preprocessing with other utilities: When I took over maintenance of lineno.sty, I needed hard work to get the awk script running. The *Munich* awk seemed *not* to behave as the *Kiel* awk (I chose a Munich nawk and reworked the script a little). TeX seems to have better fixed functionality than other utilities!

1.2 Basics on using makedoc

At least in the long run, using makedoc should not imply commitment to a certain design or to certain LaTeX packages for typesetting listings and documentations. Therefore, makedoc.cfg (currently) contains *local* or *personal choices*, but also *experiments* with future features of niceverb. Especially, (at present) the `packagecode` environment that makedoc `\write`s must be chosen. Currently this is the `listing` environment from moreverb with some modifications or extra settings. It may be vital to `\MakeOther` the active characters from niceverb in the setup of `packagecode`. See the *example* in section 3.

Finally, each package file to be typeset will need its own *script* of makedoc commands. It should fit into the preamble of the main file for documenting the package (currently just 5 commands seem to suffice, see the *example* in section 3 and section 2.8 on typical "bundling" *script commands*). As an alternative, you may prefer to have "content only" (as much as possible) in the main typesetting file and in its preamble only `\input` a separate script file.

Yes, the idea of documenting a package *here* is to have a separate "driver" file for typsetting the documentation. It may contain an introduction and a guide for users. The documentation of the package code that has been prepared by the makedoc script will be `\input`. Alternatively, the "driver file" could have title etc. only, or preamble and a minimal `document` environment only.

So there may be many files, which may look confusing, especially as compared with the doc procedure. However,

1. "One file distribution" still is possible thanks to the `filecontents` environment.

2. The makedoc script can create a batch file (fitting the system, maybe using Will Robertson's ifplatform, or texsys.cfg, or ...) that removes certain auxiliary files or moves them to a certain directory.

3. I find it helpful to have rather little "contentual" text in the package file.

4. The procedure now runs very smoothly, once the stumbling blocks have been overcome.[2]

1.3 Examples

The documentations of fifinddo and of makedoc itself are typeset using makedoc (niceverb.pdf as well, yet comments remained scarce). fifinddo.pdf documents fifinddo.sty, typeset from fifinddo.tex, likewise makedoc.pdf. Section 3 contains listings of makedoc.cfg and the makedoc script file mkfddoc.tex especially made for fifinddo.pdf. fifinddo.doc, makedoc.doc, and niceverb.doc are the TEX input files that were made with makedoc.sty—I have only looked at them when something was wrong (often syntax mistakes in typing).

The Wikipedia syntax feature %%␣===␣subsubsection␣=== is only used in fifinddo.sty and niceverb.sty.

2 Implementation

2.1 Preliminaries

Head of file (Legalese):

```
1  %% Macro package 'makedoc.sty' for LaTeX2e,
2  %% copyright (C) 2009 Uwe L\"uck,
3  %%   http://www.contact-ednotes.sty.de.vu
4  %% -- author-maintained in the sense of LPPL below --
5  %% for preparing documentations from packages.
6
7  \def\fileversion{0.2} \def\filedate{2009/04/13}
8
9  %% This file can be redistributed and/or modified under
10 %% the terms of the LaTeX Project Public License; either
11 %% version 1.3a of the License, or any later version.
12 %% The latest version of this license is in
13 %%     http://www.latex-project.org/lppl.txt
14 %% We did our best to help you, but there is NO WARRANTY.
15 %%
16 %% Please report bugs, problems, and suggestions via
17 %%
18 %%   http://www.contact-ednotes.sty.de.vu
19 %%
20 \NeedsTeXFormat{LaTeX2e}[1994/12/01]
21 % 1994/12/01: \newcommand* etc.
22 \ProvidesPackage{makedoc}[\filedate\space v\fileversion\space
23                           TeX input from *.sty (UL)]
```

[2] niceverb v0.1 was too sloppy with some things, and self-documentation of makedoc.sty was difficult—its parsing and that from verbatim cannot distinguish between markup code and typeset code.

`\PackageCodeTrue` and `\PackageCodeFalse` set `\ifPackageCode` globally, so redefinition of `~` may be kept local. Note the capital `T` and `F`!

```
\newcommand*{\PackageCodeTrue} {\global\let\ifPackageCode\iftrue}
\newcommand*{\PackageCodeFalse}{\global\let\ifPackageCode\iffalse}
```

`\ifPackageCode` is used to determine whether a listing environment must be `\begun` or `\ended`. You may also want to suppress empty code lines, while empty lines should issue a `\par` break in "comment" mode.

Since `\newif` is not used, `\ifPackageCode` must be declared explicitly. Declaration of new `\if`s must be early in case they occur in code that is skipped by another `\if`...

```
\PackageCodeFalse
```

makedoc is an extension of fifinddo on which it depends.

```
\RequirePackage{fifinddo}[2009/04/13]
```

Both fifinddo and makedoc use the "underscore" as "private letter" and make it "other" at their end (functionality as with "at" and `\RequirePackage` is missing here). So after loading fifinddo, I must restore the new private letter.

```
\catcode`\_=11 %% underscore used in control words
```

2.2 \MakeDocCorrectHook

`\MakeDocCorrectHook` is predefined to leave its argument without the enclosing braces, otherwise unchanged:

```
\let\MakeDocCorrectHook\@firstofone
```

Less efficiently, the same could have been set up as

```
% \newcommand*{\MakeDocCorrectHook}[1]{\ProcessStringWith{#1}{LEAVE}}
```

according to fifinddo.

It may be *redefined* in a *configuration* file like makedoc.cfg or the makedoc script file applying to a single package file in order to, e.g., converting plain text to TEX input conforming to typographical conventions, making `$\dots$` from `$\dots$`, e.g. Replace `LEAVE` in the previous suggestion by an identifier whose job you have defined before, and use `\renewcommand` in place of `\newcommand`. See an example in makedoc.cfg.

You can *test* your own `\MakeDocCorrectHook` by

`\typeout{\MakeDocCorrectHook{`⟨*test-string*⟩`}}`

... provided (sometimes) `\MakeOther\␣` ... You can even change it using `\IfInputLine` from fifinddo in the midst of preprocessing a package documentation.

2.3 Distinguish package code from comments

Use of comment marks is a matter of personal style. Only lines starting with the sequence %%␣ are typeset in TEX quality under the present release. Lines just containing %% (without the space) are used to suppress empty code lines preceding section titles (while keeping some visual space in the package file). There is a preferable way to do this, however not in the present release ...

The parsing macros must be set up reading % and ␣ as "other" characters. Using the optional arguments for this creates difficulties that can be somewhat avoided by redefining `\PatternCodes`.

```
\renewcommand*{\PatternCodes}{\MakeOther\%\MakeOther\ }%% 2009/04/02
```

Look here: the line became too long and could not be broken. Must we really introduce new comment marks?

```
% \MakeSetupSubstringCondition{comment}[\MakeOther\%\MakeOther\ ]
```

The next line sets the "sandbox" for the detecting macro, as it is coined in the documentation of fifinddo, with "identifier" `PPScomment`.

```
\MakeSetupSubstringCondition{PPScomment}{%% }{{#1}}
```

The last argument stores the expanded input line for reference by macros called. The next line is a test whether the setup works.

```
%  \expandafter \show \csname \setup_substr_cond PPScomment\endcsname
```

Here comes the definition of the corresponding testing macro. #3 is the expanded input line from above. The `\If...` commands, `\fdInputLine`, `\fdInputLine`, and `\RemoveDummyPatternArg` are from fifinddo.

```
% \MakeSubstringConditional{comment}[\MakeOther\%\MakeOther\ ]
\MakeSubstringConditional{PPScomment}{%% }#3{%% #3 entire test string
  \IfFDinputEmpty{\OnEmptyInputLine}{%
```

The empty line test comes early to offer control with `\OnEmptyInputLine` both code and comment mode. Maybe it should always?

```
    \IfFDempty{#1}%%
              {\TreatAsComment{%
                 \RemoveDummyPatternArg\MakeDocCorrectHook{#2}}}%
              {\ifx\fdInputLine\PPstring
                 \ifPackageCode\else \WriteResult{}\fi%% 2009/04/05
                 %% <- allow paragraphs in comments
                \else \TreatAsCode{#3}\fi}}
%  \expandafter \show \csname \substr_cond PPScomment\endcsname
```

`\PPstring` stores the line suppressing empty code lines.

```
\newcommand*{\PPstring}{} \xdef\PPstring{\PercentChar\PercentChar}
```

`comment` will be a "generic" identifier to call a comment line detector. It might be predefined to issue an "undifined" error; however this release predefines it to behave like `PPScomment`.

```
\CopyFDconditionFromTo{PPScomment}{comment}
```

Alternative still to be considered:

```
% \@namedef{\setup_substr_cond comment}{%
%   \PackageError{makedoc}{Job 'comment' not defined}%
%   {Use \string\CopyFDconditionFromTo{comment}}}
```

2.4 Dealing with comments

`\TreatAsComment{`⟨*text*⟩`}` writes ⟨*text*⟩ to the documentation file. If we had "package code" (were in "code mode") so far, the listing environment is ended first.

```
\newcommand*{\TreatAsComment}[1]{%
  \ifPackageCode
    \WriteResult{\string\end{packagecode\@empty}}%
```

The `\@empty` here is a lazy trick to save self-documentation fighting verbatim's "highlight" of finding ends of listings (to be improved).

We always use `\string` to prevent macro expansion in `\write`ing in place of LaTeX's `\protect`, as long as fifinddo simply uses the primitive `\write` in place of LaTeX's `\proteced@write` ...

```
    \PackageCodeFalse
    \EveryComment
%     \_empty_code_lines_false
  \fi
  \WriteResult{#1}}
```

Here, `\EveryComment` is a macro hook for inserting material that should not appear in a listing environment.

```
\global\let\EveryComment\relax %% should be changed globally.
```

2.5 Sectioning

We provide a facility from wiki.sty that imitates the sectioning syntax used in editing *Wikipedia* pages, in a different implementation (better compatibility) and in a more general way. On Wikipedia, `==␣Definition␣==` works similar as `\section{Definition}` does with LaTeX. With the present implementation, you can type, e.g.,

`%%%%%%%%%%%%%%%%%%%%%%␣==␣Definition␣==␣%%%%%%%%%%%%%%%%%%%%%%`

to get a similar result. The number of `%` characters doesn't matter, and there can be other stuff, however: additional `=` symbols may harm. Three sectioning levels are supported, through `==`⟨*text*⟩`==`, `===`⟨*text*⟩`===`, and `====`⟨*text*⟩`====` (deepest).

There are three detector macros made for programmers. The most general one is In the following definitions, there is a single tilde to prevent `=` symbols being gobbled by the test (realized by accident). `\SectionLevelThreeParseInput`:

```
\newcommand*{\SectionLevelThreeParseInput}{%
  \expandafter \test_sec_level_iii \fdInputLine ~========&}
```

`\SectionLevelTwoParseInput`

```
\newcommand*{\SectionLevelTwoParseInput}{%
  \expandafter \test_sec_level_ii \fdInputLine    ~======&}
```

and `\SectionLevelOneParseInput`

```
\newcommand*{\SectionLevelOneParseInput}{%
  \expandafter \test_sec_level_i \fdInputLine       ~====&}
```

allow skipping deeper levels for efficiency.

In the terminology of the fifinddo documentation, the previous three commands are "sandbox builders." The following three commands are the corresponding "substring conditionals." However, fifinddo so far only deals with *single* substrings, while here we are dealing with *pairs* of substrings. We are not using general setup macros, but define the parsing macros "manually," as it is typical in many other macros in latex.ltx and other LaTeX packages. You can fool our macros easily, there is no syntax check.

```
\def\test_sec_level_iii#1====#2====#3&{%
  \IfFDempty{#2}%
            {\test_sec_level_ii #1======&}%
            {\WriteSection\mdSectionLevelThree{#2}}}
\def\test_sec_level_ii#1===#2===#3&{%
  \IfFDempty{#2}%
            {\test_sec_level_i    #1====&}%
            {\WriteSection\mdSectionLevelTwo{#2}}}
\def\test_sec_level_i#1==#2==#3&{%
  \IfFDempty{#2}%
            {\RemoveTildeArg \ProcessStringWith{#1}{comment}}%
            {\WriteSection\mdSectionLevelOne{#2}}}
```

`\ProcessStringWith` here passes the expanded `\fdInputLine` to the general comment detector.

`\WriteSection{`⟨*command*⟩`}{`⟨*text*⟩`}` replaces an input line with a line

⟨*command*⟩`{\hspace{1sp}\ignorespaces␣`⟨*text*⟩`\unskip␣}`

in the documentation file and switches into "comment mode." `\hspace{1sp}` ensures that niceverb's package name feature works. `\ignorespaces` and `\unskip` undo the spaces between title text and the `=` symbols that usually are typed for readability.

```
\newcommand*{\WriteSection}[2]{%
  \TreatAsComment{%
    ^^J#1{\string\hspace{1sp}\ignorespaces
          \MakeDocCorrectHook{#2}\unskip}^^J}}
```

We insert `\section` using `\mdSectionLevelOne` etc. which the programmer can redefine, e.g., when the documentation is part of a `\section` (or even deeper) according to the "documentation driver" file.

```
\newcommand*\mdSectionLevelOne  {\string\section}
\newcommand*\mdSectionLevelTwo  {\string\subsection}
\newcommand*\mdSectionLevelThree{\string\subsubsection}
```

This sectioning feature is not used in makedoc.sty since the *definitions* of the parsing macros fool the same macros ...

2.6 Commented code

`\TreatAsCode{`⟨*text*⟩`}` is the opposite to `\TreatAsComment{`⟨*text*⟩`}`:

```
\newcommand*{\TreatAsCode}[1]{%
  \ifPackageCode
%     \_empty_code_lines_true
  \else
    \WriteResult{\string\begin{packagecode}}%
    \PackageCodeTrue
  \fi
  \WriteResult{#1}%
%   \WriteResult{\maybe_result_empty_line #1}%
%   \let\maybe_result_empty_line\empty
}
```

2.7 Dealing with empty input lines

`\OnEmptyInputLine` is a default setting (or hook) for what to do with empty lines in the input file. The default is to insert an empty line in the output file:

```
\newcommand*{\OnEmptyInputLine}{\WriteResult{}}
```

`\NoEmptyCodeLines` changes the setting to suppressing empty code lines, while in "comment mode" an empty input line *does* insert an empty line, for starting a new paragraph:

```
\newcommand*{\NoEmptyCodeLines}{%% suppress empty code lines
  \renewcommand*{\OnEmptyInputLine}{%
    \ifPackageCode \else \WriteResult{}\fi}}
```

There is a better policy—didn't work so far ...

2.8 Bundling typical things: script commands

First practical experiences suggest the following shorthands. They should simplify matters so much that the makedoc script for a single package really should need about five lines only, and even *they* should be so simple that you should hardly spend a minute about them.

`\LaTeXresultFile{`⟨*output*⟩`}` saves you the extra line for inserting the `\ProvidesFile` line ... no, actually it is makedoc that wants to be mentioned with `\ProvidesFile` ... (otherwise copied from fifinddo) ...

```
\newcommand*{\LaTeXresultFile}[1]{%
  \ResultFile{#1}%% \WriteProvides}
  \WriteResult{%
    \string\ProvidesFile{\result_file_name}%
      [\the\year/\two@digits\month/\two@digits\day\space
       automatically generated with makedoc.sty]}}%
```

`\MakeDoc{`⟨*input*⟩`}` preprocesses ⟨*input*⟩ to render input for LaTeX, considering what is typical for a LaTeX package as the ⟨*input*⟩ one here:

```
\newcommand*{\MakeDoc}[1]{%
```

In case of a "header" (see below) we change into "code mode":

```
    \ifnum\header_lines>\z@
          \WriteResult{\string\begin{packagecode}}%
          \PackageCodeTrue %% TODO both lines makedoc command!?
                           %%      2009/04/08
    \else \PackageCodeFalse \fi
```

The loop follows. There is a placeholder `\makd_doc_line_body` that is predefined below and can be changed while processing the ⟨*input*⟩ file.

```
    \ProcessFileWith{#1}{%
      \CountInputLines %% stepping line counter is standard
      \make_doc_line_body
      \process_line_message}%
```

Currently the "VERSION HISTORY" is typeset verbatim (for "tabbing"), we then must leave "code mode" here:

```
    \ifPackageCode
      \WriteResult{\string\end{packagecode\@empty}}%% self-doc-trick
      \PackageCodeFalse %% TODO both lines makedoc command!? 2009/04/08
    \fi
```

When the ⟨*input*⟩ file has been processed, certain default settings might be restored—in case another ⟨*input*⟩ file is processed for the same documentation document:

```
%   \HeaderLines{0}%
%   \MainDocParser{\SectionLevelThreeParseInput}%% TODO!? 2009/04/08
}
```

2.8.1 File preamble

A LaTeX package typically has a "header" or "preamble" (automatically inserted by docstrip) with very scarce information on which file it is and what it provides, and with much more Legalese. Typesetting it in TeX quality may be more misleading than typesetting it *verbatim*. So we typeset it *verbatim*. Now: where does the "header" end? `\NeedsTeXFormat` might be considered the border.—Yet it seems to be more simple and reliable just to act in terms of the *number of lines* that the header should be long. This length ⟨*how-many-lines*⟩ is declared by `\HeaderLines{`⟨*how-many-lines*⟩`}`:

```
\newcommand*{\HeaderLines}{\def\header_lines}
\HeaderLines{0}
```

So the default is that there aren't any header lines, unless another `\HeaderLines` is issued before some `\MakeDoc`. The way input is parsed after the "header" is set by `\MainDocParser{`⟨*parsing-command*⟩`}`.

```
\newcommand*{\MainDocParser}{\def\main_doc_parser}
```

`\SectionLevelThreeParseInput` from section 2.5 is the default, two alternatives are defined there, another one is `\ProcessInputWith{comment}` from fifinddo.

```
\MainDocParser{\SectionLevelThreeParseInput}
```

Here is how `\HeaderLines` and `\MainDocParser` act:

```
\def\make_doc_line_body{%
  \IfInputLine{>\header_lines}%
              {\let\make_doc_line_body\main_doc_parser
                   \make_doc_line_body}%    %% switch to deciding
              {\TreatAsCode{\fdInputLine}}} %% header verbatim
```

`\ProcessLineMessage{`⟨*command*⟩`}` is designed to define a screen (or log) message ⟨*command*⟩. `\ProcessLineMessage{\message{.}}` has a result like with docstrip. You just get one dot on screen per input line as a simple confirmation that the program is not hung up. However, the message may slow down a run considerably (if so, choose `\ProcessLineMessage{}` in the script). But it is better for beginner users of the package, so made default.

```
\newcommand*{\ProcessLineMessage}{\def\process_line_message}
% % \ProcessLineMessage{} %% no, still more efficient:
% \let\process_line_message\relax
\ProcessLineMessage{\message{.}}
```

2.9 Leave the package

```
\catcode`\_=8   %% restores underscore use for subscripts

\endinput
```

2.10 VERSION HISTORY

```
v0.1   2009/04/03   very first version, tested on morgan.sty
v0.2   2009/04/05   \OnEmptyInputLine, \NoEmptyCodeLines
                    comment -> PPScomment, \IfFDinputEmpty,
                    \EveryComment, \PPstring may be par break
       2009/04/08   \InputString -> \fdInputLine,
                    \section -> \subsection; documentation!
       2009/04/08f. \MakeDoc
       2009/04/12   ``line too long'' w/o redefining \PatternCodes;
                    \MakeDocCorrectHook
       2009/04/13   tilde with sectioning

```

The previous empty code line is the one TEX insists to add at every end of a file it writes.

3 Examples: documentation of fifinddo

3.1 makedoc.cfg

fifinddo.pdf and makedoc.pdf were typeset with the following configuration file makedoc.cfg:

```
\ProvidesFile{makedoc.cfg}[2009/04/15
                           local settings for ‘makedoc.sty’ etc.]

\RequirePackage{moreverb}
\newenvironment{packagecode}
  {\PackageCode}
  {\endPackageCode}
\gdef\PackageCode{%
  \small
%% Get rid of 'niceverb' stuff:
  % \MakeOther\‘\MakeOther\’%% probably OK with moreverb
  \MakeOther<\MakeOther\|%
    %% <- TODO should be 'niceverb' command 2009/04/08
  \listing{1}}
\gdef\endPackageCode{%
  \endlisting
  \global\def\PackageCode{%
    \small
      % \MakeOther\‘\MakeOther\’%% probably OK with moreverb
      \MakeOther<\MakeOther\|% niceverb
        \listingcont}%
  \global\let\endPackageCode\endlistingcont}
\renewcommand*{\listinglabel}[1]{%
  \llap{\scriptsize\rmfamily\the#1}\hskip\listingoffset\relax}

\RequirePackage{niceverb}[2009/04/11] %% (' and ''; ‘etc.’\@
\DeclareRobustCommand{\cs}[1]{\texttt{\char‘\\#1}}
  %% <- ‘&\@tempa’ and ‘&\_tempa’ fail 2009/04/14
\RequirePackage{color}

%% TODO rather in 'niceverb' 2009/04/06:
\CatCode\|\active
\newcommand*{\CmdBox}{%
  \ifvmode \pagebreak[1]\fi %% TODO!? 2009/04/06
  \begingroup \let\do\MakeOther \dospecials \tt \TypesetCmdBox}
\def\TypesetCmdBox#1|{%% redefine for changing design
```

```
%     \fboxrule=.6pt \fboxsep=-\fboxrule
%     \fcolorbox[cmyk]{0,0,0,1}{.1,0,.2,.1}{%
%       \kern2pt\strut\CmdSyntaxVerb#1\kern2pt}}}
%       \kern2pt\strut#1\kern2pt}%% \dospecials version
%     \fboxrule=.6pt \fboxsep=.2pt
%     \fbox{%
%       \fboxrule=0pt \fboxsep=-1pt
      \fboxrule=0pt \fboxsep=0pt
%       \kern.2pt
      \colorbox[cmyk]{.1,0,.2,.05}{%
        \kern1.6pt\strut#1\kern1.6pt}%
%       \kern.2pt
%     }%
  \endgroup
  \nopagebreak[3]} %% TODO!? 2009/04/06
\let|\CmdBox

\pagestyle{headings}

\endinput
```

3.2 mkfddoc.tex

fifinddo.pdf was typeset with the following makedoc script file mkfddoc.tex:

```
\ProvidesFile{mkfddoc.tex}[2009/04/15
                           prepare typesetting fifinddo.sty]

  \begingroup %% generate fifinddo.doc
\RequirePackage{makedoc}[2009/04/13]
\input{mdcorr.cfg}

% \NoEmptyCodeLines %% TODO
\ProcessLineMessage{}
\LaTeXresultFile{fifinddo.doc}
\HeaderLines{23}
\MainDocParser{%
%   \IfInputLine{=33}{\tracingmacros=1 }{}
%   \IfInputLine{=35}{\tracingmacros=0 }{}
  \SectionLevelTwoParseInput}
%  \tracingmacros=1
\MakeDoc{fifinddo.sty}
\CloseResultFile
  \endgroup %% fifinddo.doc ready
\endinput
```

3.3 mdcorr.cfg

fifinddo.pdf and makedoc.pdf were typeset with the following typographical corrections in mdcorr.cfg:

```
\ProvidesFile{mdcorr.cfg}[2009/04/15
                      local typographical corrections with makedoc.sty]
%% Also demonstrates 'niceverb.sty'.
\renewcommand*{\PatternCodes}{\MakeOther\\\MakeOther\ }
\renewcommand*{\MakeDocCorrectHook}[1]{\ProcessStringWith{#1}{dots}}
%% |\MakeExpandableAllReplacer{<id>}{<find>}{<replace>}{<id-next>}|%
%% \footnote{Yes,
%% &\MakeExpandableAllReplacer{<id>}{<find>}{<replace>}{<id-next>}.}
\MakeExpandableAllReplacer{dots}{...}{$\dots$}{cf}
\MakeExpandableAllReplacer{cf}{cf.}{cf.\ }{etc}
\MakeExpandableAllReplacer{etc}{etc. }{etc.\ }{LEAVE}
%% So you can keep inter-sentence space after 'etc.'
%% by a code line break
\renewcommand*{\PatternCodes}{\fdPatternCodes}
%% ... restores 'fifinddo' default.
```

This code also exemplifies the syntax niceverb provides for writing about LaTeX macros. It is typeset here with makedoc.sty again and then looks thus:

```
\ProvidesFile{mdcorr.cfg}[2009/04/15
                    local typographical corrections with makedoc.sty]
```

Also demonstrates niceverb.sty.

```
\renewcommand*{\PatternCodes}{\MakeOther\\\MakeOther\ }
\renewcommand*{\MakeDocCorrectHook}[1]{\ProcessStringWith{#1}{dots}}
```

`\MakeExpandableAllReplacer{`⟨*id*⟩`}{`⟨*find*⟩`}{`⟨*replace*⟩`}{`⟨*id-next*⟩`}`[3]

```
\MakeExpandableAllReplacer{dots}{...}{$\dots$}{cf}
\MakeExpandableAllReplacer{cf}{cf.}{cf.\ }{etc}
\MakeExpandableAllReplacer{etc}{etc. }{etc.\ }{LEAVE}
```

So you can keep inter-sentence space after `etc.` by a code line break

```
\renewcommand*{\PatternCodes}{\fdPatternCodes}
```

... restores fifinddo default.

[3] Yes, `\MakeExpandableAllReplacer{`⟨*id*⟩`}{`⟨*find*⟩`}{`⟨*replace*⟩`}{`⟨*id-next*⟩`}`.