fifinddo

—

Filtering TEX(t) Files by TEX*

Uwe Lück†

March 17, 2012

FIDO, FIND!
or:
FIND FIDO!
oder:
FIFI, SUCH!

Abstract

fifinddo starts implementing parsing of plain text or TEX files using TEX, generalizing the philosophy behind docstrip, based on how TEX reads macro arguments. Rather than typsetting the edited input stream immediately, results are written to another file, in the first instance as input for TEX. Rather than presenting a "complete study" of a computer-scientific idea, it aims at practical applications. The main one at present is makedoc which removes certain comment marks from package files and inserts listing commands. Parsing macros are not defined anew at every input chunk, but once before a file is processed. This also allows for *expandable* sequences of replacements, e.g., with `txt` → TEX functionality. The method of testing for substrings is carefully discussed, revealing an earlier mistake (then) shared with substr.sty and LATEX's internal `\in@`.

Keywords: text filtering, macro programming, .txt to .tex enhancement

Contents

*This file describes version v0.51 of fifinddo.sty as of 2012/03/17.

†http://contact-ednotes.sty.de.vu

1 Introduction: The Gnome of the Aim

1.1 Parsing by TEX—are you mad?

The package name fifinddo is a `\listfiles`-compatible abbreviation of 'file-find-do'[1] (or think of 'if found do'). fifinddo implements (or aims at) general parsing (extracting, replacing [converting], expanding, ...) using TEX where texhax posters strongly urge to use sed, awk, or Perl. fifinddo's opposed rationales are:

- It works instantly on any TEX installation. (*Restrictions:* Some TEX versions `\write` certain hex codes for certain characters, cf. TEXbook p. 45, I have seen this with PCTEX. However, some applications of fifinddo are nothing but technical steps where you will read the result files rarely anyway.
- You can apply and customize it like any TEX macros, knowing just TEX (or even only the documentation of some user-friendly extension of fifinddo), without the need of learning any additional script language.
- The syntax of usual utilities (e.g., "wildcards") is sometimes difficult with *TEX* files with all their backslashes, square brackets, stars, question marks ...

At least the first item is just the philosophy of the docstrip program, standard for installing TEX packages; and while I am typing this, I find at least 14 other similar packages in Jürgen Fenn's *Topic Index* of the *TEX Catalogue:*

`http://mirror.ctan.org/help/Catalogue/bytopic.html#parsingfiles`

(Some of them may have been *reactance* to texhax and other postings urging not to try something like this; some seem just to be celebrations of the power of TEX—yes, celebrate!)

Actually, TEX's mechanism of collecting macro arguments is hard-wired parsing at quite a high level. LATEX hides this from "simple-minded" users by a convention *not* to use that full power of TEX for *end-user macros*. *Internally*, LATEX *does* use it in reading lists of options and file dates as well as to implement certain `FOR`- and `WHILE`-like loop programming structures. LATEX's `\in@`/`\ifin@` construction is an implementation of a "⟨*string1*⟩ occurs in ⟨*string2*⟩" test. More packages seem to use this idea for extracting file informations, like texshade.[2]

However, such packages don't make much ado about parsing, there seems to be no general setup mechanism as are presented by fifinddo. Indeed, tayloring parsing macros to specific applications may often be more efficient than a general approach.

[1] 'file' possibly for "searching TEX(t) files" (I don't remember my thoughts!), while there were requests for doing replacements on LATEX *environments* on texhax. However, the package might be enhanced in this direction ... so the name may be wrong ... but now I like it so much ... Or the reason was that results are written to a *separate file*, not typeset immediately.—Let me also mention that *'Fifi'* (as the package name starts) is a kind of German equivalent to the "English" *'Fido'*, or may have been.

[2] `http://ctan.org/pkg/texshade`

1.2 Useful for ...

My main application of fifinddo at present is typesetting documentations of packages using makedoc which removes certain percent marks and inserts listing commands, so you edit a package file with as little documentation markup as possible. This may be extended to other kinds of documents as an alternative to easylatex or wiki (the approach of which is dangerous and incompatible with certain other things).

I have used a similar own package txtproc successfully, where more features were implemented for practical purposes than are here so far, yet I don't like its implementation, want to improve it here. This package also *created batch files*, e.g., to remove temporary files. This could be used for package handling: typset the documentation at the desired place in the tree, write the packages to another, write a batch file to remove files that are not needed any more after installation (cf. make).

I used txtproc also for *large-scale substitutions* (it had been decided to change the orthography in a part of a book). Other large-scale substitutions may be:

- inserting `\index` commands;
- inserting (soft) hyphenation commands near accents;
- manual umlaut-conversion.[3]
- typographical (or even orthographical) corrections (same mistake many times on each of hundreds of pages). You may turn `...` into `$\dots$` and `etc.` into `etc.\` etc.[4] This could replace packages like easylatex,[5] txt2latex,[6] txt2tex[7] in a customizable way, using, e.g., the "correct" hook from makedoc.sty as exemplified in mdoccorr.cfg (see examples section of makedoc.pdf). You should find fdtxttex.tpl, a fifinddo script to try or apply `\MakeDocCorrectHook` from mdoccorr.cfg, as well as fdtxttex.tex that runs a dialogue for the same purpose if you can manage to run it (WinShell?). You can then try to create your own `\MakeDocCorrectHook`. Section 6 provides setup for macros of this kind.
- as to easylatex again, *lists* could be detected and transformed into LaTeX list commands. This could re-implement the lists functionality of wiki.sty that is somewhat dangerous.
- introduce your own *shorthands* to be expanded not as TeX macros, but by text substitution;

[3] If you know the "names" of the encodings, Heiko Oberdiek's stringenc may be preferable.

[4] But what when a new sentence is starting indeed? Well, `cf.` is an easier example.—`etc.` even showed a problem in niceverb. mdoccorr.cfg replaces `etc.` only, so you can keep the extra space by a code line break.

[5] `http://ctan.org/pkg/easylatex`

[6] `http://ctan.org/pkg/txt2latex`

[7] `http://ctan.org/pkg/txt2tex`

In certain cases, insertions deteriorate readability, hyphenation corrections even make text search difficult. It is therefore suggested to

1. keep editing the file without the insertions,
2. run the script (commands based on fifinddo) for insertions in the preamble of the main file ("`\jobname.tex`", maybe `\input` the script file) and
3. `\input` the result file within the `document` environment.

In general, differences to "manual" replacing by the substitution function of your *text editor* is that

- you first keep the original version,
- you can check the resulting file before you replace the original file by it,
- you can store the replacement script in order to check for mistakes at a later stage of your work,
- you can do *all* the replacements in *one run* (by *one* script to check for mistakes),
- you can store replacement scripts for future applications, so you needn't type the patterns and replacement strings anew.

1.2.1 Comparisons

It should be noted (perhaps here) that the present approach to parsing is a quite *simple* one and in this respect much different to the string handling mechanisms of stringstrings,[8] ted,[9] xstrings[10] (as I understand them, perhaps also coolstr[11]) which are *much more powerful* than what is offered here—but perhaps slow and for practical applications possibly replaceable by the present approach. *Expandable replacement* seems not to exist outside fifinddo (2009/04/13).

Much is missing, I know.[12] I am just implementing what I actually need and what could show that this approach is worth being pursued.

1.3 For insiders

Warning: You may (at least at the present state of the work) have little success with this package, if you don't know about TEX's category codes and how TEX macros are defined. The package rather provides tools for package writers. You may, however, be able to run other packages which just load fifinddo as required background.

[8] `http://ctan.org/pkg/stringstrings`
[9] `http://ctan.org/pkg/ted`
[10] `http://ctan.org/pkg/xstrings`
[11] `http://ctan.org/pkg/coolstr`
[12] There is more in my badly implemented txtproc.sty.

That fifinddo acts on "TEX(t)" files or so means that (at present) I think of applications on "plain text" files which will usually be TEX input files. "At present" they are read without "special characters," so essentially category codes of input characters are either 11 ("letter") or 12 ("other"). This way some things are easier than with usual TEX applications:

1. You can "look into" curly braces and "behind" comment characters.

2. There are exact or safe tests especially of *empty macro arguments* that are "expandable," i.e., they are "robust," don't need assignments, can be executed in `\write`ing and in `\edef` definitions. "Usually," the safe way to test emptiness is storing a macro argument as a macro, say `\tempo`, in order to test `\ifx\tempo\empty` where `\empty` has been defined by `\def\empty{}` in the format. But this requires some `\def\tempo{#`⟨*n*⟩`}` which breaks in "mere expanding" (TEX *evaluates* `\tempo` instead of defining it). An *expandable* test on emptiness is, e.g. `\ifx$#`⟨*n*⟩`$`, where we hope that it becomes `\iftrue` just if macro argument `#`⟨*n*⟩ is empty indeed. However, "usually" it may *also* become `\iftrue` when `#`⟨*n*⟩ starts with `$`—if the latter has category code 3 ("math shift"). But fifinddo does not assign category code 3 to any character from the input file! Therefore `\ifx$#`⟨*n*⟩`$` is `\iftrue` *exactly* if `#`⟨*n*⟩ is empty.

3. You can avoid interference with packages that are needed for typesetting. You can do the "preprocessing" in one run with typesetting, but you should do the preprocessing before you load packages needed for typesetting. One may even try to keep the macros and settings for preprocessing local to a group.

The essential approach of fifinddo to looking for single strings is described in some detail in section 4.

The implementation of fifinddo is as follows. User commands are specially highlighted (boxed/coloured), together with their syntax description.

2 Preliminaries

2.1 Head of file (Legalese)

```
%% Macro package 'fifinddo.sty' for LaTeX2e,    %% FIDO, FIND!
%% copyright (C) 2009-2012 Uwe L\"uck,
%%   http://www.contact-ednotes.sty.de.vu
%% -- author-maintained in the sense of LPPL below --
%% for processing tex(t) files
%% (checking, filtering, converting, substituting, expanding, ...)

\def\fileversion{0.51} \def\filedate{2012/03/17}

%% This file can be redistributed and/or modified under
%% the terms of the LaTeX Project Public License; either
```

```
%% version 1.3c of the License, or any later version.
%% The latest version of this license is in
%%
%%     http://www.latex-project.org/lppl.txt
%%
%% We did our best to help you, but there is NO WARRANTY.
%% Please report bugs, problems, and suggestions via
%%
%%   http://www.contact-ednotes.sty.de.vu
%%
%% For the full documentation, look for ‘fifinddo.pdf’.
%% Its source starts in ‘fifinddo.tex’.
```

2.2 Format and package version

```
\NeedsTeXFormat{LaTeX2e}[1994/12/01]
% 1994/12/01: \newcommand* etc.
\ProvidesPackage{fifinddo}[\filedate\space v\fileversion\space
                           filtering TeX(t) files by TeX (UL)]
```

2.3 Category codes

We use the “underscore” as “compound identifier.”

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

`\MakeOther` is a synonym for `\@makeother`, needed for matching special characters from the input file. It is exemplified by `\fdPatternCodes` which is the default of `\PatternCodes`. The latter is used in setup macros for reading patterns. We offer `\SetPatternCodes{`⟨*commands*⟩`}` (redefining `\PatternCodes`) and `\ResetPatternCodes` (for returning to `\fdPatternCodes`) so setup scripts such as mdoccorr.cfg have shorter lines.

```
\@ifundefined{MakeOther}{\let\MakeOther\@makeother}{}
\newcommand*{\fdPatternCodes}{\MakeOther\&\MakeOther\$}
\newcommand*{\SetPatternCodes}{\def\PatternCodes}
\newcommand*{\ResetPatternCodes}{\let\PatternCodes\fdPatternCodes}
\newcommand*{\PatternCodes}{} \ResetPatternCodes
  %% TODO adding/removing; ‘*’ may be wrong 2010/03/29
```

It would be bad to have `\MakeOther\%` and `\MakeOther\␣` here in that this may have unexpected, weird effects with arguments of setup macros. (With `\MakeOther\␣` you must not indent within a setup command, and if you add `\MakeOther\%` the setup command must stay in one line.) Therefore neither `\dospecials` nor `\@sanitize` are used. Curly braces remain untouched as default delimiters in setup macros. For matching them, you must use `\MakeOther\{` and `\MakeOther\}` in your `\PatternCodes`, or `\Delimiters` to introduce new ones at the same time, e.g., `\Delimiters\[\]`:

```
\newcommand*{\Delimiters}[2]{%
  \MakeOther\{\MakeOther\}\catcode‘#1\@ne \catcode‘#2=\tw@}
```

For replacing strings or for defining other strings of "other" characters by `\edef`, you can use some LaTeX constructs—here are copies `\PercentChar` and `\BackslashChar` of them (do you need more?):

```
\newcommand*{\PercentChar}{}   \let\PercentChar\@percentchar
\newcommand*{\BackslashChar}{} \let\BackslashChar\@backslashchar
```

`\BasicNormalCatcodes` restores Plain TeX's **macro parsing** and comment character:

```
\newcommand*{\BasicNormalCatCodes}{%
  \catcode`\\\z@ \Delimiters\{\}%
%   \restorecr !?
  \catcode`\ =10 \catcode`\%=14}
```

However, reading files *line by line* makes parsing of macro parameters somewhat difficult when the parameter code spans code lines. A line must not end with a curly brace when a macro requires another parameter; instead, it must contain the curly left brace for the next parameter.

`\MakeActiveDef\`⟨*char*⟩`{`⟨*expand-to*⟩`}` makes ⟨*char*⟩ an active character expanding to ⟨*expand-to*⟩

```
\newcommand*{\MakeActiveDef}[1]{%
    \catcode`#1\active
    \begingroup
        \lccode`\~`#1\relax \lowercase{\endgroup \def~}}
```

(cf. `\@sverb`/`\do@noligs` in LaTeX's doc.sty). This even allows defining active characters with parameters (suggested by Heiko Oberdiek LATEX-LIST 2010/09/18, may be nice for UTF-8). The macro has been used for conversion of text encodings.

3 File handling

Peter Wilson's newfile provides more powerful file handling.

3.1 Opening, Writing to, Closing Output

```
\newwrite\result_file %% or write to \@mainaux!? TODO
```

`\ResultFile{`⟨*output*⟩`}` opens (and empties) a file ⟨*output*⟩ to be written into.

```
\newcommand*{\ResultFile}[1]{%
  \def\result_file_name{#1}%
  \typeout{`fifinddo' generating `\result_file_name'}%  %% 2011/10/23
  \immediate\openout\result_file=#1}
```

`\WriteResult{`⟨*balanced*⟩`}` writes a ⟨*balanced*⟩ line into ⟨*output*⟩ (or more lines with `^^J`).

```
\newcommand*{\WriteResult}[1]{%
  \immediate\write\result_file{#1}}
```

`\WriteProvides` writes a `\ProvidesFile` command to the opened ⟨*output*⟩ file. This should be used when ⟨*output*⟩ is made as LaTeX 2_ε input.

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

`\CloseResultFile` closes ⟨*output*⟩:

```
\newcommand*{\CloseResultFile}{%
    \immediate\closeout\result_file
    \typeout{‘fifinddo’                    %% 2011/10/23
             \space\space\space            %% 2011/10/26
             closing ‘\result_file_name’}}
```

3.2 Processing Input

`\ProcessFileWith[`⟨*changes*⟩`]{`⟨*input*⟩`}{`⟨*loop-body*⟩`}` opens a file ⟨*input*⟩ and runs a loop on its lines the main body of which is ⟨*loop-body*⟩. When the ⟨*loop*⟩ starts, a new line of ⟨*input*⟩ is stored as macro `\fdInputLine`. The optional argument ⟨*changes*⟩ may change category codes used in reading ⟨*input*⟩. It may be useful to read macros with arguments and active characters expanding in writing to the output file. Even these expansions may be defined here (local to the group like everything else happening here, unless ...). Macros `\BasicNormalCatcodes` and `\MakeActiveDef` have been created for this purpose (see previous section TODO). (It may be better to store these ⟨*changes*⟩ in another macro ⟨*macro*⟩ and to call `\ProcessFileWith[`⟨*macro*⟩`]{`⟨*input*⟩`}{`⟨*loop-body*⟩`}`). More possible uses of some usual TeX category codes may be (some of)

- avoiding matching substrings of control words,
- skipping blank spaces as TeX does it usually,
- catching *balanced* input pieces,
- ignoring comments,
- ignoring certain characters.

```
\newcommand*{\ProcessFileWith}[3][]{%
```

v0.5: Variant of LaTeX's `\IfFileExists`—failed so far because I had omitted the blank space:

```
    \openin\@inputcheck#2 % space essential!  2011/11/19
    \ifeof\@inputcheck
      \PackageError{fifinddo}{File `#2' not here}%
                             {Mistyped?}%
    \else
      \typeout{`fifinddo' processing `#2'}%%  2010/04/15
```

... moves into conditional with v0.5. Resetting line counter:

```
      \global\c@fdInputLine=\z@
      \begingroup
        \MakeOther\{\MakeOther\}\@sanitize
```

... switching into "plain text mode"; from `docstrip.tex`:

```
        % \MakeOther\^^A\MakeOther\^^K%% irrelevant, not LaTeX
```

← cf. TEXbook pp. 43ff., 368ff., "extended keyboards", up-/downarrow; → "math specials", cf. "space specials"

```
        \endlinechar\m@ne
        \MakeOther\^^I% ASCII horizontal tab -- guessed!? ^^L!?
```

With v0.31, we support non-ASCII:

```
        \count@=128
        \loop
        \ifnum\count@<\@cclvi
          \catcode\count@=12
          \advance\count@\@ne
        \repeat
        #1%
        \loop \ifeof\@inputcheck \else
          \read\@inputcheck to \fdInputLine
          \ignorespaces #3%
```

v0.42 supports `\IfFDpreviousInputEmpty`, cf. section 5.1:

```
          \expandafter \let
            \expandafter \IfFDpreviousInputEmpty
              \ifx\fdInputLine\@empty \@firstoftwo
                \else                 \@secondoftwo \fi
        \repeat
      \endgroup
    \fi
    \closein\@inputcheck
```

Added for v0.5, cf. Sec. 3.3

```
    \ifFinalInputFile
      \CloseResultFile \FinalInputFilefalse
    \fi
  }
```

`\CopyFile[⟨changes⟩]{⟨file⟩}` is an application of `\ProcessFileWith` that "copies" the content of file ⟨*file*⟩ into the file specified by `\ResultFile`. However, optional ⟨*changes*⟩ allows some "modifications" while "copying"—especially, conversion of text encodings by active characters and expanding macros for generating HTML or other code. The "starred" variant `\CopyFile*` copies one empty line only when one empty line in the input file is followed by more of them.

```
\newcommand*{\CopyFile}{%
    \@ifstar{\let\FD@copy@style\FD@compress@voids \FD@copyfile}%
            {\let\FD@copy@style\CopyLine          \FD@copyfile}}
\newcommand*{\FD@copyfile}[2][]{%
    \ProcessFileWith[#1]{#2}{\FD@copy@style\message{.}}}
```

You should find a file `copyfile.tex` providing a dialogue for "compressing" files this way. As soon as you have a useful conversion mapping file (defining `\TextCodes`), you can also use it for text encoding conversions.

`\CopyLine`:

```
\newcommand*{\CopyLine}{\WriteResult\fdInputLine}
```

(... added `\space` without success with macro arguments 2010/04/26 — `\BlogCodes` has used a better solution later).

```
\newcommand*{\FD@compress@voids}{%
    \IfFDinputEmpty{\IfFDpreviousInputEmpty\relax\CopyLine}%
                   \CopyLine}
```

3.3 A Combining Shorthand

Before v0.5, the general idea was that some number of **input** files would be combined for creating a single **output** file (e.g., as opposed to verbatimcopy, regarding `\CopyFile`). According to experience, however, a **single** input file typically suffices for generating an output file. Then `\CloseResultFile` (Sec. 3.1) may be called "implicitly". The flag `\ifFinalInputFile` will control (and generalize!) this (in `\ProcessFileWith`, Sec. 3.2), and will be reset if changed (thinking of generating another file in the same TEX run).

So `\ProcessFinalFileWith[⟨changes⟩]{⟨input⟩}{⟨loop-body⟩}` works as with `\ProcessFileWith` except that `\CloseResultFile` is issued after processing, so you can omit the latter.

```
\newif\ifFinalInputFile
\newcommand*{\ProcessFinalFileWith}{%
    \FinalInputFiletrue \ProcessFileWith}
```

4 Basic handling of substring conditionals

4.1 "Substring Theory"

I wished I could study string theory,
but I only could study substring theory.

A TEX macro, say, `\find` whose parameter text (cf. TEXbook p. 203) starts with `#1`⟨*pattern*⟩`#2&` stops TEX with an error if it does not find ⟨*pattern*⟩ and then `&`. Otherwise we have a situation `\find`⟨*split1*⟩⟨*pattern*⟩⟨*split2*⟩`&`, and `\find` reads ⟨*split1*⟩ as `#1` and ⟨*split2*⟩ as `#2`. An important point to note is that ⟨*split1*⟩ will not contain ⟨*pattern*⟩, but possibly ⟨*pattern*⟩ has more occurrences in ⟨*split2*⟩. In this sense, `\find` uses the *first* occurrence of ⟨*pattern*⟩ it finds in order to delimit `#1`. Finding the *last* occurrence of ⟨*pattern*⟩ therefore needs a special idea.

In order to use `\find` for a test whether ⟨*pattern*⟩ is in ⟨*target*⟩, we build a "sandbox" `\find`⟨*sand*⟩`&`, where ⟨*sand*⟩ contains ⟨*target*⟩ *and additionally* ⟨*pattern*⟩—as a "dummy;" so `&` delimits the search and `\find` finds ⟨*pattern*⟩ either in ⟨*target*⟩ or somewhere else before `&`.

Consider the simple sandbox `\find`⟨*target*⟩⟨*pattern*⟩`&`. We can test `#1` and `#2` on being empty by `\ifx$#1$` and `\ifx$#2$`. If `#2` is empty, ⟨*pattern*⟩ is *not* in ⟨*target*⟩. If `#1` is empty at the same time, ⟨*target*⟩ is empty. If `#1` is empty and `#2` is not, ⟨*pattern*⟩ *starts* ⟨*target*⟩![13] This can be used to implement Wikipedia-like lists and to distinguish package code from comments in makedoc.[14]

If `#2` is *not* empty, ⟨*pattern*⟩ occurs in ⟨*target*⟩—or this once was *thought*, some time in developping the present package, as well as in the version of substr.sty marked `2005-11-29`,[15] try (if that version still is installed)[16]

`\IfSubStringInString{`⟨*str1*⟩⟨*str2*⟩⟨*str1*⟩`}{`⟨*str1*⟩⟨*str2*⟩`}{YES}{NO}`

which works *verbatim* as well as considering ⟨*str1*⟩ and ⟨*str2*⟩ *placeholders*, e.g., for

`\IfSubStringInString{day␣after␣day}{day␣after␣}{YES}{NO}`[17]
`\IfSubStringInString{AMSTERDAM}{AMSTERD}{YES}{NO}`
`\IfSubStringInString{TORONTO}{TORON}{YES}{NO}`
`\IfSubStringInString{bonbon}{bon}{YES}{NO}`[18]
`\IfSubStringInString{bonobo}{bono}{YES}{NO}` (an ape)

[13] This is just as wrong as the next claim.

[14] Due to the special substrings to test in this application, this is true although the surrounding claims are wrong.

[15] `http://ctan.org/pkg/substr`. substr does not change category codes as fifinddo does and uses `\@nil` as delimiter instead of our `&`.

[16] The "feature" has been fixed with v1.2 as of 2009/10/20 of substr.sty.

[17] Likewise `t\^ete-\‘a-t\^ete` ...

[18] Polynesian: `aku aku`, `rongorongo`, `wiki wiki` ...

or `\IfSubStringInString{ionization}{ionizat}{YES}{NO}`.[19] Same with LaTeX's internal `\in@`:[20]

```
\makeatletter␣\in@{bonbon}{bon}\ifin@␣YES\else␣NO\fi␣\makeatother
```

In general, the previous approach *fails if and exactly if* ⟨*pattern*⟩ has a *period* p—less than its length—in the sense of that the pth token to the right or left of each token in ⟨*pattern*⟩ is the *same* token. `AMSTERDAM` has a period 7, `day␣after␣day` 10, `bonbon` 3, `bonobo` 4. There is a counterexample ⟨*target*⟩ of length p iff ⟨*pattern*⟩ has period p, namely the first substring of ⟨*pattern*⟩ having length p. If the length of ⟨*pattern*⟩ exceeds a multiple mp of its period, the first mp tokens of ⟨*pattern*⟩ form a counterexample ⟨*target*⟩.

Therefore, a sandbox must have something between ⟨*target*⟩ and ⟨*pattern*⟩.[21] We choose `\find`⟨*target*⟩`~`⟨*pattern*⟩`$&` as standard. The `$` will be used as an argument delimiter to get rid of the dummy ⟨*pattern*⟩ in ⟨*split2*⟩, as well as to decide whether the match was in ⟨*target*⟩ or in the dummy part of the sandbox. The `$` can be replaced by another tilde `~` in order to test whether ⟨*target*⟩ *ends* on a ⟨*pattern*⟩, defining a macro like `\findatend` whose parameter text starts with `#1`⟨*pattern*⟩`~#2&`.

4.2 Plan for proceeding

When we check a file for several patterns, we seem to need *two* macros for each pattern: one that has the pattern in its parameter text and one that stores the pattern for building the sandbox.[22] We use a separate *"name space"* for each of both kinds. The parsing macro and the macro building the sandbox will have a common *"identifier"* by which the user or programmer calls them. Actually, she will usually (first) call the sandbox box builder. The sandbox builder calls the parsing macro. When *all* occurrences of a pattern in the target are looked for, the parser may call itself.

Actually, the parsing macro will execute certain actions depending on what it finds in the sandbox, so we call it a *"substring conditional"*. It may read additional arguments after the sandbox that store information gathered before. This is especially useful for designing *"expandable"* chains (sequences) of conditionals where macros cannot store information in macros. The macro setting up the sandbox will initialize such extra arguments at the same time.

[19] Read substr.sty or try "normal" things to convince yourself that the syntax indeed is `\IfSubStringInString{`⟨*pattern*⟩`}{`⟨*target*⟩`}{`⟨*yes*⟩`}{`⟨*no*⟩`}`.

[20] `\in@` has been fixed after my warning on Heiko Oberdiek's proposal—at least in the repository.—On 2009/04/21 I learn from Manuel Pégourié-Gonnard that the first versions of his ted had a similar bug, fixed on v1.05 essentially like here; Steven Segletes confirms that his stringstrings doesn't suffer the problem (returning positions of substrings and numbers of occurrences).

[21] Must? Actually, I preferred this solution to other ideas like measuring the length of ⟨*split2*⟩.

[22] If it were for the pattern only, the parsing macro might suffice and the macro calling it might extract the pattern from a "dummy expansion." Somewhat too much for me now; on the other hand the calling macro also hands some "current" informations to the parsing macro—oh, even this could be handled by a general "calling" macro ...

It may be more efficient *not* to use the following setup macros but to type the macros yourself, just using the following as templates. The setup macros are especially useful with patterns that contain "special characters," as when you are looking for lines that might be package comments.

4.3 Meta-Setup

A setup command ⟨*setup-cmd*⟩ will have the following syntax:

`⟨setup-cmd⟩{⟨job-id⟩}[⟨changes⟩]{⟨pattern⟩}⟨more-args⟩`

⟨*changes*⟩ will, in the first instance, be category code changes for reading ⟨*pattern*⟩ overriding the settings in `\PatternCodes`. They are executed after the latter in a local group. It may be safer to redefine `\PatternCodes` instead of using the optional ⟨*changes*⟩ argument.

A macro

`\StartFDsetup{⟨do-setup⟩}{⟨job-id⟩}[⟨changes⟩]`

shared by setup commands may read ⟨*job-id*⟩ and ⟨*changes*⟩ for ⟨*setup-cmd*⟩. `⟨do-setup⟩` will be the macro that reads ⟨*pattern*⟩ (and more) and processes it. It must contain `\endgroup` to match `\begingroup` from `\FD_prepare_pattern`. ⟨*job-id*⟩ is stored in a macro `\fdParserId`. The default for ⟨*changes*⟩ is *nothing*.

```
\newcommand*{\StartFDsetup}[1]{%
    \let\FD_do_setup#1%
    \afterassignment\FD_prepare_pattern
    \def\fdParserId}
\newcommand*{\FD_prepare_pattern}[1][]{%
    \begingroup \PatternCodes #1\FD_do_setup}
```

So ⟨*setup-cmd*⟩ should be set up about as follows:

```
\newcommand*{⟨setup-cmd⟩}{\StartFDsetup⟨do-setup⟩}
\newcommand*{⟨do-setup⟩}[⟨args⟩]{⟨action⟩}
```

⟨*do-setup*⟩'s first argument will be the ⟨*pattern*⟩ argument of ⟨*setup-cmd*⟩.— With v0.5, we learn from the previous and provide

`\MakeSetupCommand{⟨setup-cmd⟩}{⟨do-setup⟩}[⟨args⟩]{⟨action⟩}`

```
\newcommand*{\MakeSetupCommand}[2]{\newcommand*#1{\StartFDsetup#2}%
                                   \newcommand*#2}
```

4.4 Setup for conditionals

`substr_cond` is the "name space" for substring conditionals. A colon separates it from *"job identifiers"* in the actual macro names.

```
\def\substr_cond{substr_cond:}
```

`\MakeSubstringConditional{`⟨*id*⟩`}[`⟨*changes*⟩`]{`⟨*pattern*⟩`}` starts the definition of a conditional with identifier ⟨*id*⟩ and pattern ⟨*pattern*⟩. ⟨*changes*⟩ optionally add commands to be executed after `\PatternCodes` in a local group. `\begingroup` `\mk_substr_cond{`⟨*pattern*⟩`}` can be directly called by other programmer setup commands when `\fdParserId` and ⟨*pattern*⟩ have been read.

```
120  \MakeSetupCommand{\MakeSubstringConditional}{\mk_substr_cond}[1]{%
121    %% #1 pattern string
122    \endgroup \@namedef{\substr_cond \fdParserId}##1#1##2&}
```

This really is not LaTeX. We are starting defining a macro `\substr_cond:`⟨*id*⟩ in primitive TeX with `\def` in the form

`\def\substr_cond:`⟨*id*⟩`#1`⟨*pattern*⟩`#2&`

where `\csname` etc. render '`:`⟨*id*⟩' part of the macro name.[23] The user or programmer macro produces the part of the definition until the delimiter `&` to match the sandbox. You have to add (maybe) `#3` etc. and the `{`⟨*definition-text*⟩`}` just as with primitive TeX.

4.5 Setup for sandboxes

There was a *question:* will we rather see *string macros* or *strings from macro arguments*? The input file content always comes as `\fdInputLine` first, so we at least *must account* for the possibility of string macros as input.

One easy way to apply several checks and substitutions to `\fdInputLine` before the result is written to ⟨*output*⟩ is `\let\OutputString\fdInputLine` and then let `\OutputString` be to what each job refers as *its* input and output, finally `\WriteResult{\OutputString}`. (`\fdInputLine` might better not be touched, it could be used for a final test whether any change applied for some message on screen, even with an entirely expandable chain of actions.) This way each job, indeed each recursive substitution of a single string must start with expanding `\OutputString`.

On the other hand, there is the idea of *"expandable" chains of substitutions*. We may, e.g., define a macro, say, `\manysubstitutions{`⟨*macro-name*⟩`}`, such that `\WriteResult{\manysubstitutions{\fdInputLine}}` writes to ⟨*output*⟩ the result of applying many expandable substitutions to `\fdInputLine`. Such a macro `\manysubstitutions` may read `\fdInputLine`, but it must not redefine any macros. Instead, the substitution macros it calls must read results of previous substitutions as *arguments*.

Another aspect: the order of substitutions should be easy to change. Therefore expanding of string macros should rather be controlled by the way a job is

[23] Loosely speaking of "the parser" ⟨*parser*⟩ around here somehow refers to this macro—but rather to its "parameter text" only, according to TeXbook p. 203. Such a macro, however, won't "parse" only, but it will also execute some job on the results of parsing. Or: a "mere parsing" macro might be macro that transforms a "weird" Plain TeX parameter text into a "simple" parameter text of another macro, consisting of hash marks and digits only. E.g.: `\def\Foo#1`⟨*pattern*⟩`#2&{\foo{#1}{#2}}`.

called, not right here at the *definition* of the job. For this reason, a variant of the sandbox builder expanding some macro was given up.

`setup_substr_cond` is the name space for macros that build sandboxes and initialize arguments for conditional macros.

```
\def\setup_substr_cond{setup_substr_cond:}
```

`\MakeSetupSubstringCondition{`⟨*id*⟩`}[`⟨*changes*⟩`]{`⟨*pattern*⟩`}{`⟨*more-args*⟩`}`

—same ⟨*id*⟩, ⟨*changes*⟩, ⟨*pattern*⟩ as for `\MakeSubstringConditional` (this is bad, there may be `\MakeSubstringConditional*{`⟨*more-args*⟩`}`)—creates the corresponding sandbox, by default without tilde wrap. ⟨*more-args*⟩ may contain `{#1}` to store the string that was tested, also `{`⟨*id*⟩`}` for calling repetitions and `{`⟨*pattern*⟩`}` for screen or log informations.

```
\MakeSetupCommand{\MakeSetupSubstringCondition}
                 {\mk_setup_substr_cond}[2]{%
```

`\mk_setup_substr_cond{`⟨*pattern*⟩`}{`⟨*more-args*⟩`}` can be directly called by other programmer setup commands after `\fdParserId` and ⟨*pattern*⟩ have been read.

```
  %% #1 pattern string
  %% #2 additional arguments, e.g., '{#1}' to keep tested string
  \endgroup
  \expandafter \protected@edef                  %% protected 2011/11/21
```

... keeps `\protect` instead of just not expanding, but I cannot implement `\UseBlogLigs` otherwise right now.

```
    \csname \setup_substr_cond \fdParserId \endcsname ##1{%
      \noexpandcsname \substr_cond \fdParserId \endcsname
```

By `\edef`, the name of the substring conditional is stored here as a single token. The rest of the sandbox follows.

```
          ##1\FD_noexpand~#1\dollar_tilde&#2}%
```

← `\noexpand~` as before v0.5 replaced for v0.51 according to Sec. 4.7.

```
  \let\dollar_tilde\sandbox_dollar}
```

If a tilde `~` has been used instead of `$`, the default is restored.

```
\def\sandbox_dollar{$}
\let\dollar_tilde\sandbox_dollar
```

The following general tool `\noexpandcsname` has been used (many definitions in latex.ltx could have used it):

```
% \def\make_not_expanding_cs#1{%
%     \expandafter \noexpand \csname #1\endcsname}
```

← 2011/11/20 →

```
\def\noexpandcsname{\expandafter\noexpand\csname}
```

4.6 Getting rid of the tildes

`\let~\TildeGobbles` can be used to suppress dummy patterns (contained in ⟨*split2*⟩) in `\write`ing or with `\edef`. ... will probably become obsolete ... however, it is helpful in that you needn't care whether there is a dummy wrap left at all. (2009/04/13)

```
\newcommand{\TildeGobbles}{}        \def\TildeGobbles#1$${}
```

`\RemoveDummyPattern` is used to remove the dummy pattern *immediately*, not waiting for `\write`ing or other "total" expansion:

```
\newcommand{\RemoveDummyPattern}{} \def\RemoveDummyPattern#1~#2$${#1}
```

`\RemoveDummyPatternArg`⟨*macro*⟩`{`⟨*arg*⟩`}` executes `\RemoveDummyPattern` in the next argument:

```
\newcommand*{\RemoveDummyPatternArg}[2]{%
  \expandafter #1\expandafter {\RemoveDummyPattern #2}}
```

`\RemoveTilde` is used to remove the tilde that separated the dummy pattern from ⟨*split1*⟩.

```
\newcommand{\RemoveTilde}{} \def\RemoveTilde#1~{#1}
```

`\RemoveTildeArg`⟨*macro*⟩`{`⟨*arg*⟩`}` executes `\RemoveTilde` in the next argument:

```
\newcommand*{\RemoveTildeArg}[2]{%
  \expandafter #1\expandafter {\RemoveTilde #2}}
```

4.7 Alternative Setup

blog.sty v0.7—"ligatures"—has problems with the tilde. We call a different setup by `\FDpseudoTilde` and go back to the original one by `\FDnormalTilde` (which, however, works only for new processing jobs and processing another file—at present, TODO 2011/11/21):

```
\newcommand*{\FDnormalTilde}{%
    \let\FD_noexpand\noexpand                          %% v0.51
```

v0.5 had a modification of ~ that corrupted typesetting.

```
    \let\RemoveTilde\FD_remove_normal_tilde}
\let\FD_remove_normal_tilde\RemoveTilde
\FDnormalTilde
\newcommand*{\FDpseudoTilde}{%
    \def\FD_noexpand~{\noexpand\pseudo_tilde}%         %% v0.51
    \let\RemoveTilde\FD_remove_pseudo_tilde}
\def\FD_remove_pseudo_tilde#1\pseudo_tilde{#1}
```

TODO 2012/01/20: This way outer braces of splits or the target token list are removed. With blog.sty, this has been relevant for displaying code only (where using source braces for displaying braces, instead of using `\{` and `\}`, it was a bad habit anyway). In order to fix this, the target token list must be surrounded with some additional dummy things, and `\RemoveTildeArg` must add another trick.

4.8 Calling conditionals

`\ProcessStringWith{`⟨*target-string*⟩`}{`⟨*id*⟩`}` builds the sandbox to search ⟨*target-string*⟩ for the ⟨*pattern*⟩ associated with the parser-conditional that is identified by ⟨*id*⟩, the sandbox then calls the parser.

```
\newcommand*{\ProcessStringWith}[2]{%
  \csname \setup_substr_cond #2\endcsname{#1}}
```

`\ProcessExpandedWith{`⟨*string-macro*⟩`}{`⟨*id*⟩`}` does the same but with a *macro* ⟨*string-macro*⟩ (like `\fdInputLine` or `\OutputString`) that stores the string to be tested. `\ProcessExpandedWith{\the`⟨*toks*⟩`}{`⟨*id*⟩`}` with a token list parameter or register ⟨*toks*⟩ may be used as well.

```
\newcommand*{\ProcessExpandedWith}[2]{%
  \csname \setup_substr_cond #2\expandafter \endcsname
    \expandafter{#1}}
```

I would have preferred the reversed order of arguments which seems to be more natural, but the present one is more efficient. Macros with reversed order are currently stored after `\endinput` in section 8, maybe they once return.

Anyway, most desired will be `\ProcessInputWith{`⟨*id*⟩`}` just applying to `\fdInputLine`:

```
\newcommand*{\ProcessInputWith}[1]{%
%   \csname \setup_substr_cond #1\expandafter \endcsname
%     \expandafter{\fdInputLine}}
% %% (Definition almost copied for efficiency.)
  \ProcessExpandedWith\fdInputLine{#1}}            %% 2011/11/21
```

TODO: error when undefined 2009/04/07

4.9 Copy jobs

A job identifier ⟨*id*⟩ may also be considered a mere *hook*, a *placeholder* for a parsing job. What function actually is called may depend on conditions that change while reading the ⟨*input*⟩ file. `\CopyFDconditionFromTo{`⟨*id1*⟩`}{`⟨*id2*⟩`}` *creates or redefines a sandbox builder* with identifier ⟨*id2*⟩ that afterwards behaves like the sandbox builder ⟨*id1*⟩. So you can store a certain behaviour as ⟨*id1*⟩ in advance in order once to change the behaviour of ⟨*id2*⟩ into that of ⟨*id1*⟩.

```
\newcommand*{\CopyFDconditionFromTo}[2]{%
  \expandafter \let
    \csname \setup_substr_cond #2\expandafter \endcsname
      \csname \setup_substr_cond #1\endcsname}
```

(Only the *sandbox* is copied here—what about changing conditionals?)

An "almost" example is typesetting documentation from a package file where the "Legalese" header might be typeset verbatim although it is marked as "comment." (The present example changes "hand-made" macros instead.)

This feature could have been placed more below as a "programming tool."

5 Programming tools

5.1 Tails of conditionals

When creating complex *expandable* conditionals, this may amount to have primitive `\if` ... `\fi` conditionals nested quite deeply, once perhaps too deep for TEX's memory. To avoid this, you can apply the common `\expandafter` trick which finishes the current `\if` ... `\fi` before an inside macro is executed (cf. TEXbook p. 219 on "tail recursion").

Internally tests whether certain strings are present at certain places will be carried out by tests on emptiness or on starting with `~`. E.g., "`#1` = ⟨*split1*⟩ empty" indicates that either the ⟨*pattern*⟩ starts a line or the line is empty altogether (this must be decided by another test).

`\IfFDempty{`⟨*arg*⟩`}{`⟨*when-empty*⟩`}{`⟨*when-not-empty*⟩`}` is used to test ⟨*arg*⟩ on emptyness (without expanding it):

```
\newcommand*{\IfFDempty}[1]{%
  \ifx$#1$\expandafter \@firstoftwo  \else
          \expandafter \@secondoftwo \fi}
```

`\IfFDinputEmpty{`⟨*when-empty*⟩`}{`⟨*when-not-empty*⟩`}` is a variant of the previous to execute ⟨*when-empty*⟩ if the loop processing ⟨*input*⟩ finds an empty line—otherwise ⟨*when-not-empty*⟩.

```
\newcommand*{\IfFDinputEmpty}{%
  \ifx\fdInputLine\@empty \expandafter \@firstoftwo  \else
                          \expandafter \@secondoftwo \fi}
```

`\IfFDdollar{`⟨*arg*⟩`}{`⟨*when-dollar*⟩`}{`⟨*when-not-dollar*⟩`}` is another variant, testing ⟨*split2*⟩ for being `$`, main indicator of there is a match anywhere in ⟨*target*⟩ (as opposed to starting or ending match):

```
\newcommand*{\IfFDdollar}[1]{%
  \ifx$#1\expandafter \@firstoftwo  \else
         \expandafter \@secondoftwo \fi}
```

It is exemplified and explained in section 6. (The whole policy requires that `~` remains active in any testing macros here!)

However, you might always just type the replacement text (in one line) instead of such an `\If`... (for efficiency ...)

If expandability is not desired, you can just chain macros that rework (so re-define) `\OutputString` or so.

2009/04/11: tending towards combining ... Keeping empty input and empty arguments apart is useful in that *one* test of emptiness per input line should suffice—it may be left open whether this should be the first of all tests ...

`\IfFDpreviousInputEmpty{`⟨*when-empty*⟩`}{`⟨*when-not-empty*⟩`}` (v0.42) is a companion of `\IfFDpreviousInputEmpty` referring to `\fdInputLine` as of the *previous* run of the loop in `\ProcessFileWith`, cf. section 3.2, where its choice among its two arguments is determined. It is initialized as follows:

```
178 \newcommand*{\IfFDpreviousInputEmpty}[2]{#2}
```

—which is same as

```
179 \let\IfFDpreviousInputEmpty\@secondoftwo
```

... working like `false`, somewhat. Together with `\IfFDinputEmpty`, it can be used to compress multiple adjacent empty lines into a single one when copying a file.

5.2 Line counter

A LaTeX counter `fdInputLine` may be useful for screen or log messages, moreover you can use it to control processing of the ⟨*input*⟩ file "from outside," not dependent on what the parsing macros find. The header of the file might be typeset verbatim, but we may be too lazy to define the "header" in terms of what is in the file. We just decide that the first ... lines are the "header," even without counting just trying whether the output is fine. It may be necessary to change that number manually when the header changes.

You also can insert lines in ⟨*output*⟩ which have no counterpart in ⟨*input*⟩—if you know what you are doing. With `makedoc`, there is a hook `\EveryComment` that can be used to issue commands "from outside" at a place where executing the command is safe or appropriate.

```
180 \newcounter{fdInputLine}
```

You then must insert `\CountInputLines` in the second argument of `\ProcessFileWith` (or in a macro called from there) so that the counter is stepped.

```
181 \newcommand*{\CountInputLines}{\global\advance\c@fdInputLine\@ne}
```

At present the counter is reset by `\ProcessFileWith`, this may change.

`\IfInputLine{`⟨*relation*⟩⟨*number*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`, when called from the processing loop (second argument of `\ProcessFileWith`) issues ⟨*true*⟩ commands if `\value{fdInputLine}`⟨*relation*⟩⟨*number*⟩ is true, otherwise ⟨*false*⟩. ⟨*relation*⟩ is one out of `<`, `=`, `>`.

```
\newcommand*{\IfInputLine}[1]{%
  \ifnum\c@fdInputLine#1\relax \expandafter \@firstoftwo
  \else                        \expandafter \@secondoftwo \fi}
```

5.3 "Identity job" LEAVE and "default job" *

The job with identifier `LEAVE` *leaves* an (expandable) chain of jobs (as expandable replacement in section 6) and *leaves* the processed string without changing it and without the braces enclosing it:

```
\expandafter \let
  \csname \setup_substr_cond LEAVE\endcsname \@firstofone
```

I.e., `\ProcessStringWith{`⟨*string*⟩`}{LEAVE}` expands to ⟨*string*⟩ ... (Indeed!)

`LEAVE` is used for "chaining" jobs—there will be a routine (Sec. 6.3) to define the action of a job, including what job to run *next* after the present one has been finished. Using this routine, the final job will call `LEAVE`.

`*` may be considered a "meta-job"—it does not directly set up a sandbox, it just "redirects" to the job that has been declared most recently. When job ids are assigned automatically, with `*` you can call that job without knowing its actual id:

```
\@namedef{\setup_substr_cond*}{%
    \csname \setup_substr_cond \fdParserId \endcsname}
```

As an important example, when all the jobs in the chain are expandable, you can call the chain by

```
\WriteResult{\ProcessInputWith{*}}
```

in the body of the file processing loop (Sec. 3.2). TODO: test!

6 Setup for expandable chains of replacements

By the following means, you can create macros (`\Transform` among them) such that, e.g.,

```
\edef\OutputString{\Transform{⟨string⟩}}
```

renders `\OutputString` the result of applying a chain (sequence) of stringwise replacements to ⟨*string*⟩. You can even write a transformed input ⟨*string*⟩ to a file without defining anything after `\read␣to...` In this case however, you don't get any statistical message about what happened or not. With `\edef\OutputString` you can at least issue some `changed!` or `left!` (maybe `\message{!}` vs. `\message{.}`). There is an application in `makedoc` for "typographical upgrading" from plain text to TEX input.

6.1 The backbone macro

`\repl_all_chain_expandable` will be the backbone of the replacements. It is called by some parsing macro ⟨*parser*⟩ and receives from the latter ⟨*split1*⟩ = `#1` and ⟨*split2*⟩ = `#2`. `#3` is the result of what happened so far.

```
\def\repl_all_chain_expandable#1#2#3#4#5#6{%
  %% #1, #2 splits, #3 past,   #4 substitute,
  %% #5 repeat parser,         #6 pass to
%   \ifx~#2\expandafter\@firstoftwo\else\expandafter\@secondoftwo\fi
```

The previous line (or something similar!?) would be somewhat faster, but let us exemplify `\IfFDdollar` from section 5.1 instead:

```
  \IfFDdollar{#2}%
```

If `#2` starts with `$`—with category code 3, "math shift"!, it *is* `$`, due to not reading `$` from input with its standard category code 3 and the sandbox construction (where `$` appears with its standard category code). And this is the case *exactly* when the ⟨*pattern*⟩ from ⟨*parser*⟩ didn't match, again due to the input category codes. Now on *no* match, the sandbox builder `#6` is called with target string `#3#1` where the last tested string is attached to previous results. The ending `~` is removed, `#6` inserts a new wrap for the new dummy pattern.

```
                {\RemoveTildeArg #6{#3#1}}%
```

Otherwise ... the *sandbox builder* ⟨*sandbox*⟩ (that will be shown below) that called ⟨*parser*⟩ initialized `#5` to be that ⟨*parser*⟩ itself. (⟨*parser*⟩ otherwise wouldn't know who it is.) So ⟨*parser*⟩ calls itself with another sandbox `#2&`. Note that `#2` contains '`~`⟨*pattern*⟩`$`' due to the initial ⟨*sandbox*⟩ building.

```
                {#5#2&{#3#1#4}{#4}#5#6}}
```

`#4` is the replacement string that ⟨*sandbox*⟩ passed to ⟨*parse*⟩. The first argument after the `&` is previous stuff plus the recently skipped ⟨*split1*⟩ plus `#4` replacing the string ⟨*pattern*⟩ that was matched.

Finally, `#5` and `#6` again "recall" ⟨*parser*⟩ and the sandbox builder to which to change in case of no other match.

6.2 The basic setup interface macro

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

creates sandbox and parser with common identifier ⟨*id*⟩ and search pattern ⟨*find*⟩. Each occurrence of ⟨*find*⟩ will be replaced by ⟨*replace*⟩. When ⟨*find*⟩ is not found, the sandbox builder for ⟨*id-next*⟩ is called. This may be another replacing macro of the same kind. To return the result without further transformations, call job `LEAVE` (section 5.3). Optional argument ⟨*chng*⟩ changes category codes locally for reading ⟨*find*⟩ and ⟨*replace*⟩.

```
\MakeSetupCommand{\MakeExpandableAllReplacer}
                 {\mk_setup_xpdbl_all_repl}[3]{%
  %% #1 pattern, #2 substitute, #3 pass to
  \endgroup
```

We take pains to call next jobs by single command strings and store them this way, not by `\csname`, as `\ProcessStringWith` would do it. `\edef\@tempa` is used for this purpose, but ...

```
    \protected@edef\@tempa{%                %% protected 2011/11/21
      \noexpand\mk_setup_substr_cond{#1}{%
        {}{#2}%
        \noexpand\noexpand
```

That `\edef\@tempa` must *not expand* tokens computed from `\csname` etc. Moreover, expansion of the parser commands must be avoided another time when `\@tempa` is executed.

```
          \noexpandcsname \substr_cond \fdParserId \endcsname
        \noexpand\noexpand
          \noexpandcsname \setup_substr_cond #3\endcsname}}%
```

Those internal setup commands start with `\endgroup` to switch back to standard category codes. We must match them here by `\begingroup`.

```
    \begingroup \@tempa
    \begingroup \mk_substr_cond{#1}{%
      \repl_all_chain_expandable{##1}{##2}}}
```

The final command is the one that we explained first.

6.3 Automatic chaining

With v0.31,

`\PrependExpandableAllReplacer{`⟨*id*⟩`}[`⟨*cat*⟩`]{`⟨*find*⟩`}{`⟨*replace*⟩`}`

was hoped to be a slight relief in composing replacement chains. It does something like invoking `\MakeExpandableAllReplacer` with ⟨*prev-setup-id*⟩ for the last ⟨*next-id*⟩ argument where ⟨*prev-setup-id*⟩ is the ⟨*id*⟩ of the job that was set up most recently. If you have adjacent lines

`\MakeExpandableAllReplacer{`⟨*id-0*⟩`}{`⟨*find-0*⟩`}{`⟨*subst-0*⟩`}{LEAVE}`
`\PrependExpandableAllReplacer{`⟨*id-1*⟩`}{`⟨*find-1*⟩`}{`⟨*subst-1*⟩`}`
`\PrependExpandableAllReplacer{`⟨*id-2*⟩`}{`⟨*find-2*⟩`}{`⟨*subst-2*⟩`}`

and call ⟨*id-2*⟩, it will call ⟨*id-1*⟩, and the latter will call ⟨*id-0*⟩. So you can reorder the chain by moving `\Prepend`... lines.

With v0.5, `\PrependExpandableAllReplacer*[`⟨*cat*⟩`]{`⟨*find*⟩`}{`⟨*replace*⟩`}` saves you from inventing and typing ⟨*id*⟩, it is automatically generated; or the `*` replaces the ⟨*id*⟩ argument.

```
\newcommand*{\PrependExpandableAllReplacer}{%
    \let\fdParserId_before\fdParserId
    \@ifstar{\stepcounter{fd_line_job}%
             \edef\@tempa{%
                \noexpand\StartFDsetup
                    \noexpand\prep_xpdbl_all_repl
                        {\number\value{fd_line_job}}}%
             \@tempa}%
            {\StartFDsetup\prep_xpdbl_all_repl}}
\newcommand*{\prep_xpdbl_all_repl}[2]{%
    \mk_setup_xpdbl_all_repl{#1}{#2}{\fdParserId_before}}%
    %% #1 pattern, #2 substitute, #3 pass to
```

\StartPrependingChain makes \MakeExpandableReplacer superfluous, in the sense that the above chain setup can be achieved as well like this:

```
\StartPrependingChain
\PrependExpandableAllReplacer{⟨id-0⟩}{⟨find-0⟩}{⟨subst-0⟩}
\PrependExpandableAllReplacer{⟨id-1⟩}{⟨find-1⟩}{⟨subst-1⟩}
\PrependExpandableAllReplacer{⟨id-2⟩}{⟨find-2⟩}{⟨subst-2⟩}
```

or with v0.5:

```
\StartPrependingChain
\PrependExpandableAllReplacer*{⟨find-0⟩}{⟨subst-0⟩}
\PrependExpandableAllReplacer*{⟨find-1⟩}{⟨subst-1⟩}
\PrependExpandableAllReplacer{⟨start⟩}{⟨find-2⟩}{⟨subst-2⟩}
```

This adds a code line, but this way you can choose the final "real" job more easily. So you can think of \StartPrependingChain as "initializing a chain of prependments." As to the * version, the example suggests using an explicit ⟨*id*⟩ argument finally if you want to invoke the chain explicitly by a line job identity (without counting the declaration lines—however, note job * according to Sec. 5.3).

```
\newcommand*{\StartPrependingChain}{%
    \def\fdParserId{LEAVE}%
    \setcounter{fd_line_job}{0}} %% 2011/11/13; TODO \Nameprefix...
\newcounter{fd_line_job}
```

6.4 CorrectHook launching the replacement chain

\MakeDocCorrectHook{⟨*string*⟩} belongs to makedoc, but in the meantime (nicetext release 0.3) I have proposed to use it with fifinddo only as well (running files fdtxttex.tpl, fdtxttex.tex). Therefore I offer some simplification \SetCorrectHookJob{⟨*job-id*⟩} for defining \MakeDocCorrectHook *here*.

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

\SetCorrectHookJobLast just uses the job that was set up most recently.

```
\newcommand*{\SetCorrectHookJobLast}
            {\SetCorrectHookJob\fdParserId}
```

`\CorrectedInputLine` results from `\MakeDocCorrectHook` when the latter is applied to `\fdInputLine`:

```
\newcommand*{\CorrectedInputLine}{%
    \expandafter \MakeDocCorrectHook \expandafter{\fdInputLine}}
```

7 Leave package mode

We restore the underscore _ for math subscripts. (This might better depend on something ...)

```
\catcode`\_=8
\endinput
```

TEX ignores the rest of the file when it is *input* "in the sense of `\input`", as opposed to just reading the file line by line to a macro like `\fdInputLine`.

8 Pondered

```
  %% TODO abbreviated commands (aliases) \MkSubstrCond...
  %% TODO \@onlypreamble!?
\newcommand*{\ApplySubstringConditional}[1]{%
  %% #1 identifier; text to be searched expected next
  \csname setup_substr_cond:#1\endcsname}
\newcommand*{\ApplySubstringConditionalToExpanded}[1]{% 2009/03/31+
  \csname setup_substr_cond:#1\expandafter \endcsname \expandafter}
\newcommand*{\ApplySubstringConditionalToInputString}[1]{% 2009/03/31+
  \csname setup_substr_cond:#1\expandafter \endcsname
    \expandafter {\fdInputLine}}
  %% TODO or `\OutputString', even `\read' to `\OutputString'!?
% \newcommand*{\ApplySubstringConditionalToExpanded}[2]{%
%   %% note: without assignments, robust!
%   %% BUT the `\csname ... \expandafter \endcsname' method is faster
%   \expandafter \reversed_apply_substr_cond
%     \expandafter {#2}{#1}}
% \newcommand*{\reversed_apply_substr_cond}[2]{%
%   \ApplySubstringConditional{#2}{#1}}
  %% ODER:
% \newcommand*{\expand_attach_arg}[2]{%% 2009/03/31
%   %% #1 command with previous args, TODO cf. LaTeX3
%   \expandafter \attach_arg \expandafter {#1}{#2}}
%   %% actually #1 may contain more than one token,
%   %% only first expanded
% \newcommand*{\attach_arg}[2]{#2{#1}}
% \newcommand*{\ApplySubstringConditionalToExpanded}[2]{%
%   \expandafter \attach_arg \expandafter
%     {#2}{\ApplySubstringConditional{#1}}}
```

9 VERSION HISTORY

```
v0.1   2009/04/03   very first version, tested on morgan.sty
v0.2   2009/04/05   counter fdInputLine, \ProvidesFile moved from
                    \ProcessFile to \ResultFile, \CopyFD...,
                    category section first, more sectioning,
                    suppressing empty code lines before section
                    titles; discussion, \Delimiters
       2009/04/06   more discussion
       2009/04/07   more discussion, factored \WriteProvides out from
                    \ResultFile, \ProcessExpandedWith corrected
       2009/04/08   \InputString -> \fdInputline;
                    removed \ignorespaces
       2009/04/09   \WhenInputLine[2] -> \IfInputline[3],
                    \ProcessInputWith, typos,
                    \WriteProvides message ‘with’
       2009/04/10   \make_not_expanding_cs
                    DISCOVERED ‘‘IF SUBSTRING" ALGORITHM WRONG
                    (<str1><str2><str1> in <str1><str2>)
v0.3   2009/04/11   SOME THINGS GIVEN UP EARLIER WILL BE REMOVED,
                    TO BE STORED IN THE COPY AS OF 2009/04/10
                    mainly: sandbox setup (tilde/dollar)
                    REAL ADDITION: setup for expandable replacing
       2009/04/12   played with ‘chain’ vs. ‘sequence’;
                    plain ‘...’, ‘cf.’, ‘etc.’ for ‘mdcorr.cfg’
       2009/04/13   \RemoveTilde...
       2009/04/15   reworked text, same mistake \in@
v0.31  2009/04/21f. comments on ted, stringstrings
       2009/12/28   "onwards)" !? "safer", not "more safe"
       2010/03/10   the loop starts
       2010/03/17   corr. t^ete; set up -> setup for
       2010/03/18   TODO EOF, ctan.org/pkg/newfile; non-ASCII
       2010/03/19   extended description of \MakeExpandableAll...;
                    ’’ -> "
       2010/03/20   \ctanpkgref
       2010/03/22   \StartFDsetup, \Prepend...
       2010/03/23   URL for ‘substr.sty’
SENT TO CTAN

v0.4   2010/03/24   removed \pagebreak before "substrings";
                    <relation> with \IfInputLine precisely
       2010/03/25   todo \ProcessExp... more precisely, etc.
       2010/03/26   ... was wrong, removed
       2010/03/29   \SetPatternCodes, \ResetPatternCodes,
                    \SetCorrectHookJob, \SetCorrectHookJobLast;
                    <relation> with \HardNVerb;
                    don’t mention \begingroup with
                    \mk_setup_substr_cond; renamed v0.4
belonged to nicetext RELEASE 0.4
v0.4a  2010/04/04   copyright 2010
```

```
belonged to nicetext RELEASE 0.41

v0.41  2010/04/06   more on \ProcessExpanded...;
                    \ProcessFile... gets opt arg
       2010/04/13   \ProcessFile{<file>}... shows <file>
used by blog.sty v0.1, v0.2
v0.42  2010/11/09   typo corr.
       2010/11/10   \IfFDpreviousInputEmpty
       2010/11/11   \BasicNormalCatcodes from blog.sty,
                    \CopyFile*, \CopyLine; v3. -> v0.3;
                    LPPL v1.3c
       2010/11/12   \CatCode replaced (implemented in niceverb only)
       2010/11/13   \CopyFile with \message{.}
       2010/11/24   reworked doc. of replacement setup;
                    \StartPrependingChain
       2010/11/25   corr. typo \@backslash...; doc. changes;
                    \CopyLine indeed, not \fdCopyLine
       2010/11/27   footnote on "parser", other doc. corr.s
       2011/01/20   corr. "period" AMSTERDAM
       2011/01/25   updated (C); footnotes to ‘substring theory’;
                    TODO with \RemoveTilde; some manual line spacings
                    (adding ‘\ ’)
       -> r0.42
v0.43  2011/08/06   doc.: mistake \WriteResult/\ResultFile,
       2011/08/22         use \acro
       2011/09/12f. \CorrectedInputLine - reworded for breaking
       -> r0.44
v0.44  2011/10/23   messages from \ResultFile and \CloseResultFile
       -> r0.46
v0.45  2011/10/26   little modification of \CloseResultFile message,
                    "sacrificing" \pagebreak before Sec. 4 -- fine!
v0.5   2011/11/13   \PrependExpandableAllReplacer*, \MakeSetupCommand;
                    doc.: {center} with too long verbatim quote
       2011/11/19   input check fixed, doc. there adjusted,
                    \ifFinalInputFile, \ProcessFinalFileWith, job ‘*’
       2011/11/20   \noexpandcsname; "default job" lowercase
       2011/11/21   \ProcessInputWith "less efficient",
                    \protected@edef, "pseudo-tilde"
       -> r0.5
v0.51  2012/01/20   updated (C); TODO on pseudo-tilde
       2012/03/17   fixed \FDnormalTilde/\FDpseudoTilde

TODO: cleveref 2010/03/18
```