

# fifinddo

## Filtering T<sub>E</sub>X(t) Files by T<sub>E</sub>X

Uwe Lück\*

March 23, 2010

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

### Abstract

**fifinddo** starts implementing parsing of plain text or T<sub>E</sub>X files using T<sub>E</sub>X, generalizing the philosophy behind **docstrip**, based on how T<sub>E</sub>X reads macro arguments. Rather than typesetting the edited input stream immediately, results are written to another file, in the first instance as input for T<sub>E</sub>X. 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**→T<sub>E</sub>X functionality. The method of testing for substrings is carefully discussed, revealing an earlier mistake (then) shared with **substr.sty** and L<sup>A</sup>T<sub>E</sub>X’s internal **\in@**.

## Contents

<b>1</b>	<b>Introduction: The Gnome of the Aim</b>	<b>2</b>
1.1	Parsing by T <sub>E</sub> X—are you mad? . . . . .	2
1.2	Useful for . . . . .	3
1.2.1	Comparisons . . . . .	5
1.3	For insiders . . . . .	5

---

\*<http://contact-ednotes.sty.de.vu>

1	INTRODUCTION: THE GNOME OF THE AIM	2
2	Preliminaries	6
2.1	Head of file (Legalese)	6
2.2	Format and package version	7
2.3	Category codes	7
3	File handling	8
4	Basic handling of substring conditionals	10
4.1	“Substring Theory”	10
4.2	Plan for proceeding	11
4.3	Meta-Setup	12
4.4	Setup for conditionals	12
4.5	Setup for sandboxes	13
4.6	Getting rid of the tildes	14
4.7	Calling conditionals	15
4.8	Copy jobs	15
5	Programming tools	16
5.1	Tails of conditionals	16
5.2	Line counter	17
5.3	The “identity job” LEAVE	17
6	Setup for expandable chains of replacements	18
7	Leave package mode	20
8	Pondered	20
9	VERSION HISTORY	21

# 1 Introduction: The Gnome of the Aim

## 1.1 Parsing by T<sub>E</sub>X—are you mad?

The package name `fifinddo` is a `\listfiles`-compatible abbreviation of ‘file-find-do’<sup>1</sup> (or think of ‘if found do’). `fifinddo` implements (or aims at) general parsing (extracting, replacing [converting], expanding, ...) using T<sub>E</sub>X where `texhax` posters strongly urge to use `sed`, `awk`, or Perl. `fifinddo`’s opposed rationales are:

- It works instantly on any T<sub>E</sub>X installation. (*Restrictions:* Some T<sub>E</sub>X versions `\write` certain hex codes for certain characters, cf. T<sub>E</sub>Xbook p. 45,

---

<sup>1</sup>‘file’ possibly for “searching T<sub>E</sub>X(t) files” (I don’t remember my thoughts!), while there were requests for doing replacements on L<sup>A</sup>T<sub>E</sub>X *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.

I have seen this with PCT<sub>EX</sub>. 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 <sub>EX</sub> macros, knowing just <sub>EX</sub> (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 <sub>EX</sub> 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 <sub>EX</sub> packages; and while I am typing this, I find at least 14 other similar packages in Jürgen Fenn’s *Topic Index* of the *<sub>EX</sub> Catalogue*:

<http://mirror.ctan.org/help/Catalogue/bytopic.html#parsingfiles><sup>2</sup>

(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 <sub>EX</sub>—yes, celebrate!)

Actually, <sub>EX</sub>’s mechanism of collecting macro arguments is hard-wired parsing at quite a high level. L<sub>AT</sub><sub>EX</sub> hides this from “simple-minded” users by a convention *not* to use that full power of <sub>EX</sub> for *end-user macros*. Internally, L<sub>AT</sub><sub>EX</sub> *does* use it in reading lists of options and file dates as well as to implement certain FOR- and WHILE-like loop programming structures. L<sub>AT</sub><sub>EX</sub>’s `\in@/ifin@` construction is an implementation of a “ $\langle string1 \rangle$  occurs in  $\langle string2 \rangle$ ” test. More packages seem to use this idea for extracting file informations, like `texshade`.<sup>3</sup>

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

## 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:

---

<sup>2</sup>Click here!

<sup>3</sup><http://ctan.org/pkg/texshade>

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.<sup>4</sup>
- 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.`<sup>5</sup> This could replace packages like `easylatex`,<sup>6</sup> `txt2latex`,<sup>7</sup> `txt2tex`<sup>8</sup> 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  $\text{\LaTeX}$  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  $\text{\TeX}$  macros, but by text substitution;

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,

<sup>4</sup>If you know the “names” of the encodings, Heiko Oberdiek’s `stringenc` may be preferable.

<sup>5</sup>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.

<sup>6</sup><http://ctan.org/pkg/easylatex>

<sup>7</sup><http://ctan.org/pkg/txt2latex>

<sup>8</sup><http://ctan.org/pkg/txt2tex>

- 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`,<sup>9</sup> `ted`,<sup>10</sup> `xstrings`<sup>11</sup> (as I understand them, perhaps also `coolstr`<sup>12</sup>) 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.<sup>13</sup> 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 T<sub>E</sub>X's category codes and how T<sub>E</sub>X 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.

That `fifinddo` acts on “T<sub>E</sub>X(t)” files or so means that (at present) I think of applications on “plain text” files which will usually be T<sub>E</sub>X 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 T<sub>E</sub>X 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 `\writeing` 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}}`

---

<sup>9</sup><http://ctan.org/pkg/stringstrings>

<sup>10</sup><http://ctan.org/pkg/ted>

<sup>11</sup><http://ctan.org/pkg/xstrings>

<sup>12</sup><http://ctan.org/pkg/coolstr>

<sup>13</sup>There is more in my badly implemented `txtproc.sty`.

which breaks in “mere expanding” (T<sub>E</sub>X *evaluates* `\tempo` instead of defining it). An *expandable* test on emptiness is, e.g. `\ifx$#\langle n\rangle$`, where we hope that it becomes `\iftrue` just if macro argument `\langle n\rangle` is empty indeed. However, “usually” it may *also* become `\iftrue` when `\langle n\rangle` starts with `$`—if the latter has category code 3 (“math shift”). But `fifinddo` does not assign category code 3 to any character from the input file! Therefore `\ifx$#\langle n\rangle$` is `\iftrue` *exactly* if `\langle n\rangle` 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.

Once there may be an option to read input with some usual T<sub>E</sub>X category codes as well, it may be useful to (some of)

- avoid matching substrings of control words,
- skip blank spaces as T<sub>E</sub>X does it usually,
- catch *balanced* input pieces,
- ignore comments,
- ignore certain characters.

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)

```

1  %% Macro package 'fifinddo.sty' for LaTeX2e,      %% FIDO, FIND!
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 processing tex(t) files
6  %% (checking, filtering, converting, substituting, expanding, ...)
7
8  \def\fileversion{0.31} \def\filedate{2010/03/23}
9
10 %% This file can be redistributed and/or modified under
11 %% the terms of the LaTeX Project Public License; either
12 %% version 1.3a of the License, or any later version.
13 %% The latest version of this license is in

```

```

14 %%
15 %%      http://www.latex-project.org/lppl.txt
16 %%
17 %% We did our best to help you, but there is NO WARRANTY.
18 %% Please report bugs, problems, and suggestions via
19 %%
20 %%      http://www.contact-ednotes.sty.de.vu
21 %%
22 %% For the full documentation, look for 'fifinddo.pdf'.
23 %% Its source starts in 'fifinddo.tex'.

```

## 2.2 Format and package version

```

24 \NeedsTeXFormat{LaTeX2e}[1994/12/01]
25 % 1994/12/01: \newcommand* etc.
26 \ProvidesPackage{fifinddo}[\filedate\space v\fileversion\space
27                      filtering TeX(t) files by TeX (UL)]

```

## 2.3 Category codes

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

```

28 \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.

```

29 \ifundefined{MakeOther}{\let\MakeOther\@makeother}{%
30 \newcommand*{\fdPatternCodes}{\MakeOther\&\MakeOther\$}
31 \newcommand*{\PatternCodes}{% \let\PatternCodes\fdPatternCodes
32   %% TODO adding/removing

```

It would be bad to have `\MakeOther%` and `\MakeOther\` here in that this may have unexpected, weird effects with arguments of setup macros. 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\[\]`:

```

33 \newcommand*{\Delimiters}[2]{%
34   \MakeOther\{\MakeOther\}\catcode'#1=1\catcode'#2=2\relax}

```

For replacing strings or for defining other strings of “other” characters by `\edef`, you can use some L<sup>A</sup>T<sub>E</sub>X constructs—here are copies `\PercentChar` and `\BackslashChar` of them (do you need more?):

```

35 \newcommand*{\PercentChar}{% \let\PercentChar\@percentchar
36 \newcommand*{\BackslashChar}{% \let\BackslashChar\@backslashchar

```

### 3 File handling

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

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

```
38 \newcommand*{\ResultFile}[1]{%
39   \def\result_file_name{#1}%
40   \immediate\openout\result_file=#1}
```

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

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

`\WriteProvides` writes a `\ProvidesFile` command to the opened `⟨output⟩` file. This should be used when `⟨output⟩` is made as L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> input.

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

`\ProcessFileWith{⟨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`.

```
48 \newcommand*{\ProcessFileWith}[2]{%
49   \openin\@inputcheck=#1%
50   % \ifeof\@inputcheck %% bad 'exists?' test
51   %   \PackageError{fifinddo}{File '#1' not here}%
52   %                                     {Mistyped?}%
53   %   \else
54   %     \global\c@fdInputLine=\z@ %% line counter reset
55   %     \begingroup
56   %       \MakeOther\{\MakeOther\}\@sanitize
57   %       %% from docstrip.tex:
58   %       % \MakeOther\^^A\MakeOther\^^K%% irrelevant, not LaTeX
59   %       \endlinechar\m@ne
60   %       %% <- cf. TeXbook "extended keyboards" up-/downarrow
61   %       -> "math specials", cf. "space specials"
62   %       \MakeOther\^^I% ASCII horizontal tab -- guessed!? ^^L!?
```

With v3.1, we support non-ASCII:

```
63   \count@=128
64   \loop
65   \ifnum\count@<\cclvi
66     \catcode\count@=12
67     \advance\count@\@ne
68   \repeat
```



```

69      \loop \ifeof\@inputcheck \else
70          \read\@inputcheck to \fdInputLine
71          \ignorespaces #2%
72      \repeat
73  \endgroup
74  % \fi
75  \closein\@inputcheck}

```

`\CloseResultFile` closes *output*.

```

76  \newcommand*{\CloseResultFile}{\immediate\closeout\result_file}

```

TODO: write EOF for debugging!?—Peter Wilson’s `newfile` provides more powerful file handling.

## 4.1 “Substring Theory”

<sup>17</sup>Polynesian: aku aku, rongorongo, wiki wiki ...

or `\IfSubStringInString{ionization}{ionizat}{YES}{NO}`.<sup>18</sup> Same with L<sup>A</sup>T<sub>E</sub>X's internal `\in@`.<sup>19</sup>

`\makeatletter\in@{bonbon}{bon}\ifin@_YES\else_NO\fi\makeatother`

In general, the previous approach *fails if and exactly if*  $\langle pattern \rangle$  has a period  $p$ —less than its length—in the sense of that the  $p$ th token to the right or left of each token in  $\langle pattern \rangle$  is the *same* token. AMSTERDAM has a period 8, `day_after_day` 10, `bonbon` 3, `bonobo` 4. There is a counterexample  $\langle target \rangle$  of length  $p$  iff  $\langle pattern \rangle$  has period  $p$ , namely the first substring of  $\langle pattern \rangle$  having length  $p$ . If the length of  $\langle pattern \rangle$  exceeds a multiple  $mp$  of its period, the first  $mp$  tokens of  $\langle pattern \rangle$  form a counterexample  $\langle target \rangle$ .

Therefore, a sandbox must have something between  $\langle target \rangle$  and  $\langle pattern \rangle$ .<sup>20</sup> We choose `\find<target>~<pattern>$&` as standard. The `$` will be used as an argument delimiter to get rid of the dummy  $\langle pattern \rangle$  in  $\langle split2 \rangle$ , as well as to decide whether the match was in  $\langle target \rangle$  or in the dummy part of the sandbox. The `$` can be replaced by another tilde `~` in order to test whether  $\langle target \rangle$  ends on a  $\langle pattern \rangle$ , 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.<sup>21</sup> 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.

<sup>18</sup>Read `substr.sty` or try “normal” things to convince yourself that the syntax indeed is `\IfSubStringInString{<pattern>}{<target>}{<yes>}{<no>}`.

<sup>19</sup>`\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).

<sup>20</sup>Must? Actually, I preferred this solution to other ideas like measuring the length of  $\langle split2 \rangle$ .

<sup>21</sup>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  $\langle setup\text{-}cmd \rangle$  will have the following syntax:

$$\boxed{\langle setup\text{-}cmd \rangle \{ \langle job\text{-}id \rangle \} [ \langle changes \rangle ] \{ \langle pattern \rangle \} \langle more\text{-}args \rangle}$$

$\langle changes \rangle$  will, in the first instance, be category code changes for reading  $\langle pattern \rangle$  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  $\langle changes \rangle$  argument.

A macro

$$\boxed{\backslash\text{StartFDsetup} \{ \langle do\text{-}setup \rangle \} \{ \langle job\text{-}id \rangle \} [ \langle changes \rangle ]}$$

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

```

77 \newcommand*\backslashStartFDsetup}[1]{%
78   \let\FD_do_setup#1%
79   \afterassignment\FD_prepare_pattern
80   \def\fdParserId}
81 \newcommand*\backslashFD_prepare_pattern}[1][ ]{%
82   \begingroup \PatternCodes #1\FD_do_setup}

```

So  $\langle setup\text{-}cmd \rangle$  should be set up about as follows:

$$\begin{aligned} &\backslash\text{newcommand}*\{ \langle setup\text{-}cmd \rangle \} \{ \backslash\text{StartFDsetup} \langle do\text{-}setup \rangle \} \\ &\backslash\text{newcommand}*\{ \langle do\text{-}setup \rangle \} [ \langle args \rangle ] \{ \langle action \rangle \} \end{aligned}$$

$\langle do\text{-}setup \rangle$ ’s first argument will be the  $\langle pattern \rangle$  argument of  $\langle setup\text{-}cmd \rangle$ .

### 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.

```

83 \def\substr_cond{substr_cond:}

```

$\boxed{\backslash\text{MakeSubstringConditional} \{ \langle id \rangle \} [ \langle changes \rangle ] \{ \langle pattern \rangle \}}$  starts the definition of a conditional with identifier  $\langle id \rangle$  and pattern  $\langle pattern \rangle$ .  $\langle changes \rangle$  optionally add commands to be executed after `\PatternCodes` in a local group.

```

84 \newcommand*\backslashMakeSubstringConditional}
85   {\backslashStartFDsetup\mk_substr_cond}

```

`\begingroup \mk_substr_cond{<pattern>}` can be directly called by other programmer setup commands when `\fdParserId` and `<pattern>` have been read.

```
86 \def\mk_substr_cond #1{%% #1 pattern string
87 \endgroup \@namedef{\substr_cond \fdParserId}##1#1##2&}
```

This really is not L<sup>A</sup>T<sub>E</sub>X. We are starting defining a macro `\substr_cond:<id>` in primitive T<sub>E</sub>X with `\def` in the form

```
\def\substr_cond:<id>#1<pattern>#2&
```

where `\csname` etc. render ‘`<id>`’ part of the macro name. 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 T<sub>E</sub>X.

## 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 *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.

```
88 \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.  $\langle more-args \rangle$  may contain  $\{\#1\}$  to store the string that was tested, also  $\{\langle id \rangle\}$  for calling repetitions and  $\{\langle pattern \rangle\}$  for screen or log informations.

```
89 \newcommand*\MakeSetupSubstringCondition{
90     {\StartFDsetup\mk_setup_substr_cond}
```

`\begingroup` `\mk_setup_substr_cond{\langle pattern \rangle}{\langle more-args \rangle}` can be directly called by other programmer setup commands after `\fdParserId` and  $\langle pattern \rangle$  have been read:

```
91 \def\mk_setup_substr_cond #1#2{% #1 pattern string,
92     % #2 additional arguments, e.g., '{#1}' to keep tested string
93     \endgroup
94     \expandafter \edef
95     \csname \setup_substr_cond \fdParserId \endcsname ##1{%
96         \make_not_expanding_cs{\substr_cond \fdParserId}%
```

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

```
97         ##1\noexpand~#1\dollar_tilde&#2}%
98     \let\dollar_tilde\sandbox_dollar}
```

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

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

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

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

## 4.6 Getting rid of the tildes

`\let~\TildeGobbles` can be used to suppress dummy patterns (contained in  $\langle split2 \rangle$ ) in `\writeing` 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)

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

`\RemoveDummyPattern` is used to remove the dummy pattern *immediately*, not waiting for `\writeing` or other “total” expansion:

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

`\RemoveDummyPatternArg\macro{\langle arg \rangle}` executes `\RemoveDummyPattern` in the next argument:

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

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

```
107 % %% An alternative policy is to pass
108 % %% <target> (as an argument) to the parsing macro.
109 \newcommand{\RemoveTilde}{} \def\RemoveTilde#1~{#1}
```

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

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

## 4.7 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.

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

`\ProcessExpandedWith{<string-macro>}{<id>}` does the same but with a *macro* (like `\fdInputLine` or `\OutputString`) in which the string to be tested is stored.

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

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

Anyway, most desired will be `\ProcessInputWith{<id>}` just applying to `\fdInputLine`:

```
117 \newcommand*{\ProcessInputWith}[1]{%
118   \csname \setup_substr_cond #1\expandafter \endcsname
119   \expandafter{\fdInputLine}}
```

(Definition almost copied for efficiency.)

```
120 %%% TODO: error when undefined 2009/04/07
```

## 4.8 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>`.

```

121 \newcommand*{\CopyFDconditionFromTo}[2]{%
122   \expandafter \let
123   \csname \setup_substr_cond #2\expandafter \endcsname
124   \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 T<sub>E</sub>X’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. T<sub>E</sub>Xbook 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 =  $\langle split1 \rangle$  empty`” indicates that either the  $\langle pattern \rangle$  starts a line or the line is empty altogether (this must be decided by another test).

`\IfDempty{ $\langle arg \rangle$ }{ $\langle when-empty \rangle$ }{ $\langle when-not-empty \rangle$ }` is used to test  $\langle arg \rangle$  on emptiness (without expanding it):

```

125 \newcommand*{\IfDempty}[1]{%
126   \ifx$#1$\expandafter \@firstoftwo \else
127   \expandafter \@secondoftwo \fi}

```

`\IfDinputEmpty{ $\langle when-empty \rangle$ }{ $\langle when-not-empty \rangle$ }` is a variant of the previous to execute  $\langle when-empty \rangle$  if the loop processing  $\langle input \rangle$  finds an empty line—otherwise  $\langle when-not-empty \rangle$ .

```

128 \newcommand*{\IfDinputEmpty}{%
129   \ifx\fdInputLine\@empty \expandafter \@firstoftwo \else
130   \expandafter \@secondoftwo \fi}

```

`\IfDdollar{ $\langle arg \rangle$ }{ $\langle when-empty \rangle$ }{ $\langle when-not-empty \rangle$ }` is another variant, testing  $\langle split2 \rangle$  for being \$, main indicator of there is a match anywhere in  $\langle target \rangle$  (as opposed to starting or ending match):

```

131 \newcommand*{\IfDdollar}[1]{%
132   \ifx$#1\expandafter \@firstoftwo \else
133   \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 ...

## 5.2 Line counter

A L<sup>A</sup>T<sub>E</sub>X 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.

```
134 \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.

```
135 \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>* may usually be just `=`.

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

## 5.3 The “identity job” LEAVE

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:

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

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

## 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 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.

`\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.

```
141 \def\repl_all_chain_expandable#1#2#3#4#5#6{%
142   %% #1, #2 splits, #3 past,   #4 substitute,
143   %% #5 repeat parser,        #6 pass to
144   % \ifx~#2\expandafter\@firstoftwo\else\expandafter\@secondoftwo\fi
```

The previous line would be somewhat faster, but let us exemplify `\IfFDdollar` from section 5.1 instead:

```
145 \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.

```
146 {\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.

```
147 {#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.

```
\MakeExpandableAllReplacer{<id>}[<chg>]{<find>}{<replace>}{<id-next>}
```

creates sandbox and parser with common identifier  $\langle id \rangle$  and search pattern  $\langle find \rangle$ . Each occurrence of  $\langle find \rangle$  will be replaced by  $\langle replace \rangle$ . When  $\langle find \rangle$  is not found, the sandbox builder for  $\langle id-next \rangle$  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  $\langle chg \rangle$  changes category codes locally for reading  $\langle find \rangle$  and  $\langle replace \rangle$ .

```
148 \newcommand*{\MakeExpandableAllReplacer}
149     {\StartFDsetup\mk_setup_xpdbl_all_repl}
150 \newcommand*{\mk_setup_xpdbl_all_repl}[3]{%
151     %% #1 pattern, #2 substitute, #3 pass to
152     \endgroup
```

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

```
153 \edef\@tempa{%
154     \noexpand\mk_setup_substr_cond{#1}{%
155         {#2}%
156     \noexpand\noexpand
```

That  $\backslash edef\@tempa$  must *not expand* the controll words after they have been computed from  $\backslash csname$  etc. Moreover, expansion of the parser commands must be avoided another time, when  $\backslash @tempa$  is executed.

```
157 \make_not_expanding_cs{\substr_cond\fdParserId}%
158 \noexpand\noexpand
159 \make_not_expanding_cs{\setup_substr_cond #3}}%
```

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

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

The final command is the one that we explained first.

```
\PrependExpandableAllReplacer{<id>}[<cat>]{<find>}{<replace>}
```

is hoped to be a slight relief in composing replacement chains. It does something like invoking  $\backslash MakeExpandableAllReplacer$  with  $\langle prev-setup-id \rangle$  for the last  $\langle next-id \rangle$  argument where  $\langle prev-setup-id \rangle$  is the  $\langle id \rangle$  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  $\langle id-2 \rangle$ , it will call  $\langle id-1 \rangle$ , and the latter will call  $\langle id-0 \rangle$ . So you can reorder the chain by moving `\Prepend...` lines.

```

163 \newcommand*\PrependExpandableAllReplacer{%
164     \let\fdParserId_before\fdParserId
165     \StartFDsetup\prep_xpdbl_all_repl}
166 \newcommand*\prep_xpdbl_all_repl}[2]{%
167     \mk_setup_xpdbl_all_repl{#1}{#2}{\fdParserId_before}}%
```

## 7 Leave package mode

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

```

168 \catcode'\_ =8    %% restores underscore use for subscripts
169 \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

```

170 %% TODO abbreviated commands (aliases) \MkSubstrCond...
171 %% TODO \@onlypreamble!?
172 \newcommand*\ApplySubstringConditional}[1]{%
173     %% #1 identifier; text to be searched expected next
174     \csname setup_substr_cond:#1\endcsname}
175 \newcommand*\ApplySubstringConditionalToExpanded}[1]{% 2009/03/31+
176     \csname setup_substr_cond:#1\expandafter \endcsname \expandafter}
177 \newcommand*\ApplySubstringConditionalToInputString}[1]{% 2009/03/31+
178     \csname setup_substr_cond:#1\expandafter \endcsname
179     \expandafter {\fdInputLine}}
180 %% TODO or '\OutputString', even '\read' to '\OutputString'!?
181 % \newcommand*\ApplySubstringConditionalToExpanded}[2]{%
182 %     %% note: without assignments, robust!
183 %     %% BUT the '\csname ... \expandafter \endcsname' method is faster
184 %     \expandafter \reversed_apply_substr_cond
185 %     \expandafter {#2}{#1}}
186 % \newcommand*\reversed_apply_substr_cond}[2]{%
187 %     \ApplySubstringConditional{#2}{#1}}
188 %% ODER:
189 % \newcommand*\expand_attach_arg}[2]{% 2009/03/31
190 %     %% #1 command with previous args, TODO cf. LaTeX3
191 %     \expandafter \attach_arg \expandafter {#1}{#2}}
192 %     %% actually #1 may contain more than one token,
193 %     %% only first expanded
194 % \newcommand*\attach_arg}[2]{#2{#1}}
195 % \newcommand*\ApplySubstringConditionalToExpanded}[2]{%
196 %     \expandafter \attach_arg \expandafter
```

197 %     {#2}{\ApplySubstringConditional{#1}}}

## 9 VERSION HISTORY

198 v0.1   2009/04/03   very first version, tested on morgan.sty  
199 v0.2   2009/04/05   counter fdInputLine, \ProvidesFile moved from  
200                      \ProcessFile to \ResultFile, \CopyFD...,  
201                      category section first, more sectioning,  
202                      suppressing empty code lines before section  
203                      titles; discussion, \Delimiters  
204                      2009/04/06   more discussion  
205                      2009/04/07   more discussion, factored \WriteProvides out from  
206                      \ResultFile, \ProcessExpandedWith corrected  
207                      2009/04/08   \InputString -> \fdInputline;  
208                      removed \ignorespaces  
209                      2009/04/09   \WhenInputLine[2] -> \IfInputline[3],  
210                      \ProcessInputWith, typos,  
211                      \WriteProvides message 'with'  
212                      2009/04/10   \make\_not\_expanding\_cs  
213                      DISCOVERED 'IF SUBSTRING' ALGORITHM WRONG  
214                      (<str1><str2><str1> in <str1><str2>)  
215 v0.3   2009/04/11   SOME THINGS GIVEN UP EARLIER WILL BE REMOVED,  
216                      TO BE STORED IN THE COPY AS OF 2009/04/10  
217                      mainly: sandbox setup (tilde/dollar)  
218                      REAL ADDITION: setup for expandable replacing  
219                      2009/04/12   played with 'chain' vs. 'sequence';  
220                      plain '...', 'cf.', 'etc.' for 'mdcorr.cfg'  
221                      2009/04/13   \RemoveTilde...  
222                      2009/04/15   reworked text, same mistake \in@  
223 v0.31   2009/04/21f. comments on ted, stringstrings  
224                      2009/12/28   "onwards)" !? "safer", not "more safe"  
225                      2010/03/10   the loop starts  
226                      2010/03/17   corr. t^ete; set up -> setup for  
227                      2010/03/18   TODO EOF, ctan.org/pkg/newfile; non-ASCII  
228                      2010/03/19   extended description of \MakeExpandableAll...;  
229                      '' -> "  
230                      2010/03/20   \ctanpkgref  
231                      2010/03/22   \StartFDsetup, \Prepend...  
232                      2010/03/23   URL for 'substr.sty'  
233  
234   TODO: cleveref 2010/03/18  
235