

# fifinddo

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

Uwe Lück<sup>†</sup>

November 17, 2012

*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@**.

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

## Contents

<b>1</b>	<b>Introduction: The Gnome of the Aim</b>	<b>3</b>
1.1	Parsing by T <sub>E</sub> X—are you mad? . . . . .	3
1.2	Useful for . . . . .	4
1.2.1	Comparisons . . . . .	5
1.3	For insiders . . . . .	5

---

\*This file describes version v0.61 of **fifinddo.sty** as of 2012/11/17.

<sup>†</sup><http://contact-ednotes.sty.de.vu>

<i>CONTENTS</i>	2
<b>2 Preliminaries</b>	<b>6</b>
2.1 Head of file (Legalese) . . . . .	6
2.2 Format and package version . . . . .	7
2.3 Category codes . . . . .	7
<b>3 File handling</b>	<b>8</b>
3.1 Opening, Writing to, Closing Output . . . . .	8
3.2 Processing Input . . . . .	9
3.3 A Combining Shorthand . . . . .	11
<b>4 Basic handling of substring conditionals</b>	<b>12</b>
4.1 “Substring Theory” . . . . .	12
4.2 Plan for proceeding . . . . .	13
4.3 Meta-Setup . . . . .	14
4.4 Setup for conditionals . . . . .	14
4.5 Setup for sandboxes . . . . .	15
4.6 Getting rid of the tildes . . . . .	17
4.7 Alternative Setup . . . . .	17
4.8 Calling conditionals . . . . .	18
4.9 Copy jobs . . . . .	18
<b>5 Programming tools</b>	<b>19</b>
5.1 Tails of conditionals . . . . .	19
5.2 Line counter . . . . .	20
5.3 “Identity job” LEAVE and “default job” * . . . . .	21
<b>6 Setup for expandable chains of replacements</b>	<b>21</b>
6.1 The backbone macro . . . . .	22
6.2 The basic setup interface macro . . . . .	22
6.3 Automatic chaining . . . . .	23
6.4 CorrectHook launching the replacement chain . . . . .	25
<b>7 Leave package mode</b>	<b>25</b>
<b>8 Pondered</b>	<b>25</b>
<b>9 VERSION HISTORY</b>	<b>26</b>

# 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, I have seen this with PCT<sub>E</sub>X. 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 T<sub>E</sub>X macros, knowing just T<sub>E</sub>X (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 T<sub>E</sub>X 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 T<sub>E</sub>X packages; and while I am typing this, I find at least 14 other similar packages in Jürgen Fenn’s *Topic Index* of the *T<sub>E</sub>X 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 T<sub>E</sub>X—yes, celebrate!)

Actually, T<sub>E</sub>X’s mechanism of collecting macro arguments is hard-wired parsing at quite a high level. L<sup>A</sup>T<sub>E</sub>X hides this from “simple-minded” users by a convention *not* to use that full power of T<sub>E</sub>X for *end-user macros*. Internally, L<sup>A</sup>T<sub>E</sub>X *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<sup>A</sup>T<sub>E</sub>X’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>2</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.

<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.

<sup>2</sup><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.<sup>3</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>4</sup> This could replace packages like `easylatex`,<sup>5</sup> `txt2latex`,<sup>6</sup> `txt2tex`<sup>7</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;

---

<sup>3</sup>If you know the “names” of the encodings, Heiko Oberdiek's `stringenc` may be preferable.

<sup>4</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>5</sup><http://ctan.org/pkg/easylatex>

<sup>6</sup><http://ctan.org/pkg/txt2latex>

<sup>7</sup><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`,<sup>8</sup> `ted`,<sup>9</sup> `xstrings`<sup>10</sup> (as I understand them, perhaps also `coolstr`<sup>11</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>12</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  $\TeX$ ’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.

---

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

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

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

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

<sup>12</sup>There is more in my badly implemented `txtproc.sty`.

That `fifinddo` acts on “ $\text{\TeX}$ (t)” files or so means that (at present) I think of applications on “plain text” files which will usually be  $\text{\TeX}$  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  $\text{\TeX}$  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” ( $\text{\TeX}$  *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)

```

1  %% Macro package 'fifinddo.sty' for LaTeX2e,      %% FIDO, FIND!
2  %% copyright (C) 2009-2012 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{0.61} \def\filedate{2012/11/17}
9
10 %% This file can be redistributed and/or modified under
11 %% the terms of the LaTeX Project Public License; either
```

```

12  %% version 1.3c 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  \RequirePackage{stacklet} \PushCatMakeLetter\_           %% 2012/08/27
    v0.6 moves definition of \MakeActiveDef to actcodes in the catcodes bundle:
29  \RequirePackage{actcodes}

```

fifinddo uses T<sub>E</sub>X’s feature of delimited macro parameters for matching *strings* of characters. In order to get expected results, *category codes* must match as well. `\PatternCodes` is used in setup macros for reading patterns for this purpose. It defaults to `\fdPatternCodes`. We offer `\SetPatternCodes{<commands>}` for redefining `\PatternCodes` and `\ResetPatternCodes` for returning to `\fdPatternCodes`, so setup scripts such as `mdoccorr.cfg` have shorter lines.

```

30  \newcommand*{\fdPatternCodes}{\MakeOther\&\MakeOther\$}
31  \newcommand*{\SetPatternCodes}{\def\PatternCodes}
32  \newcommand*{\ResetPatternCodes}{\let\PatternCodes\fdPatternCodes}
33  \newcommand*{\PatternCodes}{\ResetPatternCodes}
34  %% 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\[\]`:

```

35 \newcommand*{\Delimiters}[2]{%
36   \MakeOther{\MakeOther\}\CatCode#1\@ne \CatCode#2=\tw@}

```

`\CatCode\⟨char⟩` is introduced for v0.6. It should appear in a package `manycats` of the `catcodes` bundle, but that’s too much for 2012/08/27—[TODO](#).

```

37 \providecommand*{\CatCode}{\catcode'}

```

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?):

```

38 \newcommand*{\PercentChar}{%} \let\PercentChar\@percentchar
39 \newcommand*{\BackslashChar}{%} \let\BackslashChar\@backslashchar

```

`\BasicNormalCatCodes` restores Plain T<sub>E</sub>X’s **macro parsing** and comment character:

```

40 \newcommand*{\BasicNormalCatCodes}{%
41   \CatCode\\\z@ \Delimiters\{\}%
42   % \restorecr !?
43   \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.

`\BasicNormalCatCodes` may be reimplemented by a package `normcats` in the `catcodes` bundle—[TODO](#) ...

## 3 File handling

Peter Wilson’s `newfile` provides more powerful file handling.

### 3.1 Opening, Writing to, Closing Output

```

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

```

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

```

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

```

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

```

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

```



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

```

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

`\CloseResultFile` closes  $\langle output \rangle$ :

```

56 \newcommand*\CloseResultFile{%
57   \immediate\closeout\result_file
58   \typeout{'fifinddo'                %% 2011/10/23
59           \space\space\space        %% 2011/10/26
60   closing '\result_file_name'}}
```

### 3.2 Processing Input

`\ProcessFileWith[ $\langle changes \rangle$ ]{ $\langle input \rangle$ }{ $\langle loop-body \rangle$ }` opens a file  $\langle input \rangle$  and runs a loop on its lines the main body of which is  $\langle loop-body \rangle$ . When the  $\langle loop \rangle$  starts, a new line of  $\langle input \rangle$  is stored as macro `\fdInputLine`. The optional argument  $\langle changes \rangle$  may change category codes used in reading  $\langle input \rangle$ . 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  $\langle changes \rangle$  in another macro  $\langle macro \rangle$  and to call `\ProcessFileWith[ $\langle macro \rangle$ ]{ $\langle input \rangle$ }{ $\langle loop-body \rangle$ }`). More possible uses of some usual T<sub>E</sub>X category codes may be (some of)

- avoiding matching substrings of control words,
- skipping blank spaces as T<sub>E</sub>X does it usually,
- catching *balanced* input pieces,
- ignoring comments,
- ignoring certain characters.

```

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

v0.5: Variant of L<sup>A</sup>T<sub>E</sub>X's `\IfFileExists`—failed so far because I had omitted the blank space:

```

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

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

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

... switching into “plain text mode”; from docstrip.tex:

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

← cf. T<sub>E</sub>Xbook pp. 43ff., 368ff., “extended keyboards”, up-/downarrow; →  
“math specials”, cf. “space specials”

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

With v0.31, we support non-ASCII:

```
74      \count@=128
75      \loop
76      \ifnum\count@<\ccclvi
77      \catcode\count@=12
78      \advance\count@\@ne
79      \repeat
80      #1%
81      \loop \ifeof\@inputcheck \else
82      \read\@inputcheck to \fdInputLine
83      \ignorespaces #3%
```

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

```
84      \expandafter \let
85      \expandafter \IfFDpreviousInputEmpty
86      \ifx\fdInputLine\@empty \@firstoftwo
87      \else \@secondoftwo \fi
88      \repeat
89      \endgroup
90      \fi
91      \closein\@inputcheck
```

Added for v0.5, cf. Sec. 3.3

```
92      \ifFinalInputFile
93      \CloseResultFile \FinalInputFilefalse
94      \fi
95      }
```

`\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.

```

96 \newcommand*\CopyFile}{%
97   \@ifstar{\let\FD@copy@style\FD@compress@voids \FD@copyfile}%
98   {\let\FD@copy@style\CopyLine \FD@copyfile}}
99 \newcommand*\FD@copyfile}[2] [] {%
100   \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`:

```

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

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

```

102 \newcommand*\FD@compress@voids}{%
103   \IfFDinputEmpty{\IfFDpreviousInputEmpty\relax\CopyLine}%
104   \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  $\text{\TeX}$  run).

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

```

105 \newif\ifFinalInputFile
106 \newcommand*\ProcessFinalFileWith}{%
107   \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  $\text{\TeX}$  macro, say,  $\text{\find}$  whose parameter text (cf.  $\text{\TeX}$ book p. 203) starts with  $\text{\#1}\langle pattern \rangle\text{\#2}\&$  stops  $\text{\TeX}$  with an error if it does not find  $\langle pattern \rangle$  and then  $\&$ . Otherwise we have a situation  $\text{\find}\langle split1 \rangle\langle pattern \rangle\langle split2 \rangle\&$ , and  $\text{\find}$  reads  $\langle split1 \rangle$  as  $\text{\#1}$  and  $\langle split2 \rangle$  as  $\text{\#2}$ . An important point to note is that  $\langle split1 \rangle$  will not contain  $\langle pattern \rangle$ , but possibly  $\langle pattern \rangle$  has more occurrences in  $\langle split2 \rangle$ . In this sense,  $\text{\find}$  uses the *first* occurrence of  $\langle pattern \rangle$  it finds in order to delimit  $\text{\#1}$ . Finding the *last* occurrence of  $\langle pattern \rangle$  therefore needs a special idea.

In order to use  $\text{\find}$  for a test whether  $\langle pattern \rangle$  is in  $\langle target \rangle$ , we build a “sandbox”  $\text{\find}\langle sand \rangle\&$ , where  $\langle sand \rangle$  contains  $\langle target \rangle$  and additionally  $\langle pattern \rangle$ —as a “dummy;” so  $\&$  delimits the search and  $\text{\find}$  finds  $\langle pattern \rangle$  either in  $\langle target \rangle$  or somewhere else before  $\&$ .

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

If  $\text{\#2}$  is *not* empty,  $\langle pattern \rangle$  occurs in  $\langle target \rangle$ —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,<sup>15</sup> try (if that version still is installed)<sup>16</sup>

```
\IfSubStringInString{\str1}\str2\str1\}{\str1}\str2\}{YES}{NO}
```

which works *verbatim* as well as considering  $\langle str1 \rangle$  and  $\langle str2 \rangle$  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)
```

<sup>13</sup>This is just as wrong as the next claim.

<sup>14</sup>Due to the special substrings to test in this application, this is true although the surrounding claims are wrong.

<sup>15</sup><http://ctan.org/pkg/substr>. `substr` does not change category codes as `fifinddo` does and uses  $\text{\@nil}$  as delimiter instead of our  $\&$ .

<sup>16</sup>The “feature” has been fixed with v1.2 as of 2009/10/20 of `substr.sty`.

<sup>17</sup>Likewise  $\text{\t}\text{\^ete}\text{\-}\text{\^a}\text{\-}\text{\t}\text{\^ete}$ ...

<sup>18</sup>Polynesian: `aku aku`, `rongorongo`, `wiki wiki`...

or `\IfSubStringInString{ionization}{ionizat}{YES}{NO}`.<sup>19</sup> Same with L<sup>A</sup>T<sub>E</sub>X's internal `\in@`.<sup>20</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 7, `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>21</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>22</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>19</sup>Read `substr.sty` or try “normal” things to convince yourself that the syntax indeed is `\IfSubStringInString{<pattern>}{<target>}{<yes>}{<no>}`.

<sup>20</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>21</sup>Must? Actually, I preferred this solution to other ideas like measuring the length of  $\langle split2 \rangle$ .

<sup>22</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  $\boxed{\backslash\text{fdParserId}}$ . The default for  $\langle changes \rangle$  is *nothing*.

```

108 \newcommand*\backslashStartFDsetup}[1]{%
109     \let\FD_do_setup#1%
110     \afterassignment\FD_prepare_pattern
111     \def\fdParserId}
112 \newcommand*\backslashFD_prepare_pattern}[1][ ]{%
113     \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  $\langle setup\text{-}cmd \rangle$ .—  
With v0.5, we learn from the previous and provide

$$\boxed{\backslash\text{MakeSetupCommand} \{ \langle setup\text{-}cmd \rangle \} \{ \langle do\text{-}setup \rangle \} [ \langle args \rangle ] \{ \langle action \rangle \} }$$

```

114 \newcommand*\backslashMakeSetupCommand}[2]{\newcommand*#1{\backslashStartFDsetup#2}%
115     \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.

```

116 \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.

```

117 \MakeSetupCommand{\MakeSubstringConditional}{\mk_substr_cond}[1]{%
118   %% #1 pattern string
119   \endgroup \@namedef{\substr_cond \fdParserId}##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.<sup>23</sup> 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

<sup>23</sup>Loosely speaking of “the parser” `⟨parser⟩` around here somehow refers to this macro—but rather to its “parameter text” only, according to T<sub>E</sub>Xbook 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 T<sub>E</sub>X 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.

```
120 \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.

```
121 \MakeSetupCommand{\MakeSetupSubstringCondition}
122 \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.

```
123 %% #1 pattern string
124 %% #2 additional arguments, e.g., '{#1}' to keep tested string
125 \endgroup
126 \expandafter \protected@edef %% protected 2011/11/21

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

```
127 \csname \setup_substr_cond \fdParserId \endcsname ##1{%
128 \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.

```
129 ##1\FD_noexpand~#1\dollar_tilde&#2}%

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

130 \let\dollar_tilde\sandbox_dollar}
```

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

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

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

```
133 % \def\make_not_expanding_cs#1{%
134 % \expandafter \noexpand \csname #1\endcsname}

← 2011/11/20 →

135 \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 `\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)

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

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

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

`\RemoveDummyPatternArg<macro>{<arg>}` executes `\RemoveDummyPattern` in the next argument:

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

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

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

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

```
141 \newcommand*{\RemoveTildeArg}[2]{%
142 \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):

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

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

```
145 \let\RemoveTilde\FD_remove_normal_tilde}
146 \let\FD_remove_normal_tilde\RemoveTilde
147 \FDnormalTilde
148 \newcommand*{\FDpseudoTilde}{%
149 \def\FD_noexpand~{\noexpand\pseudo_tilde}% %% v0.51
150 \let\RemoveTilde\FD_remove_pseudo_tilde}
151 \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.

```
152 \newcommand*\ProcessStringWith}[2]{%
153   \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.

```
154 \newcommand*\ProcessExpandedWith}[2]{%
155   \csname \setup_substr_cond #2\expandafter \endcsname
156   \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`:

```
157 \newcommand*\ProcessInputWith}[1]{%
158 %   \csname \setup_substr_cond #1\expandafter \endcsname
159 %   \expandafter{\fdInputLine}}
160 % %% (Definition almost copied for efficiency.)
161 \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>`.

```

162 \newcommand*{\CopyFDconditionFromTo}[2]{%
163   \expandafter \let
164     \csname \setup_substr_cond #2\expandafter \endcsname
165     \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).

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

```

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

```

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

```

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

```

`\IfFDdollar{ $\langle arg \rangle$ }{ $\langle when-dollar \rangle$ }{ $\langle when-not-dollar \rangle$ }` 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):

```

172 \newcommand*{\IfFDdollar}[1]{%
173   \ifx$#1\expandafter \@firstoftwo \else
174     \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:

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

—which is same as

```
176 \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 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.

```
177 \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.

```
178 \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 *<, =, >*.

```

179 \newcommand*{\IfInputLine}[1]{%
180   \ifnum\c@fdInputLine#1\relax \expandafter \@firstoftwo
181   \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:

```

182 \expandafter \let
183   \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:

```

184 \@namedef{\setup_substr_cond*}{%
185   \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  $\langle parser \rangle$  and receives from the latter  $\langle split1 \rangle = \#1$  and  $\langle split2 \rangle = \#2$ .  $\#3$  is the result of what happened so far.

```

186 \def\repl_all_chain_expandable#1#2#3#4#5#6{%
187   %% #1, #2 splits, #3 past,   #4 substitute,
188   %% #5 repeat parser,        #6 pass to
189   %   \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:

```

190   \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  $\langle pattern \rangle$  from  $\langle parser \rangle$  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  $\sim$  is removed,  $\#6$  inserts a new wrap for the new dummy pattern.

```

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

```

Otherwise ... the *sandbox builder*  $\langle sandbox \rangle$  (that will be shown below) that called  $\langle parser \rangle$  initialized  $\#5$  to be that  $\langle parser \rangle$  itself. ( $\langle parser \rangle$  otherwise wouldn’t know who it is.) So  $\langle parser \rangle$  calls itself with another sandbox  $\#2\&$ . Note that  $\#2$  contains ‘ $\sim\langle pattern \rangle\$$ ’ due to the initial  $\langle sandbox \rangle$  building.

```

192   {#5#2&{#3#1#4}{#4}#5#6}}

```

$\#4$  is the replacement string that  $\langle sandbox \rangle$  passed to  $\langle parse \rangle$ . The first argument after the  $\&$  is previous stuff plus the recently skipped  $\langle split1 \rangle$  plus  $\#4$  replacing the string  $\langle pattern \rangle$  that was matched.

Finally,  $\#5$  and  $\#6$  again “recall”  $\langle parser \rangle$  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  $\langle id \rangle$  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  $\langle chng \rangle$  changes category codes locally for reading  $\langle find \rangle$  and  $\langle replace \rangle$ .

```

193 \MakeSetupCommand{\MakeExpandableAllReplacer}
194         {\mk_setup_xpdbl_all_repl}[3]{%
195     %% #1 pattern, #2 substitute, #3 pass to
196     \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 ...

```

197     \protected@edef\@tempa{%                %% protected 2011/11/21
198         \noexpand\mk_setup_substr_cond{#1}{%
199             {#2}%
200         \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.

```

201         \noexpandcsname \substr_cond \fdParserId \endcsname
202         \noexpand\noexpand
203         \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`.

```

204     \begingroup \@tempa
205     \begingroup \mk_substr_cond{#1}{%
206         \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.

```

207 \newcommand*\PrependExpandableAllReplacer{%
208     \let\fdParserId_before\fdParserId
209     \ifstar{\stepcounter{fd_line_job}%
210         \edef\@tempa{%
211             \noexpand\StartFDsetup
212             \noexpand\prep_xpdbl_all_repl
213             {\number\value{fd_line_job}}}%
214         \@tempa}%
215     {\StartFDsetup\prep_xpdbl_all_repl}}
216 \newcommand*\prep_xpdbl_all_repl}[2]{%
217     \mk_setup_xpdbl_all_repl{#1}{#2}{\fdParserId_before}}%
218     %% #1 pattern, #2 substitute, #3 pass to

```

As of 2012/11/13, mdoccorr.cfg uses \do as a shorthand for

```
\PrependExpandableAllReplacer*
```

\do sometimes has not been defined at that moment, then

```
\renewcommand*\do{\PrependExpandableAllReplacer*}
```

fails. To make it working (or superfluous):

```
219 \gdef\do{\PrependExpandableAllReplacer*}
```

`\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).

```

220 \newcommand*\StartPrependingChain{%
221     \def\fdParserId{LEAVE}%
222     \setcounter{fd_line_job}{0} %% 2011/11/13; TODO \Nameprefix...
223     \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*.

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

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

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

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

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

## 7 Leave package mode

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

```
230 \PopLetterCat\_                               %% 2012/08/27
231 \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

```
232 %% TODO abbreviated commands (aliases) \MkSubstrCond...
233 %% TODO \@onlypreamble!?
234 \newcommand*{\ApplySubstringConditional}[1]{%
235     %% #1 identifier; text to be searched expected next
236     \csname setup_substr_cond:#1\endcsname}
237 \newcommand*{\ApplySubstringConditionalToExpanded}[1]{% 2009/03/31+
238     \csname setup_substr_cond:#1\expandafter \endcsname \expandafter}
239 \newcommand*{\ApplySubstringConditionalToInputString}[1]{% 2009/03/31+
240     \csname setup_substr_cond:#1\expandafter \endcsname
241     \expandafter {\fdInputLine}}
242 %% TODO or ‘\OutputString’, even ‘\read’ to ‘\OutputString’!?
243 % \newcommand*{\ApplySubstringConditionalToExpanded}[2]{%
244 %     %% note: without assignments, robust!
245 %     %% BUT the ‘\csname ... \expandafter \endcsname’ method is faster
246 %     \expandafter \reversed_apply_substr_cond
247 %     \expandafter {#2}{#1}}
```

```

248 % \newcommand*\reversed_apply_substr_cond}[2]{%
249 %   \ApplySubstringConditional{#2}{#1}}
250 %% ODER:
251 % \newcommand*\expand_attach_arg}[2]{%% 2009/03/31
252 %   %% #1 command with previous args, TODO cf. LaTeX3
253 %   \expandafter \attach_arg \expandafter {#1}{#2}}
254 %   %% actually #1 may contain more than one token,
255 %   %% only first expanded
256 % \newcommand*\attach_arg}[2]{#2{#1}}
257 % \newcommand*\ApplySubstringConditionalToExpanded}[2]{%
258 %   \expandafter \attach_arg \expandafter
259 %     {#2}{\ApplySubstringConditional{#1}}}

```

## 9 VERSION HISTORY

260	v0.1	2009/04/03	very first version, tested on morgan.sty
261	v0.2	2009/04/05	counter fdInputLine, \ProvidesFile moved from
262			\ProcessFile to \ResultFile, \CopyFD...,
263			category section first, more sectioning,
264			suppressing empty code lines before section
265			titles; discussion, \Delimiters
266		2009/04/06	more discussion
267		2009/04/07	more discussion, factored \WriteProvides out from
268			\ResultFile, \ProcessExpandedWith corrected
269		2009/04/08	\InputString -> \fdInputline;
270			removed \ignorespaces
271		2009/04/09	\WhenInputLine[2] -> \IfInputline[3],
272			\ProcessInputWith, typos,
273			\WriteProvides message 'with'
274		2009/04/10	\make_not_expanding_cs
275			DISCOVERED 'IF SUBSTRING' ALGORITHM WRONG
276			(<str1><str2><str1> in <str1><str2>)
277	v0.3	2009/04/11	SOME THINGS GIVEN UP EARLIER WILL BE REMOVED,
278			TO BE STORED IN THE COPY AS OF 2009/04/10
279			mainly: sandbox setup (tilde/dollar)
280			REAL ADDITION: setup for expandable replacing
281		2009/04/12	played with 'chain' vs. 'sequence';
282			plain '...', 'cf.', 'etc.' for 'mdcorr.cfg'
283		2009/04/13	\RemoveTilde...
284		2009/04/15	reworked text, same mistake \in@
285	v0.31	2009/04/21f.	comments on ted, stringstrings
286		2009/12/28	"onwards)" !? "safer", not "more safe"
287		2010/03/10	the loop starts
288		2010/03/17	corr. t^ete; set up -> setup for
289		2010/03/18	TODO EOF, ctan.org/pkg/newfile; non-ASCII
290		2010/03/19	extended description of \MakeExpandableAll...;
291			' -> "
292		2010/03/20	\ctanpkgref
293		2010/03/22	\StartFDsetup, \Prepend...

294           2010/03/23   URL for 'substr.sty'  
 295   SENT TO CTAN  
 296  
 297   v0.4   2010/03/24   removed \pagebreak before "substrings";  
 298                        <relation> with \IfInputLine precisely  
 299           2010/03/25   todo \ProcessExp... more precisely, etc.  
 300           2010/03/26   ... was wrong, removed  
 301           2010/03/29   \SetPatternCodes, \ResetPatternCodes,  
 302                        \SetCorrectHookJob, \SetCorrectHookJobLast;  
 303                        <relation> with \HardNVerb;  
 304                        don't mention \begingroup with  
 305                        \mk\_setup\_substr\_cond; renamed v0.4  
 306   belonged to nicetext RELEASE 0.4  
 307   v0.4a 2010/04/04   copyright 2010  
 308   belonged to nicetext RELEASE 0.41  
 309  
 310   v0.41 2010/04/06   more on \ProcessExpanded...;  
 311                        \ProcessFile... gets opt arg  
 312           2010/04/13   \ProcessFile{<file>}... shows <file>  
 313   used by blog.sty v0.1, v0.2  
 314   v0.42 2010/11/09   typo corr.  
 315           2010/11/10   \IfFDpreviousInputEmpty  
 316           2010/11/11   \BasicNormalCatcodes from blog.sty,  
 317                        \CopyFile\*, \CopyLine; v3. -> v0.3;  
 318                        LPPL v1.3c  
 319           2010/11/12   \CatCode replaced (implemented in niceverb only)  
 320           2010/11/13   \CopyFile with \message{.}  
 321           2010/11/24   reworked doc. of replacement setup;  
 322                        \StartPrependingChain  
 323           2010/11/25   corr. typo \@backslash...; doc. changes;  
 324                        \CopyLine indeed, not \fdCopyLine  
 325           2010/11/27   footnote on "parser", other doc. corr.s  
 326           2011/01/20   corr. "period" AMSTERDAM  
 327           2011/01/25   updated (C); footnotes to 'substring theory';  
 328                        TODO with \RemoveTilde; some manual line spacings  
 329                        (adding '\ ')  
 330                        -> r0.42  
 331   v0.43 2011/08/06   doc.: mistake \WriteResult/\ResultFile,  
 332                        use \acro  
 333           2011/09/12f. \CorrectedInputLine - reworded for breaking  
 334                        -> r0.44  
 335   v0.44 2011/10/23   messages from \ResultFile and \CloseResultFile  
 336                        -> r0.46  
 337   v0.45 2011/10/26   little modification of \CloseResultFile message,  
 338                        "sacrificing" \pagebreak before Sec. 4 -- fine!  
 339  
 340   v0.5   2011/11/13   \PrependExpandableAllReplacer\*, \MakeSetupCommand;  
 341                        doc.: {center} with too long verbatim quote  
 342           2011/11/19   input check fixed, doc. there adjusted,  
 343                        \ifFinalInputFile, \ProcessFinalFileWith, job '\*'

```
344          2011/11/20  \noexpandcsname; "default job" lowercase
345          2011/11/21  \ProcessInputWith "less efficient",
346                      \protected@edef, "pseudo-tilde"
347          -> r0.5
348 v0.51  2012/01/20  updated (C); TODO on pseudo-tilde
349          2012/03/17  fixed \FDnormalTilde/\FDpseudoTilde
350          -> r0.53
351 v0.6   2012/08/27  using 'catcodes', introducing \CatCode
352          -> r0.62
353 v0.61  2012/11/17 . \gdef\do...
354
355 TODO: cleveref 2010/03/18
356
```