fifinddo

—

Filtering TEX(t) Files by TEX

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

April 16, 2009

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

Abstract

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

Contents

1 Introduction: The Gnome of the Aim

1.1 Parsing by TEX—are you mad?

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

- It works instantly on any TEX installation. (*Restrictions:* Some TEX versions `\write` certain hex codes for certain characters, cf. TEXbook p. 45, I have seen this with PCTEX. However, some applications of fifinddo are nothing but technical steps where you will read the result files rarely anyway. And I have not yet explored extended encodings.)
- You can apply and customize it like any TEX macros, knowing just TEX (or even only the documentation of some user-friendly extension of fifinddo), without the need of learning any additional script language.

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

- The syntax of usual utilities (e.g., "wildcards") is sometimes difficult with *TEX* files with all their backslashes, square brackets, stars, question marks ...

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

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

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

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

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

1.2 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.[2]
- 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.[3] This could replace packages like easylatex, txt2latex, txt2tex in a customizable way, using, e.g., the "correct" hook from makedoc.sty as exemplified in mdcorr.cfg (2009/04/12, see examples section of makedoc.pdf). Section 6 provides setup for such macros.
- as to easylatex again, *lists* could be detected and transformed into LaTeX list commands. This could re-implement the lists functionality of wiki.sty that is somewhat dangerous.
- introduce your own *shorthands* to be expanded not as TeX macros, but by text substitution;
- in EPS files, copy the bounding box to the top two lines for the graphics bundle.

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.

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

[3] But what when a new sentence is starting indeed? Well, `cf.` is an easier example.—`etc.` even showed a problem in niceverb. mdcorr.cfg replaces `etc.` only, so you can keep the extra space by a code line break (2009/04/11)

1.2.1 Missing

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, ted, xstrings (as I understand them, perhaps also coolstr) 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.[4] I am just implementing what I actually need and what could show that this approach is worth being pursued. It may need being sponsored or otherwise supported.

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.

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

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

2. There are exact or safe tests especially of *empty macro arguments* that are "expandable," i.e., they are "robust," don't need assignments, can be executed in `\write`ing and in `\edef` definitions. "Usually," the safe way to test emptiness is storing a macro argument as a macro, say `\tempo`, in order to test `\ifx\tempo\empty` where `\empty` has been defined by `\def\empty{}` in the format. But this requires some `\def\tempo{#`$\langle n\rangle$`}` which breaks in "mere expanding" (TEX *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.

[4] There is more in my badly implemented txtproc.sty.

Once there may be an option to read input with some usual TEX category codes as well, it may be useful to (some of)

- avoid matching substrings of control words,
- skip blank spaces as TEX 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)

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

\def\fileversion{0.3} \def\filedate{2009/04/15}

%% This file can be redistributed and/or modified under
%% the terms of the LaTeX Project Public License; either
%% version 1.3a of the License, or any later version.
%% The latest version of this license is in
%%
%%     http://www.latex-project.org/lppl.txt
%%
%% We did our best to help you, but there is NO WARRANTY.
%% Please report bugs, problems, and suggestions via
%%
%%   http://www.contact-ednotes.sty.de.vu
%%
%% For the full documentation, look for 'fifinddo.pdf'.
%% Its source starts in 'fifinddo.tex'.
```

2.2 Format and package version

```
\NeedsTeXFormat{LaTeX2e}[1994/12/01]
% 1994/12/01: \newcommand* etc.
```

```
\ProvidesPackage{fifinddo}[\filedate\space v\fileversion\space
                           filtering TeX(t) files by TeX (UL)]
```

2.3 Category codes

We use the "underscore" as "compound identifier."

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

`\MakeOther` is a synonym for `\@makeother`, needed for matching special characters from the input file. It is exemplified by `\fdPatternCodes` which is the default of `\PatternCodes`. The latter is used in setup macros for reading patterns.

```
\@ifundefined{MakeOther}{\let\MakeOther\@makeother}{}
\newcommand*{\fdPatternCodes}{\MakeOther\&\MakeOther\$}
\newcommand*{\PatternCodes}{} \let\PatternCodes\fdPatternCodes
  %% 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\[\]`:

```
\newcommand*{\Delimiters}[2]{%
  \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 LaTeX constructs—here are copies `\PercentChar` and `\BackslashChar` of them (do you need more?):

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

3 File handling

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

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

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

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

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

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

```
\newcommand*{\WriteProvides}{%
  \WriteResult{%
    \string\ProvidesFile{\result_file_name}%
      [\the\year/\two@digits\month/\two@digits\day\space
       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 it starts, a new line of ⟨*input*⟩ is stored as macro `\fdInputLine`.

```
\newcommand*{\ProcessFileWith}[2]{%
  \openin\@inputcheck=#1%
%   \ifeof\@inputcheck  %% bad 'exists?' test
%     \PackageError{fifinddo}{File '#1' not here}%
%                           {Mistyped?}%
%   \else
    \global\c@fdInputLine=\z@ %% line counter reset
    \begingroup
      \MakeOther\{\MakeOther\}\@sanitize
      %% from docstrip.tex:
      % \MakeOther\^^A\MakeOther\^^K%% irrelevant, not LaTeX
      \endlinechar\m@ne
      %% <- cf. TeXbook "extended keyboards" up-/downarrow
      %%    -> "math specials", cf. "space specials"
      \MakeOther\^^I% ASCII horizontal tab -- guessed!? ^^L!?
      \loop \ifeof\@inputcheck \else
        \read\@inputcheck to \fdInputLine
        \ignorespaces #2%
      \repeat
    \endgroup
%   \fi
  \closein\@inputcheck}
```

`\CloseResultFile` closes ⟨*output*⟩.

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

Peter Wilson's newfile provides more powerful file handling.

4 Basic handling of substring conditionals

4.1 "Substring Theory"

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

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

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

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

If `#2` is *not* empty, ⟨*pattern*⟩ occurs in ⟨*target*⟩—or this once was *thought*, some time in developping the present package, as well as in the version of substr.sty marked `2005-11-29`,[5] try

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

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

`\IfSubStringInString{day␣after␣day}{day␣after␣}{YES}{NO}`[6]
`\IfSubStringInString{AMSTERDAM}{AMSTERD}{YES}{NO}`
`\IfSubStringInString{TORONTO}{TORON}{YES}{NO}`
`\IfSubStringInString{bonbon}{bon}{YES}{NO}`[7]
`\IfSubStringInString{bonobo}{bono}{YES}{NO}` (an ape)

or `\IfSubStringInString{ionization}{ionizat}{YES}{NO}`.[8] Same with LATEX's internal `\in@`:

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

[5] substr does not change category codes and uses `\@nil` as delimiter instead of our `&`.
[6] Likewise `t\^ete-\‘a-t\^te` ...
[7] Polynesian: `aku aku`, `rongorongo`, `wiki wiki` ...
[8] Read substr.sty or try "normal" things to convince yourself that the syntax indeed is `\IfSubStringInString{`⟨*pattern*⟩`}{`⟨*target*⟩`}{`⟨*yes*⟩`}{`⟨*no*⟩`}`.

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

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

4.2 Plan for proceeding

When we check a file for several patterns, we seem to need *two* macros for each pattern: one that has the pattern in its parameter text and one that stores the pattern for building the sandbox.[9] 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.

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 Set up conditionals

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

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

[9] 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 ...

`\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. It may be more safe to redefine `\PatternCodes` instead.

```
\newcommand*{\MakeSubstringConditional}{%
  \afterassignment\mk_substr_cond_san \def\cond_id}
\newcommand*{\mk_substr_cond_san}[1][]{%
  \begingroup \PatternCodes #1\mk_substr_cond}
  %% #1 more changes
```

`\begingroup` `\mk_substr_cond{`⟨*pattern*⟩`}` can be directly called by other programmer setup commands when `\cond_id` and ⟨*pattern*⟩ have been read.

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

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

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

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

4.4 Set up 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.

```
\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.

```
\newcommand*{\MakeSetupSubstringCondition}{%
  \afterassignment\mk_setup_substr_cond_san \def\cond_id}
\newcommand*{\mk_setup_substr_cond_san}[1][]{%
  \begingroup \PatternCodes #1\mk_setup_substr_cond}
```

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

```
\def\mk_setup_substr_cond #1#2{%% #1 pattern string,
  %% #2 additional arguments, e.g., `{#1}' to keep tested string
  \endgroup
  \expandafter \edef
   \csname \setup_substr_cond \cond_id \endcsname ##1{%
%       \expandafter \noexpand
%         \csname \substr_cond \cond_id \endcsname %% 2009/04/10:
        \make_not_expanding_cs{\substr_cond \cond_id}%
```

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

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

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

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

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

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

4.5 Getting rid of the tildes

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

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

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

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

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

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

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

```
102 % %% An alternative policy is to pass
103 % %% <target> (as an argument) to the parsing macro.
104 \newcommand{\RemoveTilde}{} \def\RemoveTilde#1~{#1}
```

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

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

4.6 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.

```
107 \newcommand*{\ProcessStringWith}[2]{%
108   \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.

```
109 \newcommand*{\ProcessExpandedWith}[2]{%
110   \csname \setup_substr_cond #2\expandafter \endcsname
111     \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`:

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

(Definition almost copied for efficiency.)

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

4.7 Copy jobs

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

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

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

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

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

5 Programming tools

5.1 Tails of conditionals

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

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

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

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

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

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

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

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

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

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

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

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

5.2 Line counter

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

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

```
\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.

```
\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 `=`.

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

```
\expandafter \let
  \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.

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

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

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

If `#2` starts with `$`—with category code 3, "math shift"!, it *is* `$`, due to not reading `$` from input with its standard category code 3 and the sandbox construction (where `$` appears with its standard category code). And this is the

case *exactly* when the ⟨*pattern*⟩ from ⟨*parser*⟩ didn't match, again due to the input category codes. Now on *no* match, the sandbox builder `#6` is called with target string `#3#1` where the last tested string is attached to previous results. The ending `~` is removed, `#6` inserts a new wrap for the new dummy pattern.

```
141                 {\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.

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

```
143   \newcommand*{\MakeExpandableAllReplacer}{%
144     \afterassignment\mk_setup_xpdbl_all_repl_san
145     \def\cond_id}
```

... usual intermezzo for reading patterns with non-standard category codes, this time we read *two* patterns ...

```
146   \newcommand*{\mk_setup_xpdbl_all_repl_san}[1][]{%
147     \begingroup \PatternCodes #1\mk_setup_xpdbl_all_repl}
```

Here comes the real work.

```
148   \newcommand*{\mk_setup_xpdbl_all_repl}[3]{%
149     %% #1 pattern, #2 substitute, #3 pass to
150     \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 ...

```
151       \edef\@tempa{%
152         \noexpand\mk_setup_substr_cond{#1}{%
153           {}{#2}%
154           \noexpand\noexpand
```

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

```
155             \make_not_expanding_cs{\substr_cond\cond_id}%
156           \noexpand\noexpand
157             \make_not_expanding_cs{\setup_substr_cond #3}}}%
```

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

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

The final command is the one that we explained first.

Support for dozens of replacements in one sequence and for screen messages must wait for another release, sorry!

7 Leave package mode

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

```
161   \catcode`\_=8   %% restores underscore use for subscripts
162
163   \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

```
164     %% TODO abbreviated commands (aliases) \MkSubstrCond...
165     %% TODO \@onlypreamble!?
166   \newcommand*{\ApplySubstringConditional}[1]{%
167     %% #1 identifier; text to be searched expected next
168     \csname setup_substr_cond:#1\endcsname}
169   \newcommand*{\ApplySubstringConditionalToExpanded}[1]{% 2009/03/31+
170     \csname setup_substr_cond:#1\expandafter \endcsname \expandafter}
171   \newcommand*{\ApplySubstringConditionalToInputString}[1]{% 2009/03/31+
172     \csname setup_substr_cond:#1\expandafter \endcsname
173       \expandafter {\fdInputLine}}
174     %% TODO or `\OutputString', even `\read' to `\OutputString'!?
175   % \newcommand*{\ApplySubstringConditionalToExpanded}[2]{%
176   %   %% note: without assignments, robust!
177   %   %% BUT the `\csname ... \expandafter \endcsname' method is faster
178   %   \expandafter \reversed_apply_substr_cond
179   %     \expandafter {#2}{#1}}
180   % \newcommand*{\reversed_apply_substr_cond}[2]{%
181   %   \ApplySubstringConditional{#2}{#1}}
182     %% ODER:
183   % \newcommand*{\expand_attach_arg}[2]{%% 2009/03/31
```

```
%   %% #1 command with previous args, TODO cf. LaTeX3
%   \expandafter \attach_arg \expandafter {#1}{#2}}
%   %% actually #1 may contain more than one token,
%   %% only first expanded
% \newcommand*{\attach_arg}[2]{#2{#1}}
% \newcommand*{\ApplySubstringConditionalToExpanded}[2]{%
%   \expandafter \attach_arg \expandafter
%     {#2}{\ApplySubstringConditional{#1}}}
```

9 VERSION HISTORY

```
v0.1   2009/04/03   very first version, tested on morgan.sty
v0.2   2009/04/05   counter fdInputLine, \ProvidesFile moved from
                    \ProcessFile to \ResultFile, \CopyFD...,
                    category section first, more sectioning,
                    suppressing empty code lines before section
                    titles; discussion, \Delimiters
       2009/04/06   more discussion
       2009/04/07   more discussion, factored \WriteProvides out from
                    \ResultFile, \ProcessExpandedWith corrected
       2009/04/08   \InputString -> \fdInputline;
                    removed \ignorespaces
       2009/04/09   \WhenInputLine[2] -> \IfInputline[3],
                    \ProcessInputWith, typos,
                    \WriteProvides message ‘with’
       2009/04/10   \make_not_expanding_cs
                    DISCOVERED ‘‘IF SUBSTRING’’ ALGORITHM WRONG
                    (<str1><str2><str1> in <str1><str2>)
v0.3   2009/04/11   SOME THINGS GIVEN UP EARLIER WILL BE REMOVED,
                    TO BE STORED IN THE COPY AS OF 2009/04/10
                    mainly: sandbox setup (tilde/dollar)
                    REAL ADDITION: setup for expandable replacing
       2009/04/12   played with ‘chain’ vs. ‘sequence’;
                    plain ‘...’, ‘cf.’, ‘etc.’ for ‘mdcorr.cfg’
       2009/04/13   \RemoveTilde...
       2009/04/15   reworked text, same mistake \in@
```