bitelist.sty

—

"Splitting" a List at a List Inside in TEX's Mouth*

Uwe Lück†

March 29, 2012

Abstract

bitelist.sty provides commands for "splitting" a token list at the first occurrence of a contained token list (i.e., for given σ, τ, return β and shortest α s.t. $\tau = \alpha\sigma\beta$). As opposed to other packages providing similar features, (*i*) the method uses TEX's mechanism of reading delimited macro parameters; (*ii*) the splitting macros work by pure expansion, without assignments, provided the macro doing the search has been defined before processing (e.g., a file); (*iii*) instead of using one macro for a "substring" test and another one to replace the "substring"—which includes extracting corresponding prefix and suffix—, the *same* macro that detects the occurrence returns the split; (*iv*) ε-TEX is not required. (And LATEX is not required.)

This improves the author's fifinddo.sty (v0.51—and may once be used there). An elaborated approach (additionally to a simpler one) is provided that does not loose outer braces of prefix/suffix.

"Substring" detection and "string" replacement are (implicitly) included with respect to certain representations of characters by tokens. Counting occurrences and "global" replacement could be achieved by applying the operation to earlier results, etc.—so this approach seems to be "fundamental" for a certain larger set of list analysis tasks.

The documentation aims to prove the correctness of the methods with mathematical rigour.

Related packages: datatool, stringstrings, ted, texapi, xstring

Keywords: macro programming, text filtering, substrings

*This document describes version v0.1 of bitelist.sty as of 2012/03/29.

†http://contact-ednotes.sty.de.vu

Contents

1 Task, Background Reasoning, and Usage

1.1 The Task Quite Precisely

Perhaps I should not have written "splitting" before, see Section 7 why I did so though. Actually:

At first we are dealing with token lists τ and σ without braces (unless their category code has been changed appropriately) that can be stored as macros without parameter or in token list registers. We want to find out whether τ contains σ ("as a subword") in the sense that there are such token lists α and β that τ is composed as $\alpha\sigma\beta$, i.e.,

$$\tau = \alpha\sigma\beta$$

and in this case we want to get α and β of this kind with α being the *shortest* possible. I.e., if there are such γ and δ that τ is composed as $\gamma\sigma\delta$, α must be

contained as a "prefix" in γ, i.e., γ is composed as $\alpha\eta$ for some token list η. The token lists α, β, γ, δ, η, σ, and τ are allowed to be empty throughout.

The task will be extended for some braces in Section 4.

1.2 Idea of Solution

TEX's mechanism of expanding macros (TEXbook Chapter 20) at least has a built-in mechanism to return such α and β *provided* τ contains σ. Define

`\def`⟨*cmd*⟩`#1`σ`#2`θ`{`⟨*replace-def*⟩`}`

where θ must be a token list (maybe of a single token) that won't occur in τ.[1] This is a **limitation** of the approach: It works for sets of such τ only that do not contain any of a small set of tokens or combinations of them. (bitelist will use `\BiteSep`, `\BiteStop`, and `\BiteCrit`, or any other three that can be chosen.)

On the other hand, TEX's *category codes* (TEXbook Chapter 7) can ensure this quite well. E.g., we may assume that input "letters" always have category code 11 (or 12, or one of them), and for θ we can choose letters with *different* category codes such as 3. Without such tricks, you may often assume that nobody will input certain "silly" commands such as `\BiteStop`. (But it may become difficult when you use a package for replacement macros for generating its own documentation ...)

With a ⟨*cmd*⟩ as defined above, TEX will

expand ⟨*cmd*⟩$\tau\theta$ to ⟨*replace*⟩,

where ⟨*replace*⟩ will be the result of replacing (a) all occurrences of `#1` in ⟨*replace-def*⟩ by α as wanted and (b) all occurrences of `#2` in ⟨*replace-def*⟩ by β as wanted. I.e., ⟨*cmd*⟩ returns α as its first argument and β as its second argument. The reason is that ⟨*cmd*⟩'s first parameter is delimited by σ and the second one by θ in the sense of The TEXbook p. 203. Our requirement to get the *shortest* α for the composition of τ as $\alpha\sigma\beta$ is met because TEX indeed looks for the *first* occurrence of σ at the right of ⟨*cmd*⟩.

1.3 When We Don't Know ...

When σ does *not* occur in τ and we present $\tau\theta$ to ⟨*cmd*⟩ as before, TEX will throw an error saying "Use of ⟨*cmd*⟩ doesn't match its definition." When the purpose is "substring detection" only, without returning β, many packages have solved the problem by issuing something like

⟨*cmd*⟩$\tau\sigma\theta$

[1] I am still following others in confusing source code and tokens. I have better ideas, but must expand on them elsewhere. Writing `\def` rather indicates that it is source code, then σ etc. should be replaced by strings that are converted into tokens σ etc. ⟨*cmd*⟩ sometimes is a *string* starting with an escape character, or it is an active character; but sometimes it rather is an "active" *token* converted from such an escape string or an active character.

Then (still provided θ does not occurr in τ) $\langle cmd\rangle$'s second argument is empty *exactly* if σ occurs in τ. This method has, e.g., been employed in LaTeX's internal `\in@` mechanism (e.g., for dealing with package options) and by the substr package. datatool has used the latter's substring test (for σ) before calling a macro for replacing (σ by another token list, perhaps thinking of character tokens).

This way you get the wanted α as the first macro argument immediately indeed. An obstacle for getting β is that $\langle cmd\rangle$'s *second* argument now contains an occurrence of σ that is not an occurrence in τ. In fifinddo.sty I didn't have a better idea than using another macro to remove the "dummy text" from the second argument. I considered it an advantage as compared with datatool that *one* macro could do this for *all* replacement jobs, while datatool uses *two* macros with σ as a delimiter for each σ to be replaced.

But still, fifinddo has used *two* macros for each replacement, the extra one being for presenting τ to $\langle cmd\rangle$, using a job identifier. This could be improved within fifinddo, but I could never afford to take the time for this.

1.4 The Trick

The solution presented here is not very ingenious, many students would have found it in an exercise for a math course. My personal approach was looking at `\GetFileInfo` from LaTeX's doc package. There they try to get *two* occurrences of a space token this way:[2]

```
\def\@tempb#1␣#2␣#3\relax#4\relax{%
```

and `\@tempb` is called as

```
\@tempbτ\relax?␣?␣\relax\relax
```

or with $\tau = \langle list\rangle$

```
\@tempb⟨list⟩\relax?␣?␣\relax\relax
```

The final `\relax` may not be removed, but for doc it doesn't harm. It harms for *me* when I don't want to have a `\relax` in a `.log` file list. `\empty` would be better, however ...

The idea is to use a *three*-parameter macro for that *single* occurrence of σ. We introduce a "dummy separator" ζ (or $\langle sep\rangle$, `\BiteSep`) between τ and the "dummy text" and a "criterion" ρ (= $\langle crit\rangle$, `\BiteCrit`) for determining occurrence of σ (= $\langle find\rangle$) in τ (= $\langle list\rangle$). Neither ζ nor ρ must occur in τ. We will have definitions about as

```
\def⟨cmd⟩#1σ#2ζ#3θ{⟨replace-def⟩}
```

or

```
\def⟨cmd⟩#1⟨find⟩#2⟨sep⟩#3⟨stop⟩{⟨replace-def⟩}
```

[2] We are undoubling the hash marks inside the definition text of `\GetFileInfo`.

and τ will be presented with context

$$\langle cmd\rangle\tau\zeta\sigma\rho\zeta\theta \quad \text{or} \quad \langle cmd\rangle\langle list\rangle\langle sep\rangle\langle find\rangle\langle crit\rangle\langle sep\rangle\langle stop\rangle$$

This ensures that $\langle cmd\rangle$ finds its parameter delimiters σ, ζ, and θ, in this order. σ occurs in τ exactly if the second argument of $\langle cmd\rangle$ is ρ, and in this case the first occurrence of the second parameter delimiter ζ delimits τ. Then $\langle cmd\rangle$'s first argument is α, and the second one is β, as wanted.

$\langle cmd\rangle$'s *third* parameter is delimited by the final θ (`\BiteStop`). When σ occurs in τ, $\langle cmd\rangle$'s third argument starts after the first of the two ζ, so it is $\sigma\rho\zeta$. It is just ignored, this way $\langle cmd\rangle$ removes all the "dummy" material after τ. When σ does *not* occur in τ, we ignore all of its arguments, and the macro that invoked $\langle cmd\rangle$ must decide what to do next, e.g., keeping τ elsewhere for presenting it to another parsing macro resembling $\langle cmd\rangle$.

1.5 Installing and Calling

The file bitelist.sty is provided ready, installation only requires putting it somewhere where TEX finds it (which may need updating the filename data base).[3]

Below the `\documentclass` line(s) and above `\begin{document}`, you load bitelist.sty (as usually) by

```
\usepackage{bitelist}
```

between the `\documentclass` line and `\begin{document}`; or by

```
\RequirePackage{bitelist}
```

within a package file, or above or without the `\documentclass` line. Moreover, the package should work *without* LATEX and may be loaded by

```
\input bitelist.sty
```

Actually, using the package for macro programming requires understanding of pp. 20f. of The TEXbook. On the other hand, the package may be loaded (without the user noticing it) automatically by a different package that uses programming tools from the present package.

2 Implementation Part I

2.1 Package File Header (Legalize)

```
1  \def\filename{bitelist}                    \def\filedate{2012/03/29}
2  \def\fileversion{v0.1} \def\fileinfo{split lists in TeX's mouth (UL)}
3  %% Copyright (C) 2012 Uwe Lueck,
4  %% http://www.contact-ednotes.sty.de.vu
5  %% -- author-maintained in the sense of LPPL below --
```

[3] http://www.tex.ac.uk/cgi-bin/texfaq2html?label=inst-wlcf

```
%%
%% This file can be redistributed and/or modified under
%% the terms of the LaTeX Project Public License; either
%% version 1.3c of the License, or any later version.
%% The latest version of this license is in
%%    http://www.latex-project.org/lppl.txt
%% There is NO WARRANTY - this rather is somewhat experimental.
%%
%% Please report bugs, problems, and suggestions via
%%
%%   http://www.contact-ednotes.sty.de.vu
%%
```

2.2 Proceeding without LaTeX

Some tricks from Bernd Raichle's ngerman.sty—I need LaTeX's `\ProvidesPackage` for fileinfo, my package version tools. With readprov.sty, it issues `\endinput`, close conditional before:

```
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname ProvidesPackage\endcsname\relax \else
    \edef\fileinfo{\noexpand\ProvidesPackage{\filename}%
        [\filedate\space \fileversion\space \fileinfo]}
    \expandafter\fileinfo
\fi
\chardef\atcode=\catcode`\@
\catcode`\@=11 % \makeatletter
```

Providing LaTeX's `\@firstoftwo` and `\@secondoftwo`:

```
\long\def\@firstoftwo #1#2{#1}
\long\def\@secondoftwo#1#2{#2}
```

2.3 Basic Parsing (No Braces)

`\BiteMake{`⟨*def*⟩`}{`⟨*cmd*⟩`}{`⟨*find*⟩`}` provides the parameter text (TeXbook p. 203) for defining (by ⟨*def*⟩) a macro ⟨*cmd*⟩ that will search for ⟨*find*⟩:

```
\def\BiteMake#1#2#3{#1#2##1#3##2\BiteSep##3\BiteStop}
```

With `\BiteFindByIn{`⟨*find*⟩`}{`⟨*cmd*⟩`}{`⟨*list*⟩`}`, you can use a ⟨*cmd*⟩ (perhaps defined by `\BiteMake`) in order to search ⟨*find*⟩ in ⟨*list*⟩. This is expandable as promised:

```
\def\BiteFindByIn#1#2#3{%
    #2#3\BiteSep#1\BiteCrit\BiteSep\BiteStop}
```

Preparing a possible `\edef` as ⟨*def*⟩:

```
\let\BiteSep\relax  \let\BiteStop\relax
```

And this is important in any case for correct testing of occurrence:[4]

```
32  \catcode`\Q=7 \let\BiteCrit=Q \catcode`\Q=11
```

Perhaps you could increase safety of tests by using something similar to the funny `Q` for `\BiteSep` and `\BiteStop`. However, this would additionally require reimplementation of the macros for keeping braces (Section 4) using `\edef`.

2.4 Simple Conditionals

By `\BiteMakeIfOnly{`⟨*def*⟩`}{`⟨*cmd*⟩`}{`⟨*find*⟩`}`, you can make a command ⟨*cmd*⟩ that with

`\BiteFindByIn{`⟨*find*⟩`}{`⟨*cmd*⟩`}{`⟨*list*⟩`}{`⟨*yes*⟩`}{`⟨*no*⟩`}`

chooses ⟨*yes*⟩ if ⟨*find*⟩ occurs in ⟨*list*⟩ and ⟨*no*⟩ otherwise.

```
33  \def\BiteMakeIfOnly#1#2#3{\BiteMake{#1}{#2}{#3}{\BiteIfCrit{##2}}}
```

`\BiteIfCrit{`⟨*suffix*⟩`}{`⟨*yes*⟩`}{`⟨*no*⟩`}` is the basic test for occurrence of ⟨*find*⟩ in ⟨*list*⟩:

```
34  \def\BiteIfCrit#1{\ifx\BiteCrit#1\expandafter\@secondoftwo
```

If ⟨*cmd*⟩'s second argument—same as `\BiteIfCrit`'s first argument—is empty, `\BiteCrit` is compared with `\expandafter`, so ⟨*yes*⟩ is chosen. That is correct, it happens when ⟨*find*⟩ is a suffix of ⟨*list*⟩.

```
35                                  \else \expandafter\@firstoftwo \fi }
```

2.5 Passing Results Completely—No Braces

So the previous `\BiteMakeIfOnly` generates pure tests on occurrence, giving away information about prefix and suffix. It may be considered a didactical step fostering understanding of the following. When, by contrast

`\BiteMakeIf{`⟨*def*⟩`}{`⟨*cmd*⟩`}{`⟨*find*⟩`}`

has been issued, a later

`\BiteFindByIn{`⟨*find*⟩`}{`⟨*cmd*⟩`}{`⟨*list*⟩`}{`⟨*list*⟩`}{`⟨*yes*⟩`}{`⟨*no*⟩`}` (∗)

will expand to

⟨*yes*⟩`{`⟨*prefix*⟩`}{`⟨*find*⟩`}{`⟨*suffix*⟩`}`

if ⟨*list*⟩ is composed as ⟨*prefix*⟩⟨*find*⟩⟨*suffix*⟩ and ⟨*prefix*⟩ is the shortest α such that there is some β with ⟨*list*⟩ $= \alpha$⟨*find*⟩β. Otherwise, (∗) will expand to

⟨*no*⟩`{`⟨*list*⟩`}`

This gives all the information available. For actual applications, it may be too much, and the macro programmer may do something in between of `\BiteMakeIfOnly` and `\BiteMakeIf`:

[4] The idea for the "funny `Q`" is from the ifmtarg package.

```
\def\BiteMakeIf#1#2#3{%
    \BiteMake{#1}{#2}{#3}##4##5##6{%
```

In the replacement text, we first do the same as with `\BiteMakeIfOnly`:

```
      \BiteIfCrit{##2}%
```

What follows is new. ⟨*cmd*⟩'s third argument is ignored. The fourth keeps the original ⟨*list*⟩. ⟨*yes*⟩ is ⟨*cmd*⟩'s fifth and ⟨*no*⟩ is its sixth argument.

```
      {##5{##1}{#3}{##2}}%  %% if #3 in ##4
      {##6{##4}}%           %% otherwise
    }%
}
```

In (∗), ⟨*list*⟩ has been doubled. That was no mistake. It is due to a shortcoming of `\BiteFindByIn`. With

`\BiteFindByInIn{`⟨*find*⟩`}{`⟨*cmd*⟩`}{`⟨*list*⟩`}{`⟨*yes*⟩`}{`⟨*no*⟩`}`

you get the same result as with (∗):

```
\def\BiteFindByInIn#1#2#3{\BiteFindByIn{#1}{#2}{#3}{#3}}
```

TODO not sure about command names yet

3 Example Applications

3.1 Splitting at Space

This work actually arose from modifying `\GetFileInfo` as provided by LaTeX's doc package so that it would deal reasonably with "incomplete" file info—for the nicefilelist package. `\GetFileInfo` works best when the file info contains at least *two* blank spaces. But how many are there indeed?—And I wanted to do it *expandably:* while `\GetFileInfo` issues *definitions* of `\filedate`, `\fileversion`, and `\fileinfo`, date, version, and info should be passed as *macro arguments*.

`\BiteIfSpace` tries splitting at the next blank space passes results:

```
\BiteMake{\def}{\BiteIfSpace}{ }#4#5#6{%
    \BiteIfCrit{#2}{#5{#1}{#2}}{#6{#4}}}
```

The difference to the `\BiteMakeIf` construction is that we do not pass ⟨*find*⟩, the space—it's not essential. (TODO names may change . . .)

Now

`\BiteFindByInIn{␣}{\BiteIfSpace}{`⟨*list*⟩`}{`⟨*yes*⟩`}{`⟨*no*⟩`}`

will pass prefix/suffix to ⟨*yes*⟩ or ⟨*list*⟩ to ⟨*no*⟩. If this is needed frequently, here is a shorthand `\BiteGetNextWord{`⟨*list*⟩`}{`⟨*yes*⟩`}{`⟨*no*⟩`}`:

```
\def\BiteGetNextWord{\BiteFindByInIn{ }\BiteIfSpace}
```

See a test in `bitedemo.tex` (Section 6).

3.2 Splitting at Comma

... left as an exercise to the reader ...

4 Keeping Braces: Reasoning

Now we want to generalize task (Section 1.1) and solution (Section 1.4) for the case that $\tau = \langle list \rangle$ has (balanced) braces (with category codes for argument delimiters), while $\sigma = \langle find \rangle$ still has not (does not work with our method). So with $\tau = \alpha\sigma\beta$, α ("prefix") or β ("suffix") or both may contain braces. But we consider another restriction: braces must be balanced in α and in β, we don't try parsing inside braces (as opposed to the search for asterisks in Appendix D of The TEXbook).

According to TEXbook p. 204, when a macro $\langle cmd \rangle$ finds an argument formed as `{`$\langle tokens \rangle$`}`, in $\langle cmd \rangle$'s replacement text only $\langle tokens \rangle$ is used, i.e., outer braces are removed. So when $\alpha =$ `{`$\langle tokens \rangle$`}`, a parser $\langle cmd \rangle$ as defined by our methods above will return $\langle tokens \rangle$ instead of `{`$\langle tokens \rangle$`}`—likewise for β. We are now trying to *keep* outer braces in prefix/suffix by a more elaborate method.

The idea is to present $\tau = \langle list \rangle$ with context[5]

$$\langle cmd \rangle \texttt{\textbackslash empty} \langle list \rangle \langle stop \rangle \langle sep \rangle \langle find \rangle \langle crit \rangle \langle sep \rangle \langle stop \rangle$$

or in the notation of Section 1.4

$$\langle cmd \rangle \texttt{\textbackslash empty} \tau\theta\zeta\sigma\rho\zeta\theta$$

Then, if $\langle find \rangle$ occurs in $\langle list \rangle$, we must remove the `\empty` from the prefix that we get with the earlier method (easy) and $\langle stop \rangle$ from the suffix (tricky, similar problem recurs). Using old θ for a new purpose works here because $\langle cmd \rangle$ will look for θ only when it has found ζ before.

Mere testing for occurrence is not affected.

`\BiteMakeIfOnly` and `\BiteFindByIn`

still can be used. We provide an improved version of

`\BiteMakeIf` (`\BiteMakeIfBraces`)

and of

`\BiteFindInIn` (`\BiteFindInBraces`).

[5] Perhaps I am confusing `\empty` and the token list containing just `\empty` here?

5 Implementation Part II

5.1 Keeping Braces

`\BiteFindByInBraces{`⟨*find*⟩`}{`⟨*cmd*⟩`}{`⟨*list*⟩`}{`⟨*yes*⟩`}{`⟨*no*⟩`}`

varies `\BiteFindByInIn` according to the previous:

```
\def\BiteFindByInBraces#1#2#3{%
    #2\empty#3\BiteStop\BiteSep#1\BiteCrit\BiteSep\BiteStop{#3}}
```

Such a ⟨*cmd*⟩ can be made by `\BiteMakeIfBraces{`⟨*def*⟩`}{`⟨*cmd*⟩`}{`⟨*find*⟩`}`:

```
\def\BiteMakeIfBraces#1#2#3{%
    \BiteMake{#1}{#2}{#3}##4##5##6{%
      \BiteIfCrit{##2}%
```

⟨*no*⟩ works as before. For ⟨*yes*⟩, first the `\empty` in the prefix is expanded for vanishing. `\BiteTidyI` and `\BiteTidyII` continue tidying.

```
        {\expandafter \BiteTidyI                    %% if #3 in ##4
              \expandafter{##1}%                    %% prefix
```

Another `\empty` avoids that removal of `\BiteStop` in suffix by `\BiteTideII` removes outer braces:

```
                          {\BiteTidyII\empty##2}%   %% suffix
                          {#3}%                     %% find
                          {##5}}%                   %% yes
        {##6{##4}}%                                 %% otherwise
      }%
}
```

`\BiteTidyI{`⟨*prefix*⟩`}{`⟨*suffix*⟩`}` first expands `\BiteTidyII` for removing `\BiteStop` in ⟨*suffix*⟩. `\empty` from `\BiteFindByInBraces` remains and is expanded next for vanishing. Finally, `\BiteTidied` reorders arguments for operation of ⟨*yes*⟩:

```
\def\BiteTidyI#1#2{%
    \expandafter\expandafter\expandafter \BiteTidied
        \expandafter\expandafter\expandafter {#2}{#1}}
\def\BiteTidyII#1\BiteStop{#1}
\def\BiteTidied#1#2#3#4{#4{#2}{#3}{#1}}
```

5.2 Leaving the Package File

```
\catcode`\@=\atcode
\endinput
```

5.3 VERSION HISTORY

```
v0.1   2012/03/26   started
       2012/03/27   continued, restructured
       2012/03/28   continued, separate sections for "Mere Occurrence"
                    vs. ...; keeping braces, \BiteIfCrit
       2012/03/29   proceeding without LaTeX corrected, restructured

```

6 Examples/Tests

You should find a separate file `bitedemo.tex` with examples. It may be run separately with `tex` (Plain TEX)—demonstrating that bitelist is "**generic**", then finish by entering `\bye`. With "`latex␣bitedemo.tex`", end the job by entering `\stop`. **Expandability** is demonstrated by the `\BiteFind` commands running with `\typeout`.

```
\def\filename{bitedemo.tex}  \def\filedate{2012/03/29}
\def\fileinfo{demonstrating/testing bitelist.sty (UL)}
\expandafter\ifx\csname ProvidesPackage\endcsname\relax \else
    \edef\bitedemolatexstart{%
        \noexpand\ProvidesFile{\filename}%
                               [\filedate\space\fileinfo]%
        \noexpand\RequirePackage{bitelist}}
    \expandafter\bitedemolatexstart
\fi
\ifx\BiteMakeIf\undefined \input bitelist.sty \fi
\ifx\typeout\undefined
    \def\typeout{\immediate\write17}
    \newlinechar`^^J
\fi
\def\splitted  #1#2#3{>>#1|#2|#3<<}
\def\unsplitted#1{>>#1<<}
\def\spacetocomma#1#2{>>#1,#2<<}
\BiteMakeIfOnly  {\def}{\occursyesno}{no}
\BiteMakeIf      {\def}{\noshowsplit}{no}
\BiteMakeIfBraces{\def}{\noShowSplit}{no}
\typeout{^^J
    \BiteFindByIn  {no}{\occursyesno}
                   {bonobo}{YES!}{NO!}
    \BiteFindByIn  {no}{\noshowsplit}
                   {bonobo}{bonobo}{\splitted}{\unsplitted}
    \BiteFindByInIn{no}{\noshowsplit}
                   {bonobo}{\splitted}{\unsplitted}
         ^^J
    \BiteFindByIn  {no}{\occursyesno}
```

```
                {bobobo}{YES!}{NO!}
    \BiteFindByIn  {no}{\noshowsplit}
                {bobobo}{bobobo}{\splitted}{\unsplitted}
    \BiteFindByInIn{no}{\noshowsplit}
                {bobobo}{\splitted}{\unsplitted}
         ^^J
    \BiteFindByInBraces{no}{\noShowSplit}
                {{bo}no{bo}}{\splitted}{\unsplitted}
         ^^J
    \BiteGetNextWord{bo no bo}{\spacetocomma}{\unsplitted}
    \BiteGetNextWord{bo nobo} {\spacetocomma}{\unsplitted}
    \BiteGetNextWord{bonobo}  {\spacetocomma}{\unsplitted}
         ^^J}
\endinput
```

---

7 The Package's Name

This package deals with TEX's expansion mechanism. In Knuth's metaphor, this is TEX's mouth. I am not entirely sure, I have never understood it, or I have understood it only for a few days or hours. However, the package deals with "Lists in TEX's Mouth" as described in Alan Jeffrey's 1990 TUGboat paper (Volume 11, No. 2, pp. 237–245).[6]

"Splitting" in title and abstract is an attempt to describe the package briefly without speaking Mathematicalese. It roughly refers to certain string functions in various programming languages[7] with "`split`" in their name. However, there strings are splitted at separators such as commas. I am thinking here that a comma is a certain string "`,`", and this can be generalized to "splitting" at any substring. With TEX, the analogues are (a) the token with the character code of the comma and category code 12, or the token list consisting of this single token,—and (b) other lists of tokens ...

Anyway, calling a triple (α, σ, β) of token lists such that $\tau = \alpha\sigma\beta$ a "split" of τ is not necessarily a bad idea. Moreover, the blank space example (Section 3.1) is very close to the original idea of splitting at separators, a blank space is about as common as a separator as the comma is.

Finally, according to `en.wiktionary.org`, the Proto-Indo-European origin of "to bite" just means "to split."[8] So in TEX's mouth, splitting and biting is the same.

---

[6] `tug.org/TUGboat/tb11-2/tb28jeffrey.pdf`
[7] `en.wikipedia.org/wiki/String_functions#split`
[8] `en.wiktionary.org/wiki/bite#Etymology`