Chapter 1

User Manual

This section introduces how to use features provided by TXLisp. However, to benefit properly from TXLisp one has to know both TeX and ELisp, because TXLisp is merely a way to work combinedly with these two languages. One has to bear knowledge of creating TeX documents and programming in ELisp in order to be in a position to exploit new possibilities opened by TXLisp. This section will only instruct, how to install TXLisp to Emacs, how to edit TXLisp files and what additional features are provided by TXLisp Overlay minor mode.

1.1 Getting Started

In order to use TXLisp in Emacs, Emacs must load the TXLisp source files. There are several ways of loading ELisp files in Emacs. One could open 'tisp.el' like any file in a buffer by pressing 'Control-x Control-f', type in path and file name and after opening the file in a buffer a user could type 'Alt-x eval-buffer RETURN' to make TXLisp ready for use. Another way would be using 'load-file' or 'load-File' commands by pressing 'Alt-x' and entering the name of a command followed by 'RETURN'. Once again a user would need to type in a file name but this time user would not need to evaluate the buffer, because Emacs would not even have opened a file in a buffer but would only have evaluated the content of the file. This would be ok, if one would be using TXLisp only once in a while. Still, most probably the most convenient way is to add few lines into Emacs configuration file\(^1\) which will make Emacs to load TXLisp automatically, when a user starts Emacs. This way TXLisp would be available when needed.

Following lines will do this:

```lisp
(setq load-path (cons " /EmacsFiles/" load-path))
(load "txlisp")
(load "txlisp-overlay")
```

*Example 1.1: Emacs file configuration for adding an element to Emacs load path*

The first line will add 'MyEmacsFiles' to load path of Emacs. Load path is where Emacs will look for any files it requires. Following two lines will make

\(^1\)A standard Emacs configuration file is .emacs
Emacs to try to load files txisp and txisp-overlay. What happens is, that Emacs will search through the directories defined in its load-path trying to locate files txisp.el or txisp.el and txisp-overlay.el or txisp-overlay.el and, when Emacs finds a file it evaluates the contents of the file. Evaluation effectively initialises variable and function definitions and makes them available in an Emacs session.

The difference between .el and .elc is that .elc is a compiled version of .el. That is why Emacs always tries to find .elc files first and only, if a file's .elc file cannot be found, Emacs begins to look for a .el file. Now, it it only the matter of copying txisp.el and txisp-overlay.el to MyEmacsFiles directory and they will be available whenever a user starts Emacs.

There are plenty of ways how a user can enter the TXLisp mode: open Emacs with argument which is a TXLisp file, press Control-x Control-f to create a new buffer with a name ending to the TXLisp file type extension or press Alt-x and type txisp-mode RETURN. After starting Emacs and turning on TXLisp mode TXLisp provides a number of commands to help editing TXLisp files. All of these commands locate under 'Control-t', i.e. a user must press 'Control-t' and one of the following key combinations to execute a command. 'Control-c' and 'Control-v' insert a conversion begin and end tags respectively at a current point in a buffer. In similar manner 'Control-d' and 'Control-f' insert evaluation begin and end tags, 'Control-z' and 'Control-x' insert evaluate and print begin and end tags and 'Control-a' and 'Control-s' insert evaluate and print last begin and end tags. The values for these tags are stored in txisp-conv-begin-tag, txisp-conv-end-tag, txisp-conv-begin-tag, txisp-conv-end-tag, txisp-conv-print-begin-tag, txisp-conv-print-end-tag, txisp-conv-print-last-begin-tag and txisp-conv-print-last-end-tag.

A user can change values of the tags by pressing 'Alt-x' and typing 'txisp-update-tag RETURN'. The program will ask a user to type in the name of a tag whose value should be changed and then a new value. However, the command will not allow a user to proceed until it receives a valid tag name so, if the user would like to cancel the action, the user has to either press Control-o or delete all the characters from the minibuffer and press RETURN. After receiving a valid tag name the command will store the new value into the chosen variable and will also update the new value for syntax highlighting. However, this will only change the value for this session. To change the value permanently a user must add following lines to Emacs configuration file after the lines where TXLisp is loaded:

```
(setq <name of tag> "new value")
(txisp-update-keywords (<keyword-entry>))
```

*Example 1.2: Emacs file configuration for changing TXLisp tags*

<name of tag> being one of the variable names mentioned above and new value a string of characters to define a new value for a tag. In the above example <keyword-entry> is one of the following four options: txisp-conv-keywords-entry, txisp-conv-print-keywords-entry or txisp-conv-print-last-keywords-entry. The name of tag and <keyword-entry>

---

2 A user can change values of the tags by pressing 'Alt-x' and typing 'txisp-update-tag RETURN'. The program will ask a user to type in the name of a tag whose value should be changed and then a new value. However, the command will not allow a user to proceed until it receives a valid tag name so, if the user would like to cancel the action, the user has to either press Control-o or delete all the characters from the minibuffer and press RETURN. After receiving a valid tag name the command will store the new value into the chosen variable and will also update the new value for syntax highlighting. However, this will only change the value for this session. To change the value permanently a user must add following lines to Emacs configuration file after the lines where TXLisp is loaded:

```
(setq <name of tag> "new value")
(txisp-update-keywords (<keyword-entry>))
```


---

3 Default values are /<convb>, /<convc>, /<ceb>, /<ee>, /<epb>, /<epe>, /<epc>, /<epd> respectively

4 A standard keyboard quit key in Emacs

---

5 Default value is /<convb>
must be of matching category the change to take place correctly. If for example
the value of txlisp-conv-begin-tag is changed but txlisp-eval-keywords-entry is
given as an argument to txlisp-update-keywords, the new value of txlisp-conv-
begin-tag is not updated to syntax highlighting functions.

1.2 Using Tags

The following examples will show, how TXLisp tags can be used to embed ELisp
segments to \TeX\ files. The first three examples show the difference between eval-
uation tags. A conversion tag example follows these examples and the last three
examples show, how to duplicate a \TeX\ segment, how to add user interaction
and finally how to use remote file accessing. In the following examples actual
TXLisp code is shown on the left hand side column and a result of a possible
output after running the output through a \TeX\ compiler on the right hand side
column in bold.

\begin{verbatim}
/<eb>
(setq x 5)
(setq y 7)
/<ee>
\end

Example 1.3: Evaluation tag

In this first example it is important to notice that /<eb> begins an eval-
uation region and /<ee> ends the region. Another important fact is that there
is no output at all from this region. The program simply discards the output
from evaluations of the two s-expressions. What happens is that variables \textit{x}
and \textit{y} are initialised to values 5 and 7 correspondingly.

\begin{verbatim}
/<epb>
(setq x 5)
(setq y 7)
/<epe>
\end

Example 1.4: Evaluate and print tag

In the second example /<epb> begins and /<epe> ends an evaluate and
print region. The significant thing about this example is that the variables \textit{x}
and \textit{y} are not only initialised to their values but the program also prints those
values, i.e. the value from evaluation, to output.

\begin{verbatim}
/<eplb>
(setq x 5)
(setq y 7)
/<eple>
\end

Example 1.5: Evaluate and print last tag


In the third example /<epb> begins and /<eple> ends an evaluate and print last segment. Similarly to first two examples variables \textit{x} and \textit{y} are initialised to values 5 and 7 correspondingly but this time the program prints only the last value, i.e. 7, from the evaluation to the output.

\begin{verbatim}
/<epb>
(setq x
/<convb>
 Hello World!! \hfill break 
/<conve>
); closes 'setq x'
/<epe>
\end

Hello World!!
\end{verbatim}

Example 1.6: 	extit{Conversion tag together with an evaluate and print segment}

In the fourth example the important fact is that /<convb> begins and /<conve> ends a conversion segment. Conversion segments allow to pass \TeX\ segments as arguments to \textsc{Elisp} functions. In this particular case variable \textit{x}'s value is initialised to string \textit{Hello World!!} \hfill \textit{break}. Another important thing to notice is the segment processing order (see Figure 1.7). At first the program turns the conversion segment in to a valid \textsc{Elisp} string and only then proceeds to process the evaluation segment, i.e. the processing order is from inside to outside.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{conversion_segment.png}
\caption{Conversion segment processing}
\end{figure}

\begin{verbatim}
\noindent
/<epb>
(setq count 3)
(defun testfun (count)
  (if (> count 0)
    (concat
     /<convb>
    Hello World! \hfill break 
     /<conve>
    (testfun (- count 1)))
   "")
  (testfun count)
/<eple>
\end

Hello World!
Hello World!
Hello World!
\end{verbatim}

Example 1.8: 	extit{Duplicating \TeX\ segments}
1.2. USING TAGS

Fifth example begins with a TeX segment consisting of just a one command \noindent. The TeX segment is then followed by evaluate and print last segment. This ELisp segment defines a variable count and initialises it to value 3, defines a function which concatenates to a one string count times, what ever is inside the conversion segment. In this case the result is Hello World! being printed three times. What is important to notice from this example, is the fact, how the whitespaces count. Where /<eplb> begins is where the output from the evaluation of this region will begin. This means that in a case of /<eplb> beginning immediately after \noindent there would not be left any space between the first Hello World and \noindent. On the other hand the program removes all the whitespace immediately after a conversion begin tag and all the whitespace apart from one immediately before a conversion end tag. So, in this example it means that in the final concatenated string World\hfill\break is written three times and each occurrence of the sentence is separated from each other by a single whitespace, because between the last character in the sentence and conversion end tag there is a new line and several whitespaces which the program turn into a single whitespace.

\noindent
/<eb>
(setq count (read-from-minibuffer "How often:"))
(setq count (string-to-number count))
/<ee>
/<eplb>
(defun testfun (count)
(if (> count 0)
(concat
/<convb>
Write this "sentence"
/<eplb>
(cond
((equal count 1) "one")
((equal count 2) "two")
((equal count 3) "three")
/<eple>
times|hfill|break
/<<conve>
(testfun (- count 1)))
"
))
(testfun count)
/<eple>
\end

Write this "sentence" two times
Write this "sentence" two times

Example 1.9: User interaction

The sixth example begins to show the true power of TXLisp. In the first evaluation segment a user can define, how many times the program will print the sentence defined in a conversion segment. Furthermore, inside the conversion segment there is another evaluate and print last segment which modifies the output according to a user input. This example uses the same function testfun
as the previous example apart from the modified conversion segment. The ELisp segment inside the conversion segments inserts a word *one, two or three* into the conversion segment, depending on the user input. The most important thing to notice from this example is, how ELisp segments can contain conversion segments which again contain ELisp segments. There is no limitation to nesting depth apart from memory limitations but the stack system used by the software is very memory friendly so this should not really be a problem with today’s computers.

```lisp
<eb>
(if (not (load "tex-wcount" t))
  (progn
    (call-process
     "wget"
     nil nil nil
     "--passive-ftp"
     "ftp://archive.cis.ohio-state.edu/pub/emacs-lisp/archive/tex-wcount.el")
    (load (concat default-directory "tex-wcount.el")))
  )
</ee>
@end

**Example 1.10: Remote file accessing**

The seventh and final example introduces another very powerful feature of TXLisp - remote file accessing. This example tests, if *tex-wcount* package is available and, if it is not, the program will download the package and install it. What makes this feature very powerful is the fact that a document can ensure that required software will be available when the document is actually compiled. Additionally, this feature makes also possible to attach remote documents to a current document.

What could be encapsulated to these examples is really only a scratch on a surface of the real potential of TXLisp. One really need to be familiar with Emacs and ELisp to be able to realise, what woderful possibilities TXLisp opens.

The fact that Emacs allows creating subprocesses and allows access to other programs like the last example with *wget* demonstrates, already provides a TXLisp user with a huge variety of interesting possibilities but then on top that a user has all the possibilities of ELisp programming open as well, which leaves a TXLisp user almost with unlimited possibilities.

### 1.3 TXLisp Overlays Minor Mode

TXLisp Overlays minor mode divides an Emacs buffer to \TeX{} and ELisp segments, after which a user can toggle visibility of the segments, i.e. to show both kinds of segments, only \TeX{} segments, only ELisp segments or neither kinds of segments. TXLisp Overlays minor mode also keeps track of changes in a buffer. When ever a user inserts any of the special tags the program checks,

---

*Enables word counting in \LaTeX{}*
whether the tag was inserted into a segment of its own kind, e.g. conversion tag into inside a \TeX{} segment. If this is not the case the program will create a new segment. All the TeX segments have "T" delimiters and Elisp segments "L" delimiters by default. However, these delimiters are only displayed on a screen and therefore not added to the buffer. TXLisp Overlays minor mode also provides a user with commands to jump quickly from one segment to another. This is handy, because even if a segment would be hidden, it still counts for all the buffer commands.

Theoretically TXLisp Overlays minor mode can be used with any major mode, even though its use would not probably make much sense, because its behaviour is bound to TXLisp tags. Anyway, because of this reason the hotkeys to TXLisp Overlay mode commands are set under Control-e\footnote{A standard minor mode prefix key in Emacs}, if a current major mode is not TXLisp and under Control-t otherwise. In TXLisp mode a user can turn on this mode by pressing Control-t Control-b. Otherwise, a user must press Alt-x and type txlisp-overlay-mode RETURN. Once the mode has been turned on, the following key bindings are available with a valid prefix key.

Control-m and Control-l move to the end of a previous segment and to the beginning of a following segment correspondingly. Control-o and Control-p moves in a similar fashion but this time only Elisp segments are considered. And finally, Control-q and Control-w provide the same motion commands to TeX segments.

There are two ways to remove overlays - by turning off TXLisp Overlay mode or by deleting all the characters belonging to an overlay. A user can turn off the mode by pressing Control-b. This removes all the overlays from a buffer and also removes the TXLisp Overlay mode key bindings. If a user deletes all the characters from an overlay, this overlay will be removed. As an indication of overlay removal, the program removes overlay delimiters.