
ConcTeX: Generating a concordance from TeX input files

Laurence Finston

Abstract

ConcTeX is a package that generates a concordance from a plain TeX file. It has been designed specifically for books containing transcriptions of medieval manuscripts, such as facsimile editions, but it can be adapted for other types of material. ConcTeX consists of TeX code in the file `conctex.tex` and a Common Lisp program in the file `conctex.lsp`. The main advantage of ConcTeX is that the same TeX files are used for typesetting and for generating the concordance. It also performs alphabetization of arbitrary special characters and lemmatization. ConcTeX illustrates the power of using TeX and Lisp in combination. It is available under the normal conditions applying to free software.

Introduction

Designing and typesetting a facsimile edition of a medieval manuscript is a formidable challenge. The plates with the facsimile itself will be photolithographs or, in books of the finest quality, collotypes, and will therefore require no typesetting. However, facsimile editions invariably contain parts which must be typeset, such as an introduction, a transcription and a concordance.

Manuscript transcriptions must be carefully designed and they generally require frequent font changes and numerous special characters. Typesetters, if there are any left, are unlikely to be able to read the language of the manuscript, and each manuscript has its own peculiarities of language and orthography, so the process of proofreading and correction is even more difficult than for ordinary books. Nowadays, authors or editors are often expected to supply camera-ready output to the publisher. This often means a printout from one of the popular word-processing packages, which are incapable of producing typesetting of sufficient quality for the task. In today's market, using mechanically set lead type is prohibitively expensive and the number of publishers willing to typeset difficult copy in this way decreases every year.

The task becomes even more daunting when a concordance is desired.¹ Compiling a concordance by hand requires so much labor that it is no longer economically feasible. Today, concordances are

¹ A concordance is a complete listing of all words occurring in a manuscript, lemmatized, with the main forms of the lemmata sorted alphabetically.

generated by means of computer programs, but, until now, this has required preparing a specially formatted file containing the transcript. Since a separate file, or a typescript, was used for typesetting, changing either the transcript or the concordance necessitated making the corresponding change in the other. In some cases, a conversion program could be used to automate this process. Where this was not possible, either because a typescript was used or no conversion program was available, every change had to be made in two places by hand: an editorial nightmare.


ConcTeX is a package that attempts to solve these problems. It includes a file of TeX code, `conctex.tex`, containing tools for designing a facsimile edition of a manuscript, and a program written in Common Lisp, `conctex.lsp`,² for generating a concordance from the TeX input files. The concordance program performs lemmatization and alphabetization of arbitrary special characters, and its output is another TeX input file containing the concordance. ConcTeX is designed for producing facsimile editions of manuscripts, but it can be adapted for other types of material.

Using ConcTeX makes it possible to benefit from the typographic capabilities of TeX and METAFONT, of which readers of *TUGboat* need not be convinced. Apart from this, its most significant advantage is that the TeX input files are used both for typesetting and for producing the concordance, so that any changes in the input files are automatically reflected in both the printed output and the concordance.

ConcTeX is not for novices. A certain amount of TeXpertise and knowledge of Lisp are necessary to use it successfully. The version I describe in this article has been designed for a particular project. Any other project will require some customization. Many of the individual features of ConcTeX as described here are the result of decisions regarding the design of a particular book. Other books will require other decisions. I have programmed most of the routines in a general way, so that details can be changed while the basic operation of ConcTeX remains the same.

Both the TeX code in `conctex.tex` and the Lisp program `conctex.lsp` use some fairly advanced techniques, so readers may find this article somewhat difficult. I expect the description of the Lisp program will present the most problems, because Common Lisp is likely to be unfamiliar to

² Technically, it is incorrect to speak of "the program" `conctex.lsp`. In Lisp a program is not a file. It is, however, convenient to refer to the file `conctex.lsp` as "the program".

most \TeX users.³ Some of the more difficult and subtle points are in the footnotes; for others, I've borrowed Prof. Knuth's "dangerous bend" sign:  I sometimes use footnotes and "dangerous bend" paragraphs to refer to topics that are introduced later in the article. Most readers will want to skip these paragraphs on the first reading.

Generating a concordance is admittedly a special application; most \TeX users won't want to do such a thing. However, the techniques for extracting information from \TeX input files, described in this article, are of general applicability.

In the following description and examples, I use two transcriptions of Icelandic manuscripts of the 13th century, AM 234 fol and Holm perg 11 4^o, each containing a *vita* of the Virgin Mary and a collection of miracles. I wish to thank Dr. Wilhelm Heizmann, my *Doktorvater* (dissertation advisor), who prepared the transcriptions, for permission to use them in this article.⁴

In the following, many phrases have special meanings. They are all explained at their first appearance, but to avoid confusion, they are listed in a glossary on page 402.

Installation. In order to use $\text{Conc}\text{\TeX}$, the file `conctex.lsp`, containing the Lisp program, must be in your working directory, and `conctex.tex`, containing \TeX code, must be either in your working directory or in a directory in \TeX 's load path. If you don't know what this is, or how to change it, ask your local \TeX wizard, or just put the file in your working directory. The line

```
\input conctex
```

must be at the beginning of your input file.

Why not \LaTeX ? $\text{Conc}\text{\TeX}$ is designed for use with $\text{plain}\text{\TeX}$. It is theoretically possible to adapt it for use with \LaTeX , but I recommend against doing so. I use $\text{plain}\text{\TeX}$ in preference to \LaTeX in $\text{Conc}\text{\TeX}$ for the following reasons:

1. Making significant changes to one of \LaTeX 's pre-defined formats is difficult and time-consuming, whereas programming a format based on $\text{plain}\text{\TeX}$ is relatively easy.
2. \LaTeX enforces a rigid structure on formats. This makes sense for formats that are intended

³ The standard introduction to Lisp is Patrick Henry Winston and Berthold Klaus Paul Horn's *LISP*. Once you know how to program in Lisp, Guy L. Steele's *Common Lisp, The Language* is the one indispensable reference.

⁴ I would also like to thank Günter Koch and Jürgen Hattenbach of the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen, Germany for help above and beyond the call of duty.

to be used by many people, many of whom may have minimal knowledge of how \TeX works. However, it is simply not worth the time and effort to write a \LaTeX format for a single book, when a $\text{plain}\text{\TeX}$ format is just as good and can be written in a fraction of the time. And every book (or series) deserves its own design.

3. \LaTeX loads various files by default, and signals an error if it doesn't get them. Some of the things in these files might not even be necessary, but you've still got to wait until they're loaded.
4. The more macros you use, the more likely it is that they will start to interfere with each other. The problem is even worse when using a large package with macros you a) don't need and b) don't understand. \LaTeX 's macros are very difficult to understand because pieces of them are scattered all over the place. For this reason, it can be very difficult and frustrating trying to get \LaTeX to stop doing something you don't want it to do. $\text{Conc}\text{\TeX}$ changes the `\catcode` of several characters and includes a large number of macros. Therefore, the likelihood is great that there would be interference between $\text{Conc}\text{\TeX}$ and \LaTeX .

The \TeX input file

Multiple input files can be used for typesetting a document and generating a concordance, so the document can be divided into several files in the customary way. In the following, I will assume, for simplicity's sake, that there is only one input file. This file can have any name within reason.

Like any other \TeX input file, an input file for $\text{Conc}\text{\TeX}$ will contain text, control sequences for typesetting and perhaps comments (using `%`). But it will also contain Lisp code used by the program `conctex.lsp`. Therefore, \TeX input files used for generating a concordance are subject to greater restrictions than is ordinarily the case with $\text{plain}\text{\TeX}$. Some parts of the input file will only be used by \TeX , some will only be used by `conctex.lsp`, and some will be used by both, or neither. Many of the complications of $\text{Conc}\text{\TeX}$ have to do with making \TeX and/or Lisp ignore items in the input file.

Environments. $\text{Conc}\text{\TeX}$ changes some category codes and redefines some control sequences in order to format the transcription correctly. This can make it difficult to type in "normal" text using $\text{plain}\text{\TeX}$'s familiar conventions. It might be useful to do this if sections of transcription alternate with

sections of commentary. The macro `\plain` defines an environment where `\catcodes` and macros are reset to their normal values. This environment ends with the macro `\endplain`.

In the `\trans` environment, which is the default, the `\catcodes` of characters and macro expansions are set to the values needed for *transcription lines*, i.e., the lines that contain the actual transcription. The macros `\endplain` and `\endtrans` are defined like this:

```
\let\endplain=\trans
\let\endtrans=\plain
```

The changes made by `\trans` and `\plain` (and `\endplain` and `\endtrans`) are global. A `\trans` in the input file need not be matched by an `\endtrans`, but `\plain` and `\endplain` must be matched, otherwise they will wreak havoc in `conctex.lsp`.

Another environment is used for *commentaries*, which are described below. Commentaries also reset some category codes and macros, but the commentary environment is not identical to the `\plain` environment. Commentaries will usually contain material similar to that in the transcription itself, whereas material in the `\plain` environment should be formatted differently. The `\catcodes` of several characters needed in math mode are also reset by the token list `\everymath`.

⚡ A line beginning with `\trans` or `\endtrans` in an input file will be ignored by `conctex.lsp`, but `\plain` and `\endplain` are replaced by `*`, so `conctex.lsp` treats text between `\plain` and `\endplain` as a commentary.

⚡ Having the `\trans` environment be the default isn't hard-wired into `ConcTeX`; however, the definitions of `\-` and `-` assume that `\trans` is the default, so this will need to be fixed if you want `\plain` to be the default environment.

⚡ In both the `\plain` environment and within commentaries, `-` is reset to `\catcode 12` and `\-` is used for discretionary hyphens, because line breaking is not performed explicitly, but rather by `TeX`'s line breaking routine.

Transcription lines are the lines in the input file that contain the text of the transcription itself. They are processed in whole or in part by both `TeX` and `conctex.lsp`. In order to do its job, `ConcTeX` redefines the `\catcode` of several characters. Each of these changes is explained in its proper place, but there is a list for reference on page 402. Transcription lines are formatted according to the settings in the `\trans` environment, and processed

by `conctex.lsp`. Apart from the text, they may contain items like comments, commentaries, and certain macros.

Comments and commentaries. `ConcTeX` makes a distinction between “comments” and “commentaries”. A comment can be a normal `TeX` comment using `%`, but it can also be code that looks like this:

```
\begincomment{This is a comment.}
\endcomment
```

The format defined in `conctex.tex` uses a conditional (defined with `\newif`) called `\ifdraft`. Whenever `\drafttrue`, that is to say, whenever `\ifdraft≡\iftrue`, certain things are done which are useful for editing purposes, but which aren't done for the final draft. One of these things is printing out comments. If `\drafttrue`, this line:

```
helgum m{\oe}nnum
\begincomment{This is a comment}%
\endcomment {\ae}{\dh}r i.~helgari
```

⇒

```
helgum mœnnum * This is a comment *
æðr i. helgari
```

If `\draftfalse`, it yields

```
helgum mœnnum æðr i. helgari
```

A commentary, on the other hand, is text which should be processed by `TeX` and appear in the output, but should be ignored by `conctex.lsp`, i.e., not be used for generating the concordance. This is for editorial remarks within the transcription itself. Commentaries can be coded in several different ways. Usually, a commentary begins and ends in `*`.⁵

```
\catcode'\*=9
...
herbergi heilag{\slong} anda.~%
* This passage is particularly
interesting * {\oe}llvm\
helgum m{\oe}nnum {\ae}{\dh}r
i.~helgari\
{\tirok} haleitari.~er komin
at k{\ydot}n\
```

⇒

```
herbergi heilagf anda. This passage is particu-
larly interesting œllvm
helgum mœnnum æðr i. helgari
a haleitari. er komin at kÿn
```

⁵ The control symbol `\` in the following example is used for breaking the lines. It's explained on page 381. The letter `{\slong}` → `f`, used in the following examples, is an alternative form of “s”. It is simply the “f” from Computer Modern Roman, with the crossbar removed.

In this example, \TeX simply ignores the character `*`, because its `\catcode` has been changed from 12 (other) to 9 (ignored), so the commentary is printed in the same font as the transcription. If I want the commentaries to be printed in a different font, I can code something like this:

```
\catcode'\*=\active
\font\ssf=cmss10

\def\startcommentary{\begingroup\ssf}
\let\finishcommentary=\endgroup
\let\docommentary=\startcommentary

\def*\{\docommentary
\ifx\docommentary
\startcommentary
\global\let\docommentary=%
\finishcommentary
\else
\global\let\docommentary=%
\startcommentary
\fi}
```

This makes `*` an active char whose expansion switches back and forth between `\startcommentary` and `\finishcommentary`. The example above then comes out looking like this:

```
herbergi heilagf anda. This passage is particularly
interesting œllvm
helgum mœnnum æðr i. helgari
a haleitari. er komin at kÿn
```

If you want `*` to do more than just change the font, you can put more code into the expansions of `\startcommentary` and `\finishcommentary`. The macros `\begincommentary` and `\endcommentary` are both `\let` to `*`, so using them is exactly the same as using `*`. They can be useful for marking longer commentaries, e.g.,

```
herbergi heilag{\slong} anda.~{\oe}llvm\\
\begincommentary
\<This commentary is so long that it's
nice to mark it with
{\tt\string\begincommentary}
and {\tt\string\endcommentary}
to make it easier
to see in the input file, since \* and \*
might be easy to overlook.\>
\endcommentary\space
helgum m{\oe}nnum {\ae}{\dh}r
i.~helgari\\
{\tirok} haleitari.~er komin
at k{\ydot}n\\
```

```
herbergi heilagf anda. œllvm
\begincommentary
\endcommentary
\<This commentary is so long that it's nice to mark
it with \begincommentary and \endcommentary
to make it easier to see in the input file,
since * and * might be easy to overlook.\>
helgum
mœnnum æðr i. helgari
a haleitari. er komin at kÿn
```

The explicit `\space` following `\endcommentary` is necessary, because ordinary spaces following `\endcommentary` would just be swallowed up in the usual way. It would be easy enough to change by redefining `\finishcommentary`, so that it always inserts a space into the current list.

```
\def\finishcommentary{\endgroup\space}
```

However, it might be desirable to have commentaries end within a word sometimes.


Ignored lines.

- Lines that begin with `%` are ignored both by \TeX and `conctex.lsp`. (A `%` in the middle of a line causes both \TeX and `conctex.lsp` to discard the rest of the line.)
- Entirely blank lines are processed normally by \TeX (the second `\return` in a row is converted to `\par`), and ignored by `conctex.lsp`.
- Lines that begin with `\` are treated in the normal way by \TeX and ignored by `conctex.lsp`, with a few exceptions. This makes it possible to include *undelimited macros* (control sequences that are not surrounded by braces) in the input file without worrying about how the Lisp program is affected by them. These can be skips, font changes or other commands. Beginning a line with `\relax` or `\empty` makes it possible to type anything in the input file and have `conctex.lsp` ignore it.



Certain undelimited macros, such as `\lineno` and macros like `\putontop` and `\overstroke` that are used for text, do not cause `conctex.lsp` to ignore a line, so they can appear at the beginning of text lines. Other exceptions may be programmed.

Evaluated lines. Lines in the input file can contain Lisp code to be evaluated by `conctex.lsp`. The first non-blank character in these lines must be the commercial “at” symbol, `@`. The `\catcode` of `@` has been changed to 14 (comment), so \TeX ignores these lines. An evaluated line should contain a complete balanced expression (*s-expression* or *sexp*); multiline Lisp expressions are not permitted in the input file. If an `@` is not the first non-blank character in a *text line*, both \TeX and `conctex.lsp` discard the rest of the line following the `@`, and Lisp does not evaluate it.

 The most common use of evaluated lines is to reset the *text position* for a new leaf, side or column.

```
@ (set-position "28va")
```

They can also be used for adding occurrences for words which T_EX can't format in the normal way, perhaps because a word is written vertically and extends over multiple lines (see page 398).

```
@ (add-occurrences "drotning" "1va~1-7")
```

The character ¥. The program `conctex.lsp` considers all letters (`\catcode 11`) and some characters of type “other” (`\catcode 12`) in text lines as “word elements”. Some “other” characters are considered “word separators”: in particular, blanks and punctuation. The character ¥ (decimal 165, octal 245, hexadecimal A5)⁶ is a special kind of word separator: it is ignored by T_EX (`\catcode 9`). The program `conctex.lsp` will ignore most lines that begin with an *undelimited macro*, but if the line begins with ¥, `conctex.lsp` will process it.

```
¥\vskip12pt DROTNING himins ok
iar{\dh}ar ...
```

The character ¥ may look different on your terminal.⁷ However it looks, it should always be used rather than `^^a5` in your input file;⁸ T_EX will recognize `^^a5` as referring to a single token, but `conctex.lsp` will treat it as a string of 4 characters. It's never necessary to use ¥, it is merely a convenience.

Curly braces. As far as plain T_EX is concerned, the characters { and } are simply set to categories 1 and 2, beginning and end of group, respectively. They can be set to other categories, and other characters can be set to categories 1 and 2. ConcT_EX doesn't support this generality; the `\catcodes` of { and } should not be changed

⁶ It's sometimes convenient to know the decimal, octal or hexadecimal notation for an integer in another of these radices. ConcT_EX includes a lagniappe, a C program that converts between decimal, octal and hexadecimal integers easily and quickly, and several Emacs-Lisp functions for calling this program from within Emacs.

⁷ If you're using Emacs, you can enter any character by typing `Control-q` followed by an octal integer, so I can type `Control-q 245` to get ¥. You may get another character, though, depending on your editor, operating system, etc. A better way to type in special characters is to define a key sequence or an abbreviation to do it. There's more about this in the documentation supplied with ConcT_EX.

⁸ Any character can be represented in a T_EX file as `^^` followed by two lowercase hexadecimal digits (*The T_EXbook*, p. 45). If necessary, `conctex.lsp` can be made to recognize this notation.

and other characters should not be used in their place. The reason for this is that the program `conctex.lsp` uses { and } explicitly in strings. It would be possible to change this and have a general mechanism for recognizing a beginning-of-group and an end-of-group character, but I did not consider that this was worthwhile.

Braces are used in *transcription lines* for their normal purpose: to delimit macros and their arguments. T_EX normally permits “unmotivated braces”, i.e., braces that are typed into the input file for no particular reason.

```
Th{is} line has {unm}otiv{ated} br{ace}s.
⇒
```

This line has unmotivated braces.

The unmotivated braces will only have an effect if they separate parts of a ligature, as in `waf{f}le` → waffle, or prevent kerning, as in `{V}A` → VA. Sometimes, of course, as in `{shelf}ful` → shelfful,⁹ this is useful, but then the braces aren't unmotivated. ConcT_EX does not permit them, and the function `letter-function` in `conctex.lsp` will signal an error when it finds an unmotivated }. In cases like “shelfful”, where the word looks better without the ligature, are desired, they must be accounted for in `conctex.lsp`.

Changing fonts. Manuscript transcriptions often require frequent font changes, sometimes within a single word. For example, editorial emendations may be printed in italics: `prestr` (Engl. “priest”), and perhaps enclosed in brackets, too: `p[re]str`. Some letter forms may be represented in the transcription as small capitals, such as the “R” in `p[re]str`. Different fonts may be used for various other purposes, too, according to the book design. For instance, initials, large and/or decorative letters, and letters written in a different color ink in the manuscript may all be indicated by a special font in the transcription. Font changes are handled in the normal way by T_EX and discarded by `conctex.lsp`. Font changes can extend over multiple transcription lines.

Special characters and other macros. A fundamental decision in editing a manuscript transcription is, to what degree the transcription should correspond to the actual appearance of the manuscript, e.g., how special characters are represented, whether abbreviations are expanded, etc. With T_EX and METAFONT, it is possible to imitate the appearance of the manuscript to a greater degree than is possible with other methods; however it is up

⁹ *The T_EXbook*, pp. 19 and 306.

to the editor and the book designer whether to make use of this capacity or not. I believe that a transcription will usually be of greater interest if it's not normalized, and if the abbreviations are not expanded. No matter what style of transcription is chosen, a wide range of special characters is usually required.

In T_EX, special character macros can be typed with or without enclosing braces, e.g., the word “ætt” (Engl. “a quarter of the heavens, one's family”)¹⁰ can be coded as “{\ae}tt” or “\ae_{tt}”, or even “\ae_{tt}” since spaces following a control word are ignored. The second and third alternatives are impractical even under normal circumstances, because it is unclear in the input file that “\ae” and “tt” belong together. In ConcT_EX, however, all special character codings must be enclosed in braces (“{\ae}tt”). T_EX will handle “\ae_{tt}” and “\ae_{tt}” in the normal way, but `conctex.lsp` will signal an error when it reads “\ae” without enclosing braces.¹¹ All macros used in transcription lines must be accounted for in `conctex.lsp`. Others will cause an error.

In most medieval manuscripts, words are often abbreviated. There are several methods of abbreviation. One is to write some of the letters of a word and put a stroke over them, like “mm” for “mōnnum.” It may be desirable to expand these abbreviations in the transcription. One way of doing this is to print out the characters that do not appear in the manuscript, but in italics and underlined, e.g., “mōnnum”. (Other solutions may be used according to the book design.) The macro `\ustroke` is used to do this:

```
m{\ustroke{{\ohook}nnu}}m
⇒
mōnnum
```

You can type “`m{\ustroke{{\ohook}nnu}}m`” or “`m\ustroke{{\ohook}nnu}m`”, i.e., the macro `\ustroke` can be delimited or undelimited. Either way, `conctex.lsp` discards the macro and its braces, and the argument is treated as part of the word, so in this example, “mōnnum” is printed in the output and “mōnnum” will appear in the concordance (under the main form “maðr”, Engl. “man”).

Since such abbreviations occur frequently, the `\catcode` of the underline character `_` (decimal 95, octal 137, hexadecimal 5F) has been reset to `\active` and `\let` to `\ustroke`. So now you can type “`m_{\ohook}nnu}m`” to get “mōnnum”, which

¹⁰ CLEASBY-VIGFUSSON, p. 760.

¹¹ It is `letter-function` that signals the error, when it reads the `\`.

makes the input file somewhat easier to type and read. The `\catcode` of `_` is reset to 8 (subscript) in math mode, so it's available for making subscripts, and also in the `\plain` environment, so that it's possible to load files with the character `_` in their names. But it's not reset in commentaries, which might very well want to use `_` for `\ustroke`.



The macro `\ustroke` is defined as follows:

```
\def\ustroke#1{%
\def\subustroke{\leavevmode
\ifx\next.% It's a period
\setbox0=\hbox{{\it#1}}\else
\ifx\next,% It's a comma
\setbox0=\hbox{{\it#1}}\else
% It's neither a period nor a
% comma
\setbox0=\hbox{{\it#1}/}}%
\fi\fi
$\underline{\box0}$}%
\futurelet\next\subustroke}
```

Since the underlined text is put in italics, it's nice to have `\ustroke` insert the italic correction (`\)` automatically, if and only if `\ustroke`'s argument is followed by something other than a period or a comma. In order to find this out, it's necessary to peek at the following token, using `\futurelet`. If `\ustroke` uses a non-slanted font, it can be defined more simply. It uses the `\underline` macro, which is available only in math mode. This makes the underlines go below the bottom of the lowest character in `\ustroke`'s argument.

```
_ {ypq} _ {abc}
_{\ehook}{\ohook}{\oehook}}
⇒
ypq abc eqq
```

It might be nicer to have the underlines all at the same depth, preferably close to the baseline, but unfortunately, this doesn't turn out to look very good. If `\ustroke` is defined like this:

```
\def\ustroke#1{%
\def\subustroke{\leavevmode
\ifx\next.% It's a period
\setbox0=\hbox{\it#1}\else
\ifx\next,%
% It's a comma
\setbox0=\hbox{\it#1}\else
% It's neither a period
% nor a comma
\setbox0=\hbox{\it#1/}\fi\fi
% This makes a .25pt rule.
\hbox to 0pt{\vrule height -.8pt
```


```

        depth 1.05pt
        width \wd0\hss}\box0}%
\futurelet\next\substroke}
then
  _{ypq} _{abc}
  _{{\ehook}{\ohook}{\oehook}}
⇒
  ypq abc eoo

```

I think it looks worse to have the underline stroke go through the descenders of *y*, *p*, and *q*, and the ogoneks (̇) of *ę*, *ø*, and *ø*, than it does to have the underline stroke be at different heights. To do this properly, it would really be necessary to design fonts with the underline stroke included in the individual letters.¹²

Sometimes manuscript transcriptions will require the use of special characters which are not available in existing fonts. It may be possible to create them by manipulating existing sorts.¹³ For example, in Holm perg 11 4^o, many words have letters with smaller letters placed over them. A logotype¹⁴ $\overset{b}{a}$, coded as `{\bOVERa}`, can be defined using the existing letters “a” and “b”, boxes, and glue. For other sorts, like \mathfrak{z} , the Tironian symbol for Latin “et” (“and” in English and “ok” in Old Icelandic), it may be necessary to program a font using METAFONT.

 `\bOVERa` is actually defined as `\putontop{a}{b}{-}{-}`. The macro `\putontop` is defined like this:

```

1. \def\putontop{\begingroup
2. \catcode\-=12
3. \def\subputontop##1##2##3##4##5{%
4. \setbox1=\hbox{##1}%
5. \setbox3=\hbox{##3}%

```

¹² Cf. *The T_EXbook*, p. 323.

¹³ “Sort” is a technical term for a typographical unit, synonymous with “character”. The term “character” is ambiguous in the context of T_EX, because the characters in the input file differ in nature from the characters in the fonts used for typesetting, and the coding of the latter in terms of the former is subject to modification. Therefore, I sometimes prefer the unambiguous term “sort” for a character when I mean a typographic unit belonging to a font. WILLIAMSON defines a character or sort as a “single figure, letter, punctuation mark, symbol or word-space cast as a type or generated by photocomposition, CRT [cathode-ray tube] or digital system, or typed.” (368).

¹⁴ WILLIAMSON defines “logotypes” (or “logos”) as follows: “letters joined to each other & cast on a single shank.” (p. 379.) He also notes that there is a shortage of logotypes in some photocomposition systems, where they are replaced by separate characters.

```

6. \setbox4=\hbox{##4}%
7. \setbox5=\hbox{##5}%
8. %
9. %% These are the default dimensions.
10. %% For shifting the top letter
11. %% to the right or left:
12. \dimen3=-.8\wd1
13. %% For shifting the top letter
14. %% up or down:
15. \dimen4=1.2\ht1
16. %% For increasing or decreasing
17. %% the kern following \putontop:
18. \dimen5=0pt
19. %
20. \ifdim\wd3>0pt
21. \advance\dimen3 by ##3\fi
22. %
23. \ifdim\wd4>0pt
24. \advance\dimen4 by ##4\fi
25. %
26. \ifdim\wd5>0pt
27. \advance\dimen5 by ##5\fi
28. %
29. \setbox2=\hbox{\kern\dimen3
30. \raise\dimen4
31. \hbox{\small ##2}}}%
32. \leavevmode\box1
33. \hbox to 0pt{\hss\box2\hss}%
34. \kern\dimen5\endgroup}\subputontop}

```

The macro `\putontop` takes its second argument and puts it above its first argument in a smaller size. The third and fourth arguments can be empty, or they can be used to shift the position of the second argument. The fifth argument can also be empty, or it can be used to increase or decrease the space following `\putontop`. The first and second arguments to `\putontop` need not be single characters. Here are some examples of using `\putontop`.

```

\putontop{abc}{def}{-}{-} ghi
⇒
def
abc ghi

```

This shifts the raised letters to the right.

```

\putontop{abc}{def}{10pt}{-} ghi
⇒
def
abc ghi

```

This shifts them down.

```

\putontop{abc}{def}{-}{-20pt}{-} ghi
⇒
abc ghi
def

```

This increases the kern following “abc”.

```
\putontop{abc}{def}{-}{-}{20pt} ghi
```

⇒

```
  def
abc    ghi
```

And this reduces it.

```
\putontop{abc}{def}{-}{-}{-10pt} ghi
```

⇒

```
  def
abghi
```

At the beginning of `\putontop`, the `\catcode` of `-` is reset temporarily to 12 (other). This makes it possible to use negative dimensions in its third, fourth, and fifth arguments.

◊ If a certain combination of letters like $\overset{b}{a}$ appears frequently in a transcription, it’s easier to define a macro like `\bOVERa` for it rather than using `\putontop` explicitly over and over. The function `replace-items` can replace `{\bOVERa}` with “ab” or any other string, so that a word like “ $\overset{b}{r}$ abit” will appear in the concordance as “rabbit”. Another possibility is to account for `{\bOVERa}` in `letter-function`. In this case, “ $\overset{b}{r}$ abit” will be written to the concordance, unless “`r{\bOVERa}bit`” has been specified as a variant of “rabbit” in the lemmatization dictionary.

```
(generate-entry "rabbit"
 :variants "r{\bOVERa}bit")
```

◊ If a word like “eptir” (Engl. “after”) is abbreviated as `\putontop{e}{p}{-}{-}{2pt}{-}tir` → “ $\overset{p}{e}$ tir” in the input file, the first two arguments shouldn’t be discarded, because they belong to the word. However, the string “`\putontop`” and its other arguments, whether they’re empty or not, will cause an error in `letter-function`, if they reach it. The function `process-macro` inside the function `discard-items` discards the string “`\putontop`”, takes the first two arguments out of their braces and discards the rest.¹⁵

◊ This works, if words using `\putontop` always conform to this pattern. When they do, `\putontop` can be used explicitly in the input file. However, when they don’t, a different approach is required. Sometimes, for instance, letters that are put over other letters stand for parts of the word which are left out. If the word “drotning” is abbreviated frequently in the manuscript as “ $\overset{r}{d}$ ”,

¹⁵ The function `process-macro` receives two arguments which tell it which arguments to take out of their braces and which to discard.

then it would make sense to define a macro `\dr` as follows:

```
\def\dr{\putontop{d}{r}{-}{-}{-}}
```

Then, `{\dr}` can be made a variant of “drotning” in the lemmatization dictionary.

```
(generate-entry "drotning" :variants "{\dr}")
```

or

```
(add-variants "drotning" "{\dr}")
```

Alternatively, the function `replace-items` can replace all occurrences of “`{\dr}`” in the input file with “drotning” before `current-line` is passed to `process-line`. Note that “`{\dr}`” should always appear within braces, like the special character macros.¹⁶

If there are a lot of cases of this type, and the transcription does not expand the abbreviations, then it would be a good idea to add another argument to `\putontop`.

```
\putontop{ma{\dh}r}{m}{r}{-}{-}
```

In this example, the word $\overset{m}{ma}r$ (Engl. “man”) is abbreviated as $\overset{m}{m}$. Then, `discard-items` can call `process-macro` in such a way that the first argument, containing the whole word as it should appear in the concordance, is taken out of its braces so that it reaches `process-line` and `read-word`, and the other arguments are discarded. In this case, `\putontop` should discard the first argument when `TEX` is run, so that only the abbreviation is printed to the output.

◊ If the word “ $\overset{m}{ma}r$ ” appears frequently in the manuscript, which is likely, it might make sense to define a macro as follows:

```
\def\madhr{\putontop{}{m}{r}{-}{-}}
```

Here, the first argument can be empty, since `TEX` ignores it. The string “`{\madhr}`” must be made a variant of “ $\overset{m}{ma}r$ ” in the lemmatization dictionary or replaced by “`ma{\dh}r`” in the function `replace-items`, as above. The macro `\madhr` can be used in words like “`{\madhr}inn`” → “ $\overset{m}{m}inn$ ” for “ $\overset{m}{ma}drinn$ ” (nominative singular with enclitic article). Cases where a single special character is used to represent a frequently used word, like `{\hxbarhk}` → “ $\overset{h}{h}$ ” (abbreviation for “hans”, Engl. “his”, personal pronoun, masculine singular genitive) and `{\thxbarhk}` → “ $\overset{h}{\theta}$ ” for the word

¹⁶ The control word `\dr` could even be accounted for in `letter-function`, which would make “ $\overset{r}{d}$ ” appear in the concordance. If it was assigned the list (d-value r-value o-value t-value n-value i-value n-value g-value), it would even be alphabetized correctly. I don’t think this would be useful for a concordance, but it might be for some other application.

“þessu” (demonstrative pronoun, feminine singular dative), can be handled in the same way.

An advantage in using \TeX is the ability to use temporary definitions for special characters. If I hadn’t programmed \mathfrak{z} yet, I could define it as follows:

```
\def\tirok{\&\$^{\rm Tir.}\$}
```

Then, a line like the following:

```
{\tirok} haleitari.~er komin at
k{\ydot}n
```


will be printed like this:

```
&Tir. haleitari. er komin at k˙n
```

When I’ve gotten around to programming \mathfrak{z} , it will be printed like this:

```
a haleitari. er komin at k˙n
```

but I won’t have to change my input file.¹⁷

 This is the \TeX code that’s necessary for using special characters like \mathfrak{z} , assuming the font is called `specialfont`.

```
\font\specialfontnine=specialfont9
\font\specialfonttten=specialfont10
\font\specialfontttwelve=specialfont12
...
```


Now, `\specialfont` must be defined where `\rm`, `\large`, `\small` etc. are defined, so that `\specialfont` accesses the correct size of the font `specialfont`, e.g.

```
\def\rm{\let\specialfont=%
\specialfonttten ...}
\def\small{\let\specialfont=%
\specialfontnine ...}
\def\large{\let\specialfont=%
\specialfontttwelve ...}
```

Finally, `\tirok` is defined like this:

```
\def\tirok{{\specialfont\char'140}}
```

The character “ \mathfrak{z} ” is in position octal 140 (decimal 96, hexadecimal 60) in `specialfont`.

 In a similar way, \TeX makes it possible to distinguish words in the input file which should not be distinguished in the output. For instance, there are two forms of “m” in Holm perg 11 4° which are used interchangeably. Depending upon the degree of normalization desired in the transcription, these two letters can be distinguished in the output or not, as desired. One form can be coded as `m` →

“m” and the other as `{\mone}` (for “m-one”) which also prints as “m” and is defined as

```
\def\mone{m}
```

The function `replace-items` can convert each occurrence of the string “`{\mone}`” in the input file to “m” before current-line is passed on to process-line, so `{\mone}` will never appear in the concordance. If, later, a different letter form should be used to represent `{\mone}` in the output, `\mone` can be redefined, e.g.,

```
\def\mone{{\specialfont\char'001}}
```

Math mode material is always treated in the normal way by \TeX . It is treated in different ways by `conctex.lsp`. Display math material is processed normally by \TeX and discarded by `conctex.lsp`.¹⁸ Math mode is very likely to appear in transcription lines, mainly for sub- and superscripts and certain special characters. Sometimes, the math mode material will be irrelevant for the concordance, but at other times it may be an essential part of a word, or even serve to distinguish between two otherwise identical words. If math mode material is attached to a word, that is, if it isn’t separated from the word by blank space or another word-separator (like most punctuation marks), it is considered to be part of the word. If not, `conctex.lsp` discards it. For example,

```
drotning\pi
```

is treated as a word, “drotning ^{π} ”. It will appear as such in the concordance and will be a separate entry from “drotning”, if this word occurs in the transcription. If this appears in the transcription,

```
drotning\pi
```

`conctex.lsp` will discard the math mode material, “drotning _{π} ” will appear in the output and “drotning” will appear in the concordance. To have `conctex.lsp` discard the math mode material, but not separate it from the word in the output, type:

```
drotning\pi
```

In this case, `conctex.lsp` also discards the math mode material, and “drotning” appears in the concordance, but “drotning ^{π} ” appears in the output, because \mathfrak{z} is ignored by \TeX (`\catcode'\mathfrak{z}=9`), and treated as a word separator by `conctex.lsp`. The math mode material can also be put on the next input line following a %.

```
drotning%
\pi himins ...
```

¹⁷ \TeX is generous with space for storing macros; there’s room for over 2000, so there should be enough for all the special characters you need.

¹⁸ The program `conctex.lsp` converts `$$` to `*`, so display math mode material is treated as a commentary.

⇒

drotning^π himins ...

and “drotning” appears in the concordance.

The `\catcodes` of some characters, such as `-`, `_` and `<`, that have been changed for use in transcription lines, should have their normal values in math mode. The token list `\everymath` resets them.

```
\everymath={\catcode'\-=12\catcode'\_ =8
\catcode'\*=12\catcode'\<=12\relax}
```

The `\relax` at the end of `\everymath` is necessary because assignments, such as changes to `\catcodes`, only take effect when T_EX is building a list.¹⁹ The same effect could be achieved with `\vbox{}` or `\hbox{}`, or a harmless macro (but not `\empty`).

⚡ Math mode material should not extend over more than one transcription line. If it does, `conctex.lsp` will signal an error. Normally, it shouldn't be necessary anyway, since most manuscripts don't contain multiline equations. Even if yours does, the elements of the equation probably don't belong in the concordance anyway, so the multiline math mode material can be put in a commentary. In this case, it may be necessary to reset the line number explicitly with `set-position`. If your application requires a lot of math mode, it would be advisable to reprogram the routines for handling it. Display math mode material is treated as a commentary, so it can extend over any number of lines.

Line ends. The treatment of line ends is an important factor in ConcT_EX. A typeset manuscript transcription will generally have lines corresponding to the lines in the manuscript, so it is undesirable to have T_EX do the line breaking for the text lines. (It would also be a lot of work to write a hyphenation dictionary for a medieval manuscript.) However, it will rarely if ever be desirable to use `\obeylines`, because there are many other items which can appear in the input lines, making it impractical for the lines in the input file to correspond to the lines in the output: font changes, index entries, footnotes, comments, etc. Therefore, the ends of transcription lines must be indicated.

Most transcription lines should end in `\`. The control symbol `\` is `\let` to `\par` in `conctex.tex` and recognized as a line end by `conctex.lsp`. The program `conctex.lsp` keeps track of the line numbers, so this is important. A transcription line

¹⁹ *The T_EXbook*, p. 373.

in the input file that is followed by a blank line is treated as if it ended in `\`.

Normally, the lines in the output should correspond to the lines in the manuscript, so they should be wide enough, i.e., the value of `\hsize` should be large enough so that transcription lines don't have to be broken. In exceptional cases, however, this may not be possible, for example, if a line includes a long commentary.

```
\lineno{8} himins _{ok} *{\<{\ss This
is an extremely long commentary which
causes this otherwise short line to
be broken so that it results in more
than one line in the output.}\>*
iar{\dh}ar.~s{\ae}l {ok} dyr{\dh}\-
\lineno{9} lig m{\o}r Maria.~mo{\dh}ir
d_{fro}tti_{n}s\
```

⇒

8: himins ok (This is an extremely long commentary which causes this otherwise short line to be broken so that it results in more than one line in the output.) iarðar. sæl ok dyrð
9: lig mør Maria. moðir drottins

⚡ The control symbol `\` expands to `\par` and the value of `\parskip` and `\parindent` have been set to `0pt` in order that multiple blank lines won't cause excessive vertical space or unwanted indentation in the output. If `\` expanded to `\hfil\break`, a blank line following `\` would cause two `\baselineskips` to appear in the output, one from the `\break` and one from the second `^M` (`\return`), which is converted to `\par`. On the other hand, a `\par` following a `\par` is harmless.²⁰ The text in manuscripts is generally not divided into paragraphs, so it's unnecessary to use blank lines to indicate paragraphs, and it's convenient to allow them for the sake of making the input file more readable.

⚡ The control symbol `\` is not needed in the `\plain` environment, because T_EX does the line breaking there. The values of `\parskip` and `\parindent` can also be changed in the definition of `\plain`, for text that should be formatted differently from the transcription itself.

Manuscript positions are defined according to leaf (folium), side (recto or verso), column (if there are multiple columns), and line number. The

²⁰ The first `\par` puts T_EX into vertical mode, where the second `\par` has no effect, “except that the page builder is exercised ... and the paragraph shape parameters are cleared.” (*The T_EXbook*, p. 283.)

leaves of Holm perg 11 4° and AM 234 fol have two columns, so a position is identified as follows: 2ra 17 is leaf 2, recto, column a, line 17. When a new leaf, side or column begins, the input file must contain a line that looks like this:

```
@ (set-position "28vb")
```

If that column starts with line 15, perhaps because the top of the leaf is missing or unreadable, the line should look like this:

```
@ (set-position "28vb" 15)
```

Lines beginning with @ are ignored by T_EX and evaluated by `conctex.lsp`. The Lisp program counts the lines and keeps track of the position in the manuscript. However, it will usually be useful to indicate line numbers in the transcription itself, so a line in the input file might look like this:

```
31: f_{ra} savgn_{n} Mathevs
```

In the output, it will look like this:

```
31: fra savgn Mathevs
```

However, “31” should not appear as a lemma in the concordance. Therefore, `conctex.lsp` discards numbers, punctuation and blanks at the beginning of transcription lines. Another possibility is to use the macro `\lineno` to make them print or not, according to the value of a conditional.

```
\ifprintlinenumbers
\def\lineno#1{...}\else
\def\lineno#1{\relax}\fi
\lineno{1} Drotning etc.
```

```
\newif\ifprintlinenumbers
\printlinenumberstrue
```

⇒

```
1: Drotning etc.
```

```
\printlinenumberfalse
```

⇒

```
Drotning etc.
```

The macro `\lineno` is an exception to the rule about `conctex.lsp` discarding lines that begin with undelimited macros. It can be defined as follows:

```
\ifprintlinenumbers
\def\lineno#1{\leavevmode
\setbox0=\hbox{33:\space}%
\hbox to \wd0{\hss#1:\space}%
\hangindent\wd0\hangafter 1}
\else
\def\lineno#1{\relax}\fi
```

Line numbers, whether explicit or in `\lineno`, are discarded by `conctex.lsp`. Since `\`, `\-` and `-`

(explained below) and a blank line all cause a `\par` to be added to the current list, `\lineno` always begins a paragraph (assuming `\printlinenumbers-true`). Usually, this paragraph will consist of one output line, just as it corresponds to one line in the manuscript, but if the paragraph is longer than one line, the `\hangafter` and `\hangindent` macros cause the following lines to be indented so that they begin directly below the text of the first line in the paragraph.

◆ The same effect could be achieved by using the token list `\everypar`, and separate `\everypars` could be maintained for the `\plain` and `\trans` environments.

◆ The “magic number” 33 in `\box0` in the definition of `\lineno` is there simply to make the box an appropriate size. Numbers wider than “33” extend into the left-hand margin, so the colons (or periods, or whatever) always line up.

In some applications, positions may not require line numbers. For instance, positions in a concordance of the Bible should be given as book, chapter and verse. In this case, T_EX could take care of the line breaking, and `\` could be used to indicate the end of a verse. It could be `\let` to `\relax` in `conctex.tex` and cause `conctex.lsp` to increment a “verse-counter” instead of line-counter (explained below). The exact way that ConcT_EX handles line numbers in the input file can and should be set for each particular project.

◆ While it is convenient to define macros in this way:

```
\ifprintlinenumbers
\def\lineno#1{...}\else
\def\lineno#1{\relax}\fi
```

it is sometimes preferable to define them like this:

```
\def\lineno#1{\ifprintlinenumbers
... \else \relax \fi}
```


or like this:

```
\def\linenoprint#1{...}
\def\linenonoprint#1{\relax}
```

```
\ifprintlinenumbers
\let\lineno=\linenoprint
\else \let\lineno=\linenonoprint
\fi
```

The advantage of the last two alternatives is, that it is possible to put the definitions of `\linenoprint` and `\linenonoprint` in a file

used for generating a preloadable format file with INITEX. If you try to generate a format file using the first example, where the definitions themselves are within the conditional construction, one version will be skipped when INITEX runs, depending on the current value of `\ifprintlinenumbers`. However, in the second example, the conditional construction is within the definition, so it will expand according to the value of `\ifprintlinenumbers` when `\lineno` is called. This means that the expansion of `\lineno` can switch back and forth during the T_EX run. In the third example, the two definitions are assigned to two different macros, and `\lineno` is `\let` to one of them according to the value of `\ifprintlinenumbers` at the time. This conditional construction can be put in an ordinary macro file, loaded after the preloaded format. The assignment must occur before `\lineno` is called for the first time. A subsequent change to the value of `\ifprintlinenumbers` later in the T_EX run has no effect on the expansion of `\lineno`. You can, however, switch the definition explicitly by saying `\let\lineno=\linenoprint` or `\let\lineno=\linenonoprint` at any time. If you have a lot of macros, it can be useful to create a preloadable format in this way.²¹

 The Lisp program doesn't use the explicit line numbers in the input file for its line counting routine, although it could. Explicit line numbers are optional, so `conctex.lsp` needs its own line counting routine anyway. It does, however, check that the line numbers from its routine and the explicit line numbers in the input file match up. If they don't, it issues a warning, but doesn't signal an error. Warnings are written to standard output when `conctex.lsp` is run, but also to the file `warnings`. If `conctex.lsp` is set up to write code like this to `warnings`:

```
echo "At line 21 of input.tex:
Line numbering is incorrect.
(\\lineno: 12) (line-counter = 14)"
```

and `conctex.lsp` is run using a UNIX shell script, then the shell script can execute `warnings` by calling `sh warnings`, causing all of the warnings to be printed to standard output after `conctex.lsp` is done.

Broken words. Some lines may end in words that are continued on the next line. In AM 234 fol, some of the broken words have hyphens and some do not. It is necessary to distinguish between these two cases in the input file. I use - (hyphen) to indicate

a broken word with an explicit hyphen and `\-` to indicate a broken word with no hyphen. Using `\-` indicates broken words in the input file, and causes `conctex.lsp` to treat such a broken word as a unit, but no hyphen appears in the typeset output. For example:

```
tok j vpphafi {\slong}n{\slong}
Gv{\dh}{\slong}pia-
llz.~at telia {\ae}tt drottin{\slong}
ie{\slong}us
```

⇒

```
tok j vpphafi fnf Gvðfpia-
llz. at telia ætt drottinfnf iefus
```

Whereas

```
tok j vpphafi {\slong}n{\slong}
Gv{\dh}{\slong}pia\
llz.~at telia {\ae}tt drottin{\slong}
ie{\slong}us
```

⇒

```
tok j vpphafi fnf Gvðfpia
llz. at telia ætt drottinfnf iefus
```

If a line in the input file ends in - or `\-`, this obviously indicates the end of a line in the manuscript, so it would be redundant to type `-\` or `\-\`.²²

The program `conctex.lsp` recognizes - and `\-` at the end of a line as line ends for its line numbering routine. T_EX also recognizes them as line ends at the end of a line and inserts a `\`. However, - can also occur within a line, or even at the beginning of a line. Here it should not be recognized as a line end, either by T_EX or by Lisp. A - at the beginning of or within a line is simply printed as - (what else?). As far as T_EX is concerned, a `\-` within a line is harmless, but doesn't make any sense, since it doesn't print anything. Therefore, T_EX issues a warning but doesn't signal an error. The program `conctex.lsp` discards `\-`s that are neither at the end of an input line nor followed by `\`. The strings -- and --- still yield - (en-dash) and — (em-dash) respectively, but they do not cause line ends, either in T_EX or in Lisp.²³

The following code is necessary to accomplish this:

```
1. \catcode'\@=\active
2. \let@=\-
3.
```

²² Although it's redundant, it is nonetheless permitted.

²³ If a concordance is to be generated for a manuscript which uses more characters to indicate broken words, like = or =, both `conctex.tex` and `conctex.lsp` must be modified to account for these cases.

²¹ *The T_EXbook*, p. 344.

```

4. \def\hyphen{-}
5. \def\dash{--}
6. \def\Dash{---}
7.
8. \catcode\-=\active
9.
10. \begingroup
11. \catcode\^M=\active
12. \global\let^M=\empty
13.
14. \gdef\invisiblehyphen{
15. \begingroup\catcode\^M=\active
16. \def\subhyphen{\ifx\next^M\\else
17. \ifx\next\ % Do nothing
18. \else
19. \message{Ignoring \noexpand\-. %
20. Don't use \noexpand\-. %
21. in the middle of a line.}
22. \fi\fi\endgroup}
23. \futurelet\next\subhyphen}
24.
25. \global\let\-=\invisiblehyphen
26.
27. \gdef-{\begingroup\catcode\^M=\active
28. \def\aeathypens##1{
29. \futurelet\next%
30. \bEathypens}
31. \def\beathypens{
32. \ifx\next-\Dash
33. \expandafter\cEathypens\else
34. \dash\endgroup\fi}
35. \def\cEathypens##1{
36. \futurelet\next\endgroup}
37. \def\subhyphen{
38. \ifx\next^M\hyphen\\% It's a return.
39. \let\aeathypens=\endgroup\else
40. \ifx\next-% It's a hyphen
41. \let\aeathypens=%
42. \aeathypens\else
43. %% It's not a return or a hyphen.
44. \hyphen
45. \let\aeathypens=\endgroup
46. \fi\fi
47. \aeathypens}
48. \futurelet\next\subhyphen}
49.
50. \endgroup

```

The primitive `\-`, which is ordinarily used for inserting discretionary hyphens, must be redefined. Since the user decides where transcription lines are to be broken, discretionary hyphens are unnecessary there. In case they are needed, however, the character © (decimal 169, octal 251, hexadecimal

A9) is made active and `\let` to `\-` before `\-` is redefined (lines 1 and 2). Make sure you use © and not `^a9`. T_EX will treat `^a9` as a single token, but `conctex.lsp` will not, just as with `¥` and `^a5` (unless `conctex.lsp` is modified to allow this). In commentaries and the `\plain` environment, `\-` reverts to its primitive self.

T_EX's ligature routine won't work for `-`, `--`, and `---` after the `\catcode` of `-` has been changed to `\active` (line 8), so first I put `-`, `--` and `---` into the expansion of the new control words `\hyphen`, `\dash` and `\Dash`, so I can still use them. Now I define `-` and `\invisiblehyphen` and `\let\-` to `\invisiblehyphen`.²⁴ They check whether the token following `-` or `\-` is a `<return>`, ASCII code 13, which can be notated as `^M`, in which case `\-` should be inserted to break the line.²⁵

Under normal circumstances, i.e., when `\catcode\^M=5`, it wouldn't be possible to tell whether the character following an active char is a `<return>` or not. For example:

```

\def\abc#1{\show#1}
\abc
d
causes
> the letter d.
<argument> d
\abc #1->\show #1
to be printed to the screen, or standard output, to
be precise, and
\abc
d
causes an error.
?
Runaway argument?
! Paragraph ended before
\abc was complete.

```

²⁴ The macro `\invisiblehyphen` is defined in order to make it possible to switch the definition of `\-` back and forth when changing environments. This is necessary, because `\-` is already a control symbol (in fact, it's a primitive), whereas `-` has `\catcode 12` in the `\plain` environment, commentaries, and math mode. Therefore, the active character `-` can be defined globally. When the environment is switched, it suffices to change the `\catcode` of `-`.

²⁵ Incidentally, you might be wondering why I'm talking about `<return>`s. I use a computer running UNIX where the end-of-line character is `<newline>`, ASCII 10, and not `<return>`, ASCII 13. The reason is that T_EX converts the external coding scheme of the input file to its own internal coding (*The T_EXbook*, p. 43), so UNIX' `<newline>`s are converted to T_EX's `<return>`s.

```
<to be read again>
      \par
```

The first time `\abc` was invoked, it skipped the first `^^M`, which had been converted to a space, and found the “d” in the next line. The second time it was invoked, `\abc` skipped the first `^^M`, but found the second one, which had been converted to a `\par` token (*The T_EXbook*, p. 47). A normal macro can’t accept arguments that contain `\pars`, so an error was signalled. Here’s another version.

```
\long\def\abc#1{\show#1}
\abc
```

```
d
```

```
⇒
```

```
> \par=\par.
<argument> \par
```

```
\abc #1->\show #1
```

A macro defined using `\long\def` won’t signal an error if a `\par` occurs in its arguments, but the first `^^M` is still skipped. So simply reading in arguments is useless for discovering whether a macro or an active character is followed by a `^^M`. In addition, `\abc\par` will produce the same screen output as the last example, since T_EX can’t distinguish between an explicit `\par` and one that was converted from a `^^M`.

The same behavior can be demonstrated using `\futurelet`:

```
\def\abc{\futurelet\next\subabc}
\def\subabc{\show\next}
\abc
a
```

```
⇒
```

```
> \next=the letter a.
```

```
and
```

```
\abc
```

```
a
```

```
⇒
```

```
> \next=\par.
```

In order to check whether the character following a control sequence or an active character is a `^^M`, it is necessary to change the `\catcode` of `^^M`. One possibility is to change it to `\active`. I do this locally within the definitions of `-` and `\-`, so that `^^M` otherwise behaves normally. However, in order that the `^^Ms` within the actual definitions of `-` and `\-` are handled correctly while they’re being

read into T_EX’s memory, it is necessary that `\catcode\^^M=\active` and that `^^M` is set globally to `\empty` (line 11–12). It could also have been set to `\relax`.

Serendipitously, setting `^^M` to `\empty` makes it unnecessary to use `%` at the end of lines that end in `{` or `}`, such as lines 14, 21–22, etc. However, `%` is still necessary in the `\message` command in lines 19–20, otherwise there would be no space between “\.” and “Don’t” in the message, i.e., any trailing spaces before the `^^M` would be swallowed up. A `%` is also necessary in the `\futurelet` construction in line 29, otherwise `\next` would be set to `\bEathyphens`, the token following the `^^M`, which is expanded.

The way T_EX handles `^^M` at ordinary times, i.e., when `\catcode\^^M=5`, causes the global definition of `^^M` as a macro to be reset to `\par`. Therefore it’s necessary to set it globally in line 12, so that it expands to `\empty` in `-` and `\-`. Setting `\catcode\^^M=\active` in line 11 is only in effect inside the group that ends on line 50; when `-` or `\-` is invoked, *it* must begin a group and reset `\catcode\^^M` to `\active`. However, this being done, the `\global\let^^M=\empty` is in effect while `-` or `\-` is expanding. It doesn’t work to get rid of the `\global\let\^^M=\empty` and put `\let\^^M=\empty` in the definitions of `-` and `\-`. I’m afraid I don’t understand why not, but it is apparently connected with peculiarities of category code 5.

Since the definitions of `-` and `\invisiblehyphen` are inside of a group, they must be `\gdefs` (global definitions), or they wouldn’t be available outside the group. The same applies to `\-`, which is `\let` globally to `\invisiblehyphen`. The definitions inside `-` and `\invisiblehyphen`, however, are local to the groups begun in these macros.



Another possibility is to set `\catcode\^^M=\active` and `\let^^M=\empty` globally (i.e., get rid of the group begun in line 10 and ended in line 50, change `\gdef` and `\global\let` to ordinary `\def` and `\let`, and reset `\catcode\^^M=5` after the definitions.

If `-` were defined like this:

```
\gdef-#1#2{...}
```

i.e., with arguments, it wouldn’t be possible to check whether the tokens following `-` in the input file were `^^M` or not, because the arguments would have been read in and expanded before `-` had had a chance to change `^^M`’s `\catcode`. Once a token has been read in as an argument this is no longer possible. So `-` and `\invisiblehyphen` change the `\catcode` of

`^^M`, peek at the following token using `\futurelet`, and turn over control to `\subhyphen`. Both `-` and `\invisiblehyphen` have their own version of `\subhyphen`.

In `\invisiblehyphen`, `\next` contains the token following the `\-` in the input file. This token is examined by `\subhyphen`. If `\next` is `^^M`, `\subhyphen` inserts `\` to break the line. If it's `\`, `\subhyphen` does nothing (`\-` is redundant but permitted in the input files). If it's anything else, `\subhyphen` issues a warning that `\-` shouldn't be used in the middle of a line. Then `\subhyphen` ends the group begun by `\invisiblehyphen` in line 15.

The active character `-` functions in a similar, but more complicated way. It too uses `\futurelet` and `\next` to examine the following token.

⊥ If `\next` is `^^M`, `\subhyphen` inserts a hyphen with `\hyphen` (defined in line 4), a `\` to break the line, and `\lets` the control word `\eathyphens` to `\endgroup`, to end the group begun in line 27.

⊥ Else, if `\next` is not `-`, `\subhyphen` adds a `\hyphen` to the current list, but no `\`, and `\eathyphens` is `\let` to `\endgroup`.

⊥ Else, if `\next` is `-`, `\subhyphen` `\lets` `\eathyphens` to `\aEathyphens`.

Now, `\eathyphens` takes over control. In the first two cases, i.e., `\next` was `^^M` or anything else other than `-`, `\eathyphens` merely ends the group begun in line 27. However, if `\next` was `-`, matters are more complicated.

If `-` is followed by `-`, this could be `--`, which should be replaced with an en-dash, or `---`, which should be replaced with an em-dash. In neither of these cases should the line be broken. Therefore, it's necessary to peek at the token following the second `-`, but first we have to dispose of the second `-`, because it's not possible to peek at the next token but one, and the second `-` is still to be read by `TeX`'s parser. The macro `\aEathyphens` takes one argument, which disposes of the second `-`, then peeks at the following token using `\futurelet` and `\next`, and passes control to `\bEathyphens`. This macro examines `\next`. If it's not `-` (in the `\else` construction), `\bEathyphens` inserts a `\dash` into the current list, and ends the group begun in line 27. However, if `\next` is `-`, it inserts a `\Dash` into the current list and calls `\cEathyphens`. The `\expandafter` is necessary to prevent `\cEathyphens` from reading in `\else` as its argument. Instead, it reads in and thereby disposes of the third `-`. It could just end the group now, but it doesn't. It peeks at the token following the third `-` using `\futurelet` and `\next` yet again. This has

the effect that `--` or `---` at the end of a line is not followed by a blank space.

Normally, in plain `TeX`, or when using the `\plain` environment in `ConcTeX`, `--` or `---` at the end of a line is followed by a blank space.

```
... in lines 299--
547 ...
```

```
There was a sudden crash---
then silence.
```

⇒

```
... in lines 299_547 ...
```

```
There was a sudden crash_ then silence.
```

However, in the `\trans` environment, the spaces disappear.

```
... in lines 299-547 ...
```

```
There was a sudden crash—then silence.
```

This is nice, since one doesn't want spaces following en- or em-dashes. If you do want them, it's easy enough to redefine `\bEathyphens` and `\cEathyphens` so they're added before `^^M`, or you can type

```
... in lines 299--\space 547 ...
```

```
There was a sudden crash---\space
then silence.
```

However, you may be puzzled as to why they disappear. If `\cEathyphens` just eats the third `-` and ends the group, a following `^^M` will cause a space to be inserted into the current list, so it's the `\futurelet` that causes the space to disappear.²⁶ When `\futurelet` peeks at a token, it fixes its `\catcode` even though it doesn't expand it.²⁷ When the group ends and `-` is finished, `TeX`'s parser reads the `^^M` that follows the second or third `-`. The `\catcode` of this `^^M` was fixed at 13, so it expands to `\empty`, its expansion within `-` rather than a blank space.

Changing the category code of `-` to `\active` has some minor, unpleasant side effects in the `\trans` environment. It is no longer possible to use `-`, `--` and `---` in the arguments to some macros, such as `\beginchapter` and `\beginsection`, which are defined in `conctex.tex`, and code like `\vskip-2pt` will cause an error, because `-` must have `\catcode 12` (other) for this to work. The control words `\hyphen`, `\dash` and `\Dash` can be used in macro

²⁶ In the case of `--`, it's the `\futurelet` in `\aEathyphens` that causes the space to disappear.

²⁷ *The TeXbook*, p. 381: "... `TeX` stamps the category on each character when that character is first read from a file."

arguments instead; other cases should be put into a commentary or the `\plain` environment, where `-` has `\catcode=12`, e.g., `*\vskip-12pt*`.

⚡ If certain macros that use `-`, `--`, or `---` are used frequently in the `\trans` environment, and you don't want to put them in commentaries or the `\plain` environment, you can redefine them using a similar technique to that used in the definitions of `-` and `\invisiblehyphen`.

```
\begingroup
\catcode'\-=12
\gdef\mymacro{\begingroup
\catcode'\-=12
\def\submymacro##1##2##3{%
... \endgroup}}
\endgroup
```

The macro `\putontop` is defined like this. It doesn't work if you try to redefine `\vskip`, though.

Homograph identifiers. *Homographs* are words that are spelled the same, but belong to different lemmata. They must be indicated explicitly in the input file. In order to do this, I have defined the characters “<” and “>” to delimit *homograph-identifiers* in the `\trans` environment. For instance, in Old Icelandic, the word “á” can be a noun, meaning “river” or a preposition, meaning “at, on,” etc. In the input file, I can distinguish between them by typing the preposition as `{\ 'a}<p>` and the noun “á” as plain `{\ 'a}` or `{\ 'a}<n>`. `TeX` handles them in different ways, depending on the value of `\ifdraft`. For rough drafts, it will be useful to see the homograph-identifiers, whereas they should not appear in the final draft.

To accomplish this, the `\catcode` of `<` has been changed to `\active`. The definition of `<` depends on the value of `\ifdraft`:

```
\catcode'\<=\active

\def\eatit#1{\ifx#1>\else
\expandafter\eatit\fi}

\def\donteatit${\{$%
\def\subdonteatit##1{%
\ifx##1>${\}$\else##1%
\expandafter\subdonteatit\fi}%
\subdonteatit}

\ifeathomoids
\let<=\eatit
\else
\let<=\donteatit\fi
```

The macro `\eatit` simply reads an argument, tests to see if it's “>” and if it isn't, calls itself again. The arguments it reads are discarded, hence the name. The macro `\donteatit` functions in a similar way, but has the added complication that the “{” should only be put into the current list once, so the main work of not eating the string is taken over by `\subdonteatit`. When `\draftfalse`, `<` is `\let` to `\eatit`. When `\drafttrue`, the homograph-identifiers shouldn't be eaten, so `<` is `\let` to `\donteatit`, which puts the text between `<` and `>` inside curly braces, e.g., “á{p}”.

⚡ Homograph identifiers can be as long as you want, because `\eatit` and `\donteatit` process the tokens (or groups) within the angle braces one-by-one. This prevents a long argument from burdening `TeX`'s memory. When `\eatit` or `\subdonteatit` calls itself recursively within the conditional construction, the `\expandafter` allows the conditional, and hence the current invocation of `\eatit` or `\subdonteatit`, to complete execution before the recursive call is expanded. Therefore, only one invocation of `\eatit` or `\subdonteatit` is active at any time. This is called “tail recursion” (cf. *The TeXbook*, p. 219). It would also be possible to define `\eatit` and `\donteatit` like this:

```
\def\eatit#1>{\relax}
\def\donteatit#1>{\${\${#1$}\$}
```

In this case, extremely long homograph identifiers will burden `TeX`'s memory, but in practice, it would be just as good, since it's unlikely that anyone would want to type in extremely long homograph identifiers.

The Lisp program `conctex.lsp`

The Lisp program `conctex.lsp` consists of several parts:

1. A lemmatization dictionary.
2. A parser that reads the input files, discards extraneous material, and passes the transcription lines on for further processing.
3. A routine for extracting information from the transcription lines, accessing Lisp symbols, and storing the information in data structures.
4. An output routine that writes the `TeX` file containing the concordance.

`ConcTeX` does not use the page numbering information generated by `TeX`'s output routine, so it doesn't require a preliminary pass. The concordance shows the position of individual words in the *original manuscript*, not the page numbers of the printed transcription. This makes it possible

to generate the concordance directly from the \TeX input file without running \TeX at all.

⦿ Making a concordance using the page numbers generated by \TeX 's page breaking routine is a more difficult matter. If the input files contain explicit commands for page breaking, i.e., $\backslash\text{eject}$, $\backslash\text{supereject}$, and/or macros that call one or the other of these macros, $\text{Conc}\text{\TeX}$ can be used with only trivial modifications. However, it will not work if \TeX is supposed to break pages automatically. One way around this problem would be to let \TeX break the pages automatically, and insert code for conctex.lsp in the final version, at the page breaks \TeX found, so conctex.lsp knows to increment a page counter. This would not be entirely satisfactory, however.

⦿ If explicit page breaks are used, or code is entered in the input file indicating where pages are broken, then explicit line breaks (using $\backslash\text{break}$ and/or $\backslash\text{par}$, or macros using one or the other of these commands) would make it possible to generate a concordance using line numbering information referring to the lines in the output. If \TeX does the line and page breaking automatically, however, and page breaks are not explicitly indicated, then conctex.lsp won't know when to increment the page counter and reset the line counter to 1. *There is no way to generate line numbering information without explicitly breaking the lines.*

⦿ It is therefore not possible to use $\text{Conc}\text{\TeX}$ for making a concordance using line and page numbers that result from \TeX 's line and page breaking routines, i.e., without explicit line and page breaking commands. But it might be possible to design a concordance program that could do this by having it process the dvi file instead of the input file. However, if this sort of concordance were desired, a better approach would be to write a new version of \TeX that incorporates the features of $\text{Conc}\text{\TeX}$ in the WEB program itself, thus making it unnecessary to use an auxiliary program. However, since conctex.lsp uses several features peculiar to Lisp, an entirely different program structure would be necessary.

Lemmatization. The most difficult problem in generating a concordance is lemmatization. There are three cases that need to be accounted for:

1. *Homographs.* Words that are spelled the same, but belong to different lemmata, like the word “spring”, which can be one of several nouns or a verb.

2. *Subsidiary forms.* Distinct words that belong to the same lemma, like “man”, “man's” and “men” in English.
3. *Variants.* Words that are spelled differently, but are really the same, and should appear under the same heading, like “color” and “colour” in English.

A lemma in the context of generating a concordance is a set of one or more distinct words which belong together, like the various forms in a morphological paradigm, e.g., the nominative, genitive, dative, and accusative; singular and plural, of “drotning” (Engl. “queen”).

	<i>Sg.</i>	<i>Pl.</i>
<i>Nom.</i>	drotning	drotningar
<i>Gen.</i>	drotningar	drotninga
<i>Dat.</i>	drotningo	drotningom
<i>Acc.</i>	drotning	drotningar

One of these words is denoted the *main form* of the lemma; with nouns, it's usually the nominative singular, with verbs, the infinitive, etc. The *subsidiary forms* should appear in the concordance below the main form, indented, with their occurrences, arranged according to grammatical criteria.


drotning (2) 1va 12, 3rb 14.
 drotningo (2) 2rb 25, 12va 30–31.
 drotninga (1) 13ra 16.

Homographs must be distinguished in the concordance. Sometimes, they can belong to the same lemma, like the three forms of “drotning” that are spelled “drotningar”: the genitive singular and the nominative and accusative plural. Sometimes they can belong to different lemmata, like the noun “á” and the preposition “á” in Old Icelandic.

Lemmatization must also account for *variants*. Where there are variants, all of the occurrences are assigned to a single form, which appears in the concordance. One way variants can arise is from the use of interchangeable letter forms in a manuscript. The manuscript transcription might use a macro or a font change to indicate the use of an alternative letter. For example, the word “prestr” may sometimes appear in the transcription as “prestr” → “prestr” and sometimes as “prestr{\r}” → “prestr”, depending on the form of “r” used in that passage in the manuscript, so “prestr” may be termed a variant of “prestr” (or vice versa). Only one of them, say “prestr”, should appear in the concordance, and the occurrences of “prestr” should be listed under “prestr”. Subsidiary forms can also have variants.

Lemmatization is the process of sorting individual words, so that each one is put into its proper

lemma. There are several possible solutions to this problem. The one described here is a *lemmatization dictionary*, which must account for all subsidiary forms and variants.

 Running `conctex.lsp` without loading a lemmatization dictionary produces a complete list of all the words in the transcription, with their occurrences, but each entry is considered a main form, and only those variants are accounted for which `replace-items` or `process-line` catch automatically. For a concordance this is clearly inadequate, but it might be useful for some other purpose.

If concordances are to be generated for normalized transcriptions, or for a group of manuscripts whose orthographic conventions are very similar, a master lemmatization dictionary can be used. In most cases, however, the orthography and the form of the language of a given manuscript will diverge enough from that of others that it will be necessary to create a special lemmatization dictionary for that particular manuscript.

One of the first things that `conctex.lsp` does is to load the lemmatization dictionary. This is one or more files of Lisp code with invocations of the functions `generate-entry`, `harmless`, and `add-variants`. These functions cause data to be stored in `word structures`.

The data for the entries in the concordance are stored in `structures` of type `word`, defined like this:

```
(defstruct word
  main-form
  occurrences-mariu-a
  occurrences-mariu-s
  sort-string
  tex-string
  forms
  root
  variants
)
```

More slots can be added for other applications; in particular, `occurrences` slots can be added for additional manuscripts.

The function `generate-entry`. The basic function for generating a lemmatization dictionary is `generate-entry`. It's invoked like this:

```
(generate-entry "David")
```

or like this:

```
(generate-entry "David" noun)
```

The string, which is the first, required argument to `generate-entry`, is used, perhaps with some modifications, as the name of a symbol which is bound to a `word-structure`. The original string is stored

in the `main-form` slot of the `word structure`. In this example, the string "David" is surrounded by `||`, and the symbol is accessed with `read-from-string`.²⁸

```
(set (read-from-string "||David|")
     (make-word :main-form "David"))
```

Surrounding a string with `||` has the effect of escaping all the characters within the string. This makes it possible to have symbol names with lowercase letters and other characters, which are normally not allowed in symbol names in Lisp. Evaluating

```
(read-from-string "David")
```

without `||` returns a symbol named `DAVID`, because Lisp converts lowercase to uppercase letters in symbol names unless they are escaped using `\` or `||`.

The second, optional argument to `generate-entry`, `noun` in the example above, is discarded by `generate-entry`. It is allowed for compatibility with another program that uses the same dictionary files for a different purpose.²⁹

Alphabetization. The first argument to `generate-entry` always refers to the main form of a lemma. It is treated as a heading in the concordance. Lemmata are sorted in the concordance in alphabetical order of the main forms. `ConcTeX` includes a special sorting routine.³⁰ It makes it possible to sort arbitrary special characters in a user-defined order by replacing ordinary characters and special character macros with 0 or more characters from Lisp's code table, which is based on the ASCII code table. In its current form, it allows up to 256 positions; however the actual number of special characters that can be sorted is much greater. This is because some characters occupy no position at all, others share a position with another character, and still oth-

²⁸ What `generate-entry` really does is a bit more complicated, but the effect is the same.

²⁹ This other program is called `LexTeX` and I plan to document it in a subsequent article. It's for generating dictionaries, vocabulary flashcards and similar things from files of Lisp code.

³⁰ The alphabetization routine for `ConcTeX` is essentially identical to the one in my `Spindex` package. For a more complete discussion of the alphabetization routine, see my previous article "Spindex—Indexing with Special Characters", *TUGboat* 18(4)/1998, pp. 255–273. Since `ConcTeX`, unlike `Spindex`, does not require a preliminary `TeX` run in order to generate page numbering information by means of `TeX`'s output routine, and therefore uses no `\write` commands, special character macros in the input file can be coded in the normal way (but with obligatory braces).

ers use the positions of more than one other character.

The string “David” is passed to the function `generate-info`, which passes each letter or special character coding to `letter-function` in order to generate a `sort-string`. Each letter or special character in the string is used to create a new string using characters from Lisp’s code table. The `sort-string` generated for “David” might be `"^D^A^[^K^D"`, depending on the way the alphabetization routine is set up.³¹ The `sort-string` is put into the car of a cons cell, with the symbol bound to the word structure in the cdr.

```
("^D^A^[^K^D" . |David|)
```

This cons cell is then put into an association list (or alist) called `var-alist`. This alist will contain a cons cell for each of the word structures created by `generate-entry` in the lemmatization dictionary. If this is the lemmatization dictionary:

```
(generate-entry "David")
(generate-entry "eiga")
(generate-entry "eptir")
```

`var-alist` will look like this:

```
((("^D^A^[^K^D" . |David|)
 ("^F^K^H^A" . |eiga|)
 ("^F^U^Y^K^W" . |eptir|))
```

⚠ Formatting commands like `\it`, `\bf`, `\sc`, etc. are ignored when the `sort-string` is generated, otherwise, they would cause an error in `letter-function`.

Words in the lemmatization dictionary can contain special characters.

```
(generate-entry "dyr{\dh}ligr")
```

Note that two backslashes are necessary for the `{\dh}` in `generate-entry`’s argument. The resulting symbol that is bound to the word structure is `|dyr{dh}ligr|` (Engl. “glorious”), and the backslashes disappear in the symbol name.

⚠ The braces surrounding special character macros are needed so `generate-info` knows where they begin and end. `TeX`’s parser is cleverer than `generate-info`. When `TeX` finds an escape character, it reads the following tokens and uses them to make the longest control sequence possible. The function `generate-info`, on the other hand, needs delimiters

³¹ Each project will have its own requirements with respect to alphabetization, so this must be customized by the user. The documentation supplied with `ConcTeX` explains in detail how to do this. The string `"^D^A^[^K^D"` consists of non-printing characters in Lisp’s printed representation.

for control sequences within words so it can pass them as a whole to `letter-function`.

Subsidiary forms. The function `generate-entry` can also have keyword arguments.

```
(generate-entry "dyr{\dh}ligr"
 :forms "dyr{\dh}lig")
```

The `:forms` argument tells `generate-entry` that “`dyrðlig`” is a subsidiary form of “`dyrðligr`” (in fact, it’s the feminine singular nominative and neuter plural nominative and accusative form). The string `"dyr{\dh}lig"` is used to access a symbol, `|dyr{dh}lig|`, which is bound to another word structure. The symbol `|dyr{dh}lig|` is added to the list in the forms slot of the word structure `|dyr{dh}ligr|`,³² and the symbol `|dyr{dh}ligr|` is put into the root slot of the word structure `|dyr{dh}lig|`. The function `generate-info` is used to generate a `sort-string` for `|dyr{dh}lig|`, and the cons cells

```
("^D^^^W^E^P^K^H^W" . |dyr{dh}ligr|)
```

and

```
("^D^^^W^E^P^K^H" . |dyr{dh}lig|)
```

are added to `var-alist`. All of the word structures are added to `var-alist`, not just the ones for main forms of lemmata.

The `:forms` argument to `generate-entry` needn’t be a single string. It can also be a list of strings.

```
(generate-entry "dyr{\dh}ligr"
 :forms '("dyr{\dh}ligs" "dyr{\dh}ligum"
 "dyr{\dh}ligan"))
```

In this case, the subsidiary forms listed are the masculine singular genitive, dative, and accusative strong forms “`dyrðligs`”, “`dyrðligum`”, and “`dyrðligan`”. Each of the forms is used to access a symbol and bound to a new word structure. The symbol `|dyr{dh}ligr|` is put in the root slot of each of these word structures, and the symbols for the subsidiary forms are put into the list in the forms slot of `|dyr{dh}ligr|`.

```
(|dyr{dh}ligs| |dyr{dh}ligum| |dyr{dh}ligan|)
```

The forms are stored in the order in which they appear in the invocation of `generate-entry`. They will appear in this order in the concordance. They are not sorted alphabetically, because subsidiary forms should be arranged in the concordance according to grammatical criteria.

Variants. Some words in a manuscript may differ from the standard form that should appear in the concordance. For example, the word for “king”

³² In Lisp, `nil` is the same as the empty list `()`, so it’s possible to add an element to the list `nil`.

in Old Icelandic is “konungr”, but sometimes the form “kongr” is used (cf. Modern Danish “konge”). If a manuscript uses both forms, “konungr” and “kongr”, but only “konungr” should appear in the concordance, `generate-entry` can be called using the `:variants` keyword argument.

```
(generate-entry "konungr" :variants "kongr")
```

The string “kongr” is used to access a symbol, `|kongr|`, but this symbol is not bound to a word structure; instead, it’s bound to the symbol `|konungr|`, i.e.,

```
(setq |kongr| ' |konungr|)
(symbol-value ' |kongr|) → |konungr|
```

(Again, what `generate-entry` really does is more complicated, but this is the effect.) The occurrences of the word “kongr” in the transcription, if any, will appear under “konungr” in the concordance.

The `:variants` keyword argument to `generate-entry` may also be a list of strings.

```
(generate-entry "konungr"
 :variants '("kongr" "ko{\n}gr"
 "kvnvngr" "konongr"))
```


In this case, all of the strings in the list are used to access symbols which are bound to `|konungr|`.

The symbols that are made from the string or strings in the `:forms` keyword argument to `generate-entry` can also have variants.

```
(generate-entry "eiga" :variants "ei{\g}a"
 :forms '("{\ 'a}"
 ("attv" :variants '("a{\t}v" "atto"))))
```

Here, there are two forms, “á” and “attv”, and the latter has two variants, “aTv” and “atto”. The `:forms` argument can be a list comprising a combination of simple strings and/or lists such as `("attv" :variants '("a{\t}v" "atto"))`.

```
(generate-entry "eiga" :variants "ei{\g}a"
 :forms '("eigir"
 ("eigi" :variants "ei{\g}i")
 "eigom"
 ("attv" :variants
 '("a{\t}v" "atto"))))
```

 The function `generate-entry` takes lists like `("eigi" :variants "ei{\g}i")` in its `:forms` ar-

gument and conses³³ (or puts) the symbol `sub-generate-entry` onto the front so it looks like this:

```
(sub-generate-entry "eigi"
 :variants "ei{\g}i")
```


Then it appends the list

```
(:root "eiga" :recursive t)
```

to the end of it, resulting in

```
(sub-generate-entry "eigi"
 :variants "ei{\g}i"
 :root "eiga" :recursive t)
```

and evaluates this list.³⁴

 The lists in `generate-entry`’s `:forms` argument can also have `:forms` arguments.

```
(generate-entry "abc"
 :forms '("def" :forms "ghi"))
```

This is meaningless in the context of a concordance, since nesting in grammatical paradigms only has two levels. It might be useful, if variants were to appear in the concordance. Then they could be indented to the value of `\parindent`, and their forms could be indented to the value of `\parindent\parindent`. It might also be useful for some purpose other than a concordance. Changes to `process-line` and `export-entry` would be necessary to make deeper nesting work.

Homograph identifiers can be assigned to a word structure in two different ways: by using the `<string>` syntax, as in the input file,

```
(generate-entry "afl<nsg>")
```

or by using the `:homograph-identifier` keyword argument.

```
(generate-entry "afl"
 :homograph-identifier "nsg")
```

The string “nsg” stands for “neuter, singular”. In the concordance, this word will appear as “afl [nsg]”. If you want “afl [n. sg.]” or “afl [neut. sing.]” to appear in the concordance instead, you can write

```
(generate-entry "afl<n.~sg.>")
```

or

³³ In Lisp, a list is a chain of cons cells. Cons cells contain two elements, the `car` and the `cdr`, in that order. In the list `(a b c d)`, the first cons cell has the symbol `a` in its `car` and a pointer to the following cons cell in its `cdr`. This in turn has the symbol `b` in its `car` and a pointer to the next cons cell in its `cdr`. The last cons cell in the list has `d` in its `car` and `nil` in its `cdr`: `(d . nil)`. So the list `(a b c d)` is really `(a . (b . (c . (d . nil))))`.

³⁴ Since the original invocation of `generate-entry` called `sub-generate-entry`, as explained below, the latter is called recursively for subsidiary forms.

```
(generate-entry "afl"
  :homograph-identifier "n.~sg.")
```

or

```
(generate-entry "afl<neut.~sing.>")
```


or

```
(generate-entry "afl"
  :homograph-identifier "neut.~sing.")
```

instead. But “afl<nsg>” is more convenient to type in the input file than “afl<neut.~sing.>”, so it’s possible to use a cons cell as the :homograph-identifier argument to generate-entry.


```
(generate-entry "afl"
  :homograph-identifier
  '("nsg" . "neut.~sing."))
```

This way, you can type “afl<nsg>” in the input file, but “afl [*neut. sing.*]” will appear in the concordance.

 If you use both the <string> syntax and an explicit :homograph-identifier argument,

```
(generate-entry "afl<n.~sg.>"
  :homograph-identifier
  '("nsg" . "neut.~sing."))
```

the explicit :homograph-identifier argument takes precedence, and the homograph-identifier using the <string> syntax is discarded. It doesn’t matter whether it’s a simple string or a cons cell in the :homograph-identifier argument.

 When you use a homograph-identifier, no matter how you create it, the name of the symbol bound to the word structure includes the homograph-identifier. For instance, in the last example, the word structure is bound to the symbol |afl<nsg>|, its main-form is "afl", its tex-string is "afl [{"neut.~sing.\\\/}]", and its homograph-identifier (i.e., the object stored in its homograph-identifier slot) is "nsg".

When the main form of a lemma is a homograph and has subsidiary forms, the latter may or may not also need homograph-identifiers. In Old Icelandic, there are two words spelled “afl”, a masculine noun meaning “an ironsmith’s forge” and a neuter noun meaning “physical strength”. If generate-entry is called like this:

```
(generate-entry "afl<nsg>"
  :forms '("afls" "afl"))
```

the genitive and dative singular forms, “afls” and “afl”, may appear in the input file with no homograph-identifier, and they will appear in the concordance as subsidiary forms of the neuter noun

“afl”. This is not very satisfactory, however, because the genitive and dative singular forms of the masculine “afl” are identical to the genitive and dative singular of the neuter “afl”. So it’s necessary to type in

```
(generate-entry "afl<nsg>"
  :forms '("afls<nsg>" "afl<nsg>"))
```

or

```
(generate-entry "afl"
  :homograph-identifier "nsg"
  :forms '("afls<nsg>" "afl<nsg>"))
```

or

```
(generate-entry "afl"
  :homograph-identifier "nsg"
  :forms '("afls"
    :homograph-identifier "nsg")
    "afl<nsg>"))
```

or one of the other possible alternatives. Note that the various possibilities of formulating the arguments are always allowed.

If no homograph-identifier is indicated for a subsidiary form of a lemma whose main form has one, the subsidiary form may be typed in the input file with no homograph-identifier. However, generate-entry automatically creates a variant using the homograph-identifier of the main form.

```
(generate-entry "afl<m>" :forms "aflar")
```

This results in a symbol |aflar| which is bound to a word structure, and a symbol |aflar<m>| which is bound to the symbol |aflar|. The word “aflar”, masculine plural nominative, is unambiguous and needs no homograph-identifier, since no other form of “afl” (masculine) or “afl” (neuter) is spelled the same way.

Another problem is that words in the same lemma sometimes have identical forms. The nominative and accusative, singular and plural of the neuter noun “afl” are all spelled “afl”. If it’s desirable to keep the morphological forms of lemmata separate in the concordance, it’s necessary to have different homograph-identifiers for the forms that are spelled the same.

```
(generate-entry "afl<nsg>"
  :forms '("afls<nsg>" "afl<nsg>"
    "afl<nsga>" "afl<npln>"
    "afl<npla>"))
```

Most of the time, the homograph-identifiers of subsidiary forms are not printed to the concordance. Since the homograph-identifier of the main form is printed to the concordance, indicating for instance that “afl” is a neuter noun, it’s not necessary to

indicate this fact for the subsidiary forms that appear below it. Sometimes, though, it's desirable to print something. The homograph-identifiers for subsidiary forms are printed to the concordance only when an explicit `:homograph-identifier` argument is used, not when the `<string>` syntax is used. The homograph-identifiers for main forms of lemmata are always printed to the concordance.

```
(generate-entry "afl<neut.>"
  :forms '("afls<nsg>" "afl<nsg>"
    '("afl"
      :homograph-identifier
      '("nsga" . "sg.~acc."))
    '("afl"
      :homograph-identifier
      '("npln" . "pl.~nom."))
    '("afl"
      :homograph-identifier
      '("npla" . "pl.~acc."))))
```

⇒

```
afl [neut.]
afls
afl
afl [sg. acc.]
afl [pl. nom.]
afl [pl. acc.]
```

Note that not all the information that's necessary for the homograph-identifier in the input file must be printed. The string `"afl<npln>"` in the input file indicates this form of "afl" unambiguously, i.e., neuter, plural, nominative. But only "pl. nom." need be printed to the concordance, because it's obvious it's the neuter noun.

The **tex-string** is what is written to the concordance. It is usually determined by a combination of `generate-entry`'s first argument and the homograph-identifier, if any. It is possible to override this by using the `:tex-string` keyword argument to `generate-entry`.

```
(generate-entry "abc" :tex-string "xyz")
```

This causes a symbol `|abc|` to be created and bound to a word structure, but "xyz" will be printed to the concordance. Occurrences of "abc" in the input file will appear under the heading "xyz", and the sort-string will be `^X^Y^Z` (depending on the values assigned by `letter-function`), i.e., based on the string "xyz" and not on "abc".

A `tex-string` can also be set by using a cons cell for the first argument to `generate-entry`.

```
(generate-entry '("eiga" . "agie"))
```

is equivalent to

```
(generate-entry "eiga" :tex-string "agie")
```

However, if you type

```
(generate-entry '("eiga" . "igea")
  :tex-string "agie")
```

the explicit `tex-string` keyword argument takes precedence over the `tex-string` in the `cdr` of the cons cell, which is discarded. I'm not sure whether the `tex-string` feature will prove to be useful, but it's available if needed.

◆ The function `generate-entry` allows the use of other keyword arguments. It doesn't matter what they are, `generate-entry` simply ignores them. This is for compatibility with the program `LexTeX`, which uses the lemmatization dictionary files for another purpose (see page 389), or for other programs that might be written. For instance,

```
(generate-entry "nema" verb
  :forms '("nam" "n{\ohook}mum"
    '("nvmenn" :variants "nomenn"))
  :class 'strong-4 :definition-english "take")
```

contains the keyword arguments `:class` and `:definition-english`, and their values. They are irrelevant to the function `generate-entry` in `ConcTeX`, which ignores them, but are used by the function called `generate-entry` in `LexTeX`.

◆ `ConcTeX` could be extended to do more than just produce a concordance. One possibility is to sort the words into a file of `TeX` code according to grammatical criteria. For instance, the words could be written to an output file nouns first, then adjectives, pronouns, verbs, adverbs, etc. Within these categories, the words could be sorted according to class. In order to accomplish this, more information must be entered in the lemmatization dictionary, i.e., `generate-entry` must be modified to process additional arguments. Alternatively, the homograph-identifiers could be used for this purpose.

◆ Actually, `generate-entry` isn't really a function at all. It's a macro, defined with `defmacro`. One reason for this is that a macro doesn't evaluate the arguments that are passed to it. If `generate-entry` were defined like this,

```
(defun generate-entry
  (main-form &optional word-type forms
    ...) ...)
```

then this would cause an error:

```
(generate-entry "{\ae}tt" noun
  :forms "{\ae}ttar")
```

because Lisp would try to evaluate the symbol `noun`, which is unbound. Since `generate-entry` is a macro,

```
(defmacro generate-entry (&rest arg) ...)
```

it's possible to manipulate its arguments before they're passed to the *real* function, which is called *sub-generate-entry*. Another reason for defining *generate-entry* as a macro is to make it easier to extend *ConcTeX*, or change the way it behaves. The arguments can be manipulated and passed to other functions, instead of or in addition to *sub-generate-entry*.

The functions harmless and add-variants are two other functions that can be used in the lemmatization dictionary. These are merely convenience functions, you can accomplish the same results using *generate-entry*.

The function *harmless* is for cases where none of *generate-entry*'s keyword arguments are needed. Some words, like prepositions and conjunctions (in the Germanic languages, at least) have no inflected forms, and some words may have no variants in a particular manuscript. (So far, I've never needed a *tex-string*.)

```
(harmless "{\\ 'a}<p>" "af"
          "{\\ '\\i}" "ok" "yfir")
```

is equivalent to

```
(generate-entry "{\\ 'a}<p>"
(generate-entry "af")
(generate-entry "{\\ '\\i}")
(generate-entry "ok")
(generate-entry "yfir"))
```

The function *harmless* takes one or more strings as its arguments, and calls (*generate-entry* *<string>*) for each of the strings. Note that a homograph-identifier is permitted, using the *<<string>>* syntax, as for *{\\ 'a}<p>*.

Calling *generate-entry* with a lot of forms and variants can be confusing,

```
(generate-entry "dyr{\\dh}ligr" adjective
  :forms '(("dyr{\\dh}lig"
  :variants "dyrdlig")
  ("dyr{\\dh}ligi" :variants "dyrdligi")
  ))
```

so the function *add-variants* can be used instead. This is equivalent to the previous example:

```
(generate-entry "dyr{\\dh}ligr" adjective
  :forms '("dyr{\\dh}lig" "dyr{\\dh}ligi"))

(add-variants "dyr{\\dh}lig" "dyrdlig")
(add-variants "dyr{\\dh}ligi" "dyrdligi")
```

All of the arguments to *add-variants* are strings. The first is used to access a symbol. This symbol might be bound to a *word structure*, which would be

fine. If, however, it's unbound, then a *word structure* is created, using the string as the only argument to *generate-entry*. Otherwise, if it's bound and not a *word structure*, *add-variants* issues a warning, and exits. Assuming the symbol is now a *word structure*, the other strings are used to access symbols which are set to the symbol derived from the first string, just as *generate-entry* sets variants (as described on page 390f).³⁵

The order of the subsidiary forms is important. It will usually not be desirable to have them sorted in alphabetical order in the concordance, although it's possible; usually, the order of the subsidiary forms will be determined by the conventional order of the forms within the paradigm, e.g., singular nominative, genitive, dative, accusative, then plural nominative, genitive, dative, accusative for nouns in Old Icelandic. If you prefer the order nominative, accusative, genitive, dative, just type the forms in this order.

The parser. After the lemmatization dictionary has been loaded, *conctex.lsp* opens the input file and the function *main* reads it line-by-line. Entirely blank lines, and lines that begin with % or \ are discarded (except for lines beginning with *\lineno*, *\putontop*, *\overstroke*, or other exceptions, if any). Lines that begin with @ are Lisp code, the @ is discarded, and the rest of the line, which must be a balanced expression (*sexp*), is evaluated. Other lines are text lines, and passed to the function *process-line*. This function strips off leading and trailing blanks, discards line numbers and punctuation from the beginning of the line, discards certain macros and perhaps some or all of their arguments, discards unattached math mode material, and replaces some strings with others. If a @ or % appears in the middle of a line, *process-line* discards it together with the rest of the line. It also tests to see whether the end of the input line corresponds to the end of a transcript line. If the input line ends with \\, -, or \-, or if the following line is blank, *process-line* treats the end of the current line as the end of a transcript line and increments the line counter (called *line-counter*) when it's done with the current line. (Advancing the leaf, side or column requires an explicit call to *set-position*, as does advancing more than one line at a time, which is sometimes necessary, as when part of a leaf is missing or unreadable.) Then *process-line* passes the line to the function *read-word*. This function strips characters off the line until it has stripped off a whole word. Then it returns the word, *current-*

³⁵ Actually, *generate-entry* uses *add-variants* to set variants.

word, and the rest of the line to `process-line`. Now the routine for access and data storage takes over.

Access and data storage. The function `read-word` returns `current-word` as a string. The function `process-line` appends `|` to the beginning and end of the string, so that, for example, "`{\ae}tt`" becomes "`|{\ae}tt|`".³⁶ Now the string is used to access a symbol (or variable) using `read-from-string`, and the symbol `current-var` is set to this symbol, e.g., `|{\ae}tt|`. (In C, you'd say "current-var is a pointer to `|{\ae}tt|`".)

This is one reason why unmotivated braces are not permitted in the input file. The braces are needed to delimit the special characters for `generate-info` and to distinguish between special character codings and ordinary characters in the symbol names.³⁷ The vertical strokes surrounding the symbol name act to escape all of the other characters, which causes `\` to disappear when the string "`|{\ae}tt|`" is converted to the symbol `|{\ae}tt|`. If the input file contained the string "`{\ae}tt`" \rightarrow "aett" and "`{\ae}tt`" \rightarrow "aett" (using "ae" instead of the ligature), and unmotivated braces were permitted, they would both map to the same symbol, namely `{\ae}tt`, and they would not be kept separate in the concordance.

Now `process-line` checks to see if the *value* of `current-var` (the symbol `|{\ae}tt|`, in our example) is already bound. If it is, this means either that the word "aett" is accounted for in the lemmatization dictionary, or that it's occurred previously in the input file. If `|{\ae}tt|` is unbound, `process-line` checks whether the symbol `|{\Ae}tt|` is bound. If it is, `current-var` is set to `|{\Ae}tt|`, otherwise `process-line` checks whether the symbol `{\AE}TT` is bound.³⁸ If it is, `current-var` is set to `{\AE}TT`.³⁹

³⁶ Backslashes in strings read in from a file are automatically escaped.

³⁷ By uncommenting two lines in `letter-function`, it's possible to allow unmotivated braces. I believe it's safer to make them signal an error, because it's a good way of discovering mistakes in the input file. Any cases where braces are required can be accounted for in `conctex.lsp`.

³⁸ No vertical strokes are needed around `{\AE}TT` because `{`, `}`, and capital letters don't need to be escaped.

³⁹ Manuscripts are often orthographically inconsistent, and transcriptions often use uppercase and lowercase letters to represent different letter forms, although the original letters do not really correspond to our upper- and lowercase. If the editor wants a particular word to be represented in the concordance in lowercase, capitalized or all capitals, and the first occurrence of this word in the transcription does not have the desired form, then the word must be entered in this form into the lemmatization dictionary. Of course, capitalized, upper- and lowercase strings which map

Actually, the functions `string-downcase`, `string-capitalize` and `string-upcase` are applied to (`symbol-name current-var`). The symbol `current-var` is evaluated once to get the symbol `|aett|`. If I wanted to examine the `symbol-name` of `|aett|` directly, I would have to type (`symbol-name ' |{\ae}tt|`) to prevent evaluation of `|{\ae}tt|`. Note that applying `string-capitalize` to a symbol name beginning with a coding like `{\ae}` fails, because the capitalized string would need to be `{\AE}`. It is, of course, possible to have `\Ae` expand to "Æ" in \TeX , so that `{\Ae}` would yield the correct result, but it seems hardly worthwhile, since words should be accounted for in the lemmatization dictionary anyway.

Unbound symbols. If `|{\ae}tt|`, `|{\Ae}tt|`, and `{\AE}TT` are unbound, a word structure is created, and `|{\ae}tt|` is bound to it. A `sort-string` is generated and stored in `|{\ae}tt|`'s `sort-string` slot, and a `tex-string` is stored in its `tex-string` slot, just as for word structures created by the lemmatization dictionary. If the word read in from the file includes the symbols `<` and `>`, the string they enclose is considered to be the homograph-identifier, which is stored in the `homograph-identifier` slot, and affects the form of the `tex-string`.

After a word structure has been created for the symbol `|{\ae}tt|`, the symbol itself is put into the `cdr` of a cons cell, while its `sort-string` is put into the `car`, e.g.,

```
("^A^F^X^X" . |{\ae}tt|)
```

This cons cell is now appended to the association list (or alist) `word-alist`, *not* `var-alist`, as are the word structures from the lemmatization dictionary. The alist `word-alist` is used to store all of the entries that are main forms of lemmata, i.e., not subsidiary forms or variants, that actually occur in the transcription. They will appear as top-level entries in the concordance. Whenever a word in the input file maps to a symbol that is unbound, i.e., if it's the first occurrence in the input file and not in the lemmatization dictionary, this word is considered to be the main form of a lemma, so all lemmata with subsidiary forms and most variants must be accounted for in the lemmatization dictionary. Variants that differ only in the use of capitalization and upper- and lowercase letters, or where certain special character macros are replaced, are handled automatically (as above).

Positions and occurrences The word structure has an `occurrences` slot, where manuscript positions are stored. It might even have multiple

to different word structures are also possible, if they're bound in the lemmatization dictionary.

occurrences slots, if ConcTeX is being used for more than one manuscript. When a word structure is created in the lemmatization dictionary, there are no occurrences to be stored yet. When an entry is created for a word in the input file, the `position-string` is stored as a string in a list in the proper occurrences slot, e.g., ("28va~7"), using `access-occurrences`.

The current position in the transcription (and the manuscript) will have been generated by the function `set-position`, which is used in the input file, and `process-line`'s line counting routine. In addition, `process-line` can tell whether a particular word is the first or last word in the transcription line. It can sometimes be useful to indicate that a word is at the beginning or end of a line in a manuscript, because the use of unusual forms is often explainable because of the position. For instance, words at the beginning of lines may have fancy initials, and words at the end may be abbreviated in order that they may be squeezed into the remaining space on the line. If a word occurs at the beginning or end of a line, this may be indicated in the concordance, if the book designer wishes. For example, if this is a line at leaf 1, verso, column a:

```
\lineno{15} seg_{ir} e{\n} gaufgi
ken_{n}i~{\m}_{a{\dh}r} [ok enn]\
```

⇒

```
15: segir eN gaufgi kenni Maðr [ok enn]
```

then one of the positions for "segir" will be "1va 15^b" and one of the positions for "enn" will be "1va 15^e".

The program `conctex.lsp` doesn't print `^b$` and `^e$` explicitly to the concordance file. Instead, it uses the token lists `\linebeginstring` and `\lineendstring`,

```
\newtoks\linebeginstring
\newtoks\lineendstring
\linebeginstring={^b$}
\lineendstring={^e$}
```

and prints `\the\linebeginstring` and `\the\lineendstring` to the concordance file. To suppress "b" and "e" in the concordance, all that's necessary is to redefine the token lists `\linebeginstring` and `\lineendstring`.

```
\linebeginstring={}
\lineendstring={}
```

This feature will probably be turned off, because it is not customary for concordances to indicate the position of words in a line.

A concordance should only be generated for one manuscript at a time. This is merely the way I've set things up; it's not hard-wired into the program. The definition of the Lisp macro `access-occurrences`

depends on the value of the variable `manuscript`. It is used to access the appropriate occurrences slot for `setf`.⁴⁰

Multiple occurrence slots are useful if you want to use the information generated by `conctex.lsp` in another program. The word structures can be written to a file using Lisp's read syntax. Another Lisp program can load this file, provided structures of type `word` are defined, and the data produced by the last run of `conctex.lsp` will be available. This could be useful for dictionaries or linguistic studies involving multiple sources.

Bound symbols. If the symbol pointed to by `current-var` (`|{ae}tt|` in our example) is bound, this means it was either bound in the lemmatization dictionary, or that the word "ætt" has already occurred in the input file, or both.⁴¹ If it has already occurred in the input file, and it's a main form, `word-alist` already contains the cons cell ("^AF^XX" . `|{ae}tt|`), but if it's only bound because it appears in the lemmatization dictionary, or it's a subsidiary form or a variant, this cons cell will not be in `word-alist`. So, `process-line` checks to see whether a cons cell containing `|{ae}tt|` is in `word-alist`, using:

```
(rassoc current-var word-alist)
```

which searches for a cons cell in `word-alist` using the `cdr` as the search key. If it's there, it means that "ætt" has already occurred in the input file.

The situation is more complicated if the symbol is not already in `word-alist`. There are three possibilities: it could be the main form of a lemma, a subsidiary form, or a variant.

‡ If the symbol is a word-structure, it is either a main form or a subsidiary form.

‡ If the root slot of the word-structure is nil, then it's a main form. In this case, the cons cell containing the symbol and its sort-string is copied from `var-alist` into `word-alist`.

‡ Else, if the root slot of the word structure is non-nil, the symbol is a subsidiary form. In this case, its root slot is accessed in order to get its main form, which is another symbol bound to a word-structure. If the cons cell containing

⁴⁰ It's worth looking at the way `access-occurrences` is defined, because it illustrates a limitation of Lisp's `setf` access function.

⁴¹ Or there's been a terrible mistake. It is unlikely that a symbol derived from a word in the transcription would conflict with a symbol that's already defined by `conctex.lsp` or the Lisp interpreter itself. However, if this problem arises, it would be easy enough to write safety routines to catch the problem symbols.

this symbol and its sort string is not already in `word-alist`, it's copied to it from `var-alist`.

- ‡ If the symbol evaluates to another symbol, it's a variant. In this case, the symbol is replaced by this other symbol, and this process is repeated for the latter.

The Lisp macro `access-occurrences` then appends the position-string for the new position, e.g., "28ra~15" to the appropriate occurrences slot in the appropriate `word-structure`.

Exporting the concordance. After every line in the input file has been processed, the loop in `main` returns nil. If you are generating a concordance using multiple input files, the first input file is closed, the next one is opened and processed in the same way. When all of the input files have been processed and closed, the alists `word-alist` and `var-alist` are sorted.

```
(setq word-alist
      (sort word-alist #'string< :key #'car))
(setq var-alist
      (sort var-alist #'string< :key #'car))
```

This puts the cons cells in `word-alist` and `var-alist` in alphabetical order according to their `cars`, i.e., their sort-strings. Now the sort-strings are now longer needed, so new lists are generated by popping off the cons cells and putting the `cdrs`, i.e., the symbols, into new lists, `word-list` and `var-list`. Only main forms of lemmata are members of `word-list`, and only of those lemmata, in which at least one form (which can be a variant) occurs in the transcription. Other main forms may be bound from the lemmatization dictionary, but they will not be in `word-list`. The list `word-list` is now passed to the function `export-words`, which writes the `TEX` file for the concordance. Exactly what `export-words` writes is a matter for the editor and the book designer, so this part of `ConcTEX` can be changed easily.

The function `export-words` pops the symbols off of the front of `word-list` one-by-one. The `tex-string` is written to the concordance file (here called `concordance.tex`, but any name within reason can be chosen), not indented. If there are no occurrences of the main form, the `tex-string` is enclosed in parentheses. If there are occurrences, the number of occurrences is printed after the `tex-string`, enclosed in parentheses, followed by the occurrences. Subsidiary forms are printed to the concordance if and only if there are occurrences for them.

If there are no occurrences and no subsidiary forms with occurrences, something is terribly wrong and `conctex.lsp` will signal an error. The way

`conctex.lsp` is set up, occurrences at the beginning and end of a line are indicated with a superscript, e.g., "27ra 31^b" and "34vb 12^e". On the other hand, if a word occurs multiple times within a line, not at the beginning or end and not broken, the position string is only printed out once, and the number of occurrences is indicated within parentheses, e.g., "51ra 5 (2)". If the word is broken, what is printed depends on whether it's broken across a line, column, side or leaf. In the extremely unlikely event that the word occurs at the beginning or end of a line, or broken, *and* multiple times within the line, the different cases are listed separately, e.g., "11ra 5^b, 11ra 5 (2), 11ra 5-6".

When a symbol is popped off of `word-list`, `export-words` checks whether forms are present in the `word-structure`'s `forms` slot. The forms should be in the order in which they should be listed in the concordance. The function `export-words` accesses the symbol for each form, and checks the appropriate occurrences slot. If it's non-nil, the `tex-string` for that form is written to `concordance.tex`, with its occurrences, as above, but indented. If a form has no occurrences, nothing is printed to `concordance.tex`. The Lisp program currently does not print out variants in any way. This would require some additional programming, but it is possible.

◆ The program `conctex.lsp` can be extended to extract grammatical information from manuscript transcriptions. One way of doing this would be to use standardized homograph-identifiers. Another would be to add one or more slots to the `word-structure`. Then, `export-words` could be modified to write files for the various grammatical categories, which could then be concatenated. After `word-list` has been generated, `word-alist` and `var-alist` are no longer needed, and `var-list` is never needed by `conctex.lsp`. They are kept or generated only for debugging purposes.

Other data. When `conctex.lsp` is run, it generates some data in addition to the concordance. It counts the number of lines and words in each input file, the total number of lines, the total number of words, the total number of distinct words and the total number of lemmata. It prints this information to the `TEX` file `count_info.tex`.

◆ The number of lines in an input file is simply the value of `line-counter` at the end of the file. The number of words in an input file is the number of times `read-word` returns a non-empty string while that file is being read. The total number of distinct words is the number of times `export-occurrences`

is called and the total number of lemmata is the length of `word-list`.

Hints on using ConcTeX. Trying to generate a concordance from an entire manuscript transcription all at once is a recipe for disaster. I recommend testing the program with one column or side at first, gradually increasing the amount of text. In this way, the user can discover which lemmata are needed, which subsidiary forms need to be assigned to a lemma, which variants need to be accounted for, and so on.

The program `conctex.lsp` writes an additional TeX file containing only the words that are *not* included in the lemmatization dictionary, and their occurrences. This makes it easy to see which words should be lemmatized or declared harmless, so the lemmatization dictionary can be built up gradually, as longer and longer portions of the transcription are read by `conctex.lsp`.

A format for a manuscript transcription is likely to be rather complex and the user will probably discover places where new special characters, fonts and macros are needed. Every new control sequence defined in `conctex.tex` and used in transcription lines will require some alteration to the Lisp program, either in `process-line`, `read-word` and/or `letter-function`. I've intentionally programmed `conctex.lsp` in such a way as to allow changes to reflect different book designs, so it may be necessary to change `export-words`, too. The best way to proceed is to take small parts and run TeX and `conctex.lsp` on the input file to make sure that everything is working properly. When you've got it running smoothly, and you've built up a complete lemmatization dictionary, then you can try running TeX and `conctex.lsp` on the whole manuscript.

Running TeX on the concordance. Since ConcTeX generates the concordance from the TeX input file without running TeX, it's no problem to include `concordance.tex` (the TeX file output by `conctex.lsp` which contains the concordance) in the input file itself:

```
\input concordance
```

But it's safer to include it like this:

```
\newread\concordance
\openin\concordance=concordance
\ifeof\concordance
\message{concordance.tex doesn't exist.
  Not inputting it.}\else
\closein\concordance
\input concordance
\fi
```

If you use UNIX, you can ensure that the concordance is always up-to-date by using a shell script.

```
# This runs the concordance program.
gcl<conctex.lsp
# Now I run tex on my input file.
tex transcription
# This executes the file "warnings"
# It prints the warning messages
# from conclsp.lsp to standard output.
sh warnings
```

It's a good idea to use a shell script anyway, or the equivalent in your operating system, if you're using two-pass features, i.e., for an index, table of contents, page references, etc.

A special case

Manuscripts often have peculiarities that are difficult to represent in set type. Often, these peculiarities occur too rarely for it to be worthwhile writing TeX macros and/or Lisp functions to cope with them, but it is often possible to invent an *ad hoc* solution. For instance, Holm perg 11 4^o has words that are written vertically, and are therefore, so to speak, broken over several lines. This problem can be solved in the following way:

```
\newskip\tempskip
\newskip\normalbaselineskip
\tempskip=.75\baselineskip
\normalbaselineskip=\baselineskip
\font\enormous=cmr17 scaled 7500
\baselineskip=\tempskip
\setbox0=\hbox{\enormous D}}%
* \lineno{1\dash 7}\copy0
\vskip-\ht0\vskip-2pt
\dimen0=\wd0\advance\dimen0 by 18pt
\hangindent\dimen0\hangafter-7
r\break o\break t \break n \break i
\break n \break g\par
\baselineskip=\normalbaselineskip *
\vskip.667\baselineskip

@ (add-occurrences "drotning" "1va~1--7")

\lineno{8} himins _{ok}
iar{\dh}ar.~s{\ae}l {ok} dyr{\dh}\-
\lineno{9} lig m{\o}r Maria.~mo{\dh}ir
d_{ro}tti_{n}s\
```

Produces the following output:

1-7: **D**<sub>r
o
t
n
i
n
g</sub>

8: himins ok iarðar. sæl ok dyrð

9: lig mör Maria. moðir drottins

Since `process-line` ignores all lines within a commentary, the complex construction for the word “Drotning” isn’t read by `process-line`, so this occurrence needs to be set explicitly.

`@(add-occurrences |drotning| "1va~1-7")`

The function `add-occurrences` uses `access-occurrences` to access the appropriate `occurrences slot` of the `word structure` bound to the symbol in its first argument, and appends its second argument, a string, to the list in the `occurrences slot`. It can be any string, so `access-occurrences` can be used to make non-standard occurrences by hand. An occurrence like “1va 1-7” cannot be created by `conctex.lsp`’s ordinary routines.

Final remarks

ConcTeX demonstrates the power of Lisp and TeX in combination. Designing and typesetting a manuscript transcription, usually as part of a facsimile edition, is a challenge under the best of circumstances, and generating a concordance is always a time-consuming task. I do not promise miracles with ConcTeX, but I do believe that it can make both of these tasks easier, and indeed possible for non-professionals.

ConcTeX’s most significant advantages are:

1. It makes it possible to take advantage of the typographic capabilities of TeX and METAFONT.
2. It uses the very same file for typesetting and generating the concordance.
3. It performs alphabetical sorting on arbitrary special characters

ConcTeX is designed to be extendable. It would be possible to adapt it for use with other languages. For languages that are written left-to-right, it will only be necessary to cope with the usual difficulties with fonts and character encoding. For right-to-left text or a mixture of left-to-right and right-to-left text, the difficulties are greater, but it should be possible to solve them. Many of its features are of general utility and could be used for other kinds of programs that extract data from TeX input files.

However, there is one significant problem from the point of view of book design. The virtues of

Computer Modern and its offspring notwithstanding, there simply aren’t enough METAFONT fonts in the public domain suitable for use in fine printed books. Understandably, most of the other fonts available either contain special symbols or alphabets for non-Western languages, and are generally designed to be compatible with Computer Modern. But even if the Computer Modern fonts were the most beautiful fonts in the world, not every book should be printed in them. I admire Knuth’s accomplishment and I like Computer Modern, but Monotype Modern 8A, on which it is based, is not universally admired.⁴² It is possible to use PostScript fonts with TeX, but it’s inconvenient, and although they are well-designed, they have the serious defect that most sizes are produced by simple magnification or reduction. I consider it an important desideratum that more METAFONT fonts are created and made available, but I don’t see how this goal can be accomplished by amateur programmers writing free software. But until such fonts are available, books typeset without the financial and technical backing of a publisher will continue to suffer from a poverty of fonts.

I am far from being an authority on the subject, but I doubt very much that any photolithographic printing technique will ever be able to equal the quality of impression of lead type. My dream is to use METAFONT for designing lead type for use on a TeX-driven, Linotype-like linecasting machine. I say a linecasting machine only because I think that it would be easier to implement glue on a linecasting machine. It might, however, be an interesting exercise to design a TeX-driven Monotype-like typesetting machine, or even a typesetting machine based on a different principle, such as casting an entire page at once. Perhaps with such machines, we can finally equal and perhaps even surpass the great typographic achievements of the past.

Sample texts and concordances

Concordances become long very quickly, so this section contains three mini-concordances and the texts from which they were generated.

The first text is the beginning of the transcription of Holm perg 11 4^o, prepared by Dr. Wilhelm Heizmann. It illustrates the use of *line footnotes*. These are footnotes which refer to lines in the

⁴² WILLIAMSON, in reference to the English Monotype Corporation’s Modern Extended series 7: “Not particularly distinguished in letter form, the face has become familiar to readers of scientific works; for some years, this was one of the few series equipped with a full range of mathematical and other special sorts.” (p. 130)

transcription. Unlike ordinary footnotes, there is no indication in the running text, such as an asterisk, a dagger, or a superscripted numeral, but the line number or numbers appear in the footnote where the footnote indicator usually goes.

Note that a different definition of `\ustroke` is used here than in the foregoing article. The emendations are in italics and underlined, but not enclosed in brackets.

Holm perg 11 4^o
1va

próloGus

1–7: **D**
r
o
t
n
i
n
g

- 8: himins ok iarðar. sæl ok dyrð
9: lig mör Maria. moðir drottins
10: Iesus Xristz. blomi hreinlifis.
11: herbergi heilags anda. øllvm
12: helgum mønnum æðri. helgari
13: ok haleitari. er komin at kyn
14: ferðj af kongligri ætt eptir þvi sem
15: segir en gaufgi ken*ni* Maðr [ok enn]

Concordance to Holm perg 11 4^o

- æðri (1): 1va 12.
ætt (1): 1va 14.
af (1): 1va 14.
(allr)
øllvm (1): 1va 11^e.
(andi)
anda (1): 1va 11.
at (1): 1va 13.
blomi (1): 1va 10.
drotning (1): 1va 1-7.
(drottinn)
drottins (1): 1va 9^e.
(dyrðligr)
dyrðlig (1): 1va 8–9.
enn (2): 1va 15, 15^e.
eptir (1): 1va 14.
er (1): 1va 13.
(göfugr)

1–7 Drotning, bis auf das initiale D sind die Buchstaben untereinander angeordnet.

8/9 dyrðlig, dafür in Ausg. zumeist dýrlig.

10 Iesus, 233 Jesu, Ausg. für den Genitiv immer Jesu; Xristz, 233 cristi, Ausg. für Xrist- fast immer Crist-, einige Male auch Krist-.

- gaufgi (1): 1va 15.
(haleitr)
haleitari (1): 1va 13.
(heilagr)
heilags (1): 1va 11.
helgari (1): 1va 12^e.
helgum (1): 1va 12^b.
herbergi (1): 1va 11.
(himinn)
himins (1): 1va 8.
(hreinlifi)
hreinlifis (1): 1va 10^e.
Iesus (1): 1va 10^b.
(iørðr)
iarðar (1): 1va 8.
kennimaðr (1): 1va 15.
(koma)
komin (1): 1va 13.
(kongligr)
kongligri (1): 1va 14.
(kynferð)
kynferðj (1): 1va 13–14.
(maðr)
mønnum (1): 1va 12.
Maria (1): 1va 9.
moðir (1): 1va 9.
mör (1): 1va 9.
ok (4): 1va 8 (2), 13^b, 15.
(sæll)
sæl (1): 1va 8.
(segja)
segir (1): 1va 15^b.
sem (1): 1va 14^e.
(Xrist)
Xristz (1): 1va 10.
þvi (1): 1va 14.

The Bible

Genesis

1. In the beginning God created the heavens and the earth. 2. The earth was without form and void, and darkness was upon the face of the deep; and the Spirit of God was moving over the face of the waters. 3. And God said, “Let there be light”; and there was light. 4. And God saw that the light was good; and God separated the light from the darkness. 5. God called the light Day, and the darkness he called Night. And there was evening and there was morning, one day.

Concordance to the Bible

and (11): *Gen.* 1:1, 2 (3), 3 (2), 4 (2), 5 (3).

be (1): *Gen.* 1:3.
 was (7): *Gen.* 1:2 (3), 3, 4, 5 (2).
 beginning (1): *Gen.* 1:1.
 (call)
 called (2): *Gen.* 1:5 (2).
 (create)
 created (1): *Gen.* 1:1.
 darkness (3): *Gen.* 1:2, 4, 5.
 day (2): *Gen.* 1:5 (2).
 deep (1): *Gen.* 1:2.
 earth (2): *Gen.* 1:1, 2.
 evening (1): *Gen.* 1:5.
 face (2): *Gen.* 1:2 (2).
 form (1): *Gen.* 1:2.
 from (1): *Gen.* 1:4.
 God (6): *Gen.* 1:1, 2, 3, 4 (2), 5.
 good (1): *Gen.* 1:4.
 he (1): *Gen.* 1:5.
 (heaven)
 heavens (1): *Gen.* 1:1.
 in (1): *Gen.* 1:1.
 let (1): *Gen.* 1:3.
 light (5): *Gen.* 1:3 (2), 4 (2), 5.
 morning (1): *Gen.* 1:5.
 (move)
 moving (1): *Gen.* 1:2.
 night (1): *Gen.* 1:5.
 of (3): *Gen.* 1:2 (3).
 one (1): *Gen.* 1:5.
 over (1): *Gen.* 1:2.
 (say)
 said (1): *Gen.* 1:3.
 (see)
 saw (1): *Gen.* 1:4.
 (separate)
 separated (1): *Gen.* 1:4.
 spirit (1): *Gen.* 1:2.
 that (1): *Gen.* 1:4.
 the (14): *Gen.* 1:1 (3), 2 (6), 4 (3), 5 (2).
 there (4): *Gen.* 1:3 (2), 5 (2).
 upon (1): *Gen.* 1:2.
 void (1): *Gen.* 1:2.
 (water)
 waters (1): *Gen.* 1:2.
 without (1): *Gen.* 1:2.

Die Bibel

Das 1. Buch Mose (Genesis)

(With homograph identifiers.)

1. Am Anfang schuf Gott Himmel und Erde.
 2. Und die{fsn} Erde war wüst und leer, und es war finster auf der{fds} Tiefe; und der{mns} Geist Gottes schwebte auf dem{nds} Wasser. 3. Und Gott

sprach: Es werde Licht! Und es ward Licht. 4. Und Gott sah, daß das{nas} Licht gut war. Da schied Gott das{nas} Licht von der{fds} Finsternis 5. Und nannte das{nas} Licht Tag und die{fas} Finsternis Nacht. Da ward aus Abend und Morgen der{mns} erste Tag.

Die Bibel

Das 1. Buch Mose (Genesis)

(Without homograph identifiers.)

1. Am Anfang schuf Gott Himmel und Erde.
 2. Und die Erde war wüst und leer, und es war finster auf der Tiefe; und der Geist Gottes schwebte auf dem Wasser. 3. Und Gott sprach: Es werde Licht! Und es ward Licht. 4. Und Gott sah, daß das Licht gut war. Da schied Gott das Licht von der Finsternis 5. Und nannte das Licht Tag und die Finsternis Nacht. Da ward aus Abend und Morgen der erste Tag.

Konkordanz zur Bibel

Abend (1): *Gen.* 1:5.
 (an)
 am (1): *Gen.* 1:1.
 Anfang (1): *Gen.* 1:1.
 auf (2): *Gen.* 1:2 (2).
 aus (1): *Gen.* 1:5.
 da (1): *Gen.* 1:5.
 (das [n.])
 das [akk. sg.] (2): *Gen.* 1:4, 5.
 dem [dat. sg.] (1): *Gen.* 1:2.
 der [m.] (2): *Gen.* 1:2, 5.
 die [f.] (1): *Gen.* 1:2.
 die [akk. sg.] (1): *Gen.* 1:5.
 der [dat. sg.] (2): *Gen.* 1:2, 4.
 Erde (2): *Gen.* 1:1, 2.
 (erst)
 erste (1): *Gen.* 1:5.
 es (3): *Gen.* 1:2, 3 (2).
 finster (1): *Gen.* 1:2.
 Finsternis (2): *Gen.* 1:4, 5.
 Geist (1): *Gen.* 1:2.
 Gott (2): *Gen.* 1:1, 3.
 Gottes (1): *Gen.* 1:2.
 Himmel (1): *Gen.* 1:1.
 leer (1): *Gen.* 1:2.
 Licht (4): *Gen.* 1:3 (2), 4, 5.
 Morgen (1): *Gen.* 1:5.
 Nacht (1): *Gen.* 1:5.
 (nennen)
 nannte (1): *Gen.* 1:5.
 (schaffen)

schuf (1): *Gen.* 1:1.
 (schweben)
 schwebte (1): *Gen.* 1:2.
 (sein [*verb*])
 war (2): *Gen.* 1:2 (2).
 (sprechen)
 sprach (1): *Gen.* 1:3.
 Tag (2): *Gen.* 1:5 (2).
 Tiefe (1): *Gen.* 1:2.
 und (10): *Gen.* 1:1, 2 (4), 3 (2), 5 (3).
 von (1): *Gen.* 1:4.
 Wasser (1): *Gen.* 1:2.
 (werden)
 werde (1): *Gen.* 1:3.
 ward (2): *Gen.* 1:3, 5.
 wüst (1): *Gen.* 1:2.

Category code list

- `\catcode'\<=\active`. For homograph identifiers. Reset to 12 (other) in math mode.
- `\catcode'\@=14` (Comment). Treated as a comment by \TeX , equivalent to `%`. If `@` is the first non-blank character in an input line, the Lisp interpreter evaluates the rest of the line, which should contain a balanced expression (`sexp`). Otherwise, if it's in a text line, `conctex.lsp` discards it and the rest of the line following it.
- `\catcode'\#9` (Ignored). The character `¥` (decimal 165, octal `245`, hexadecimal `A5`) is ignored by \TeX and treated as a *word separator* by Lisp.
- `\catcode'*=9` or `\active`. Used for commentaries. Reset to `\catcode 12` in math mode.
- `\catcode'_=\active`. The underline character is `\let` to `\ustroke`. Reset to 8 (subscript) in the `\plain` environment, so `_` can appear in the names of files loaded using `\input`, and math mode, so it can be used for subscripts.
- `\catcode'\-=\active`. The hyphen character `-` is used for line ends where words are broken. Reset to 12 (other) in commentaries, the `\plain` environment, and math mode.
- `\catcode'\^~M=\active`. This change is local to the expansions of the active character `-` and the redefined control symbol `\-`, where `^~M` is `\let` to `\empty`. Otherwise, it has its normal `\catcode` of 5 (end of line).
- `\catcode'\©=\active`. The copyright symbol (decimal 169, octal `251`, hexadecimal `A9`) is `\let` to `\-` before the latter is redefined, so it

can be used for discretionary hyphens within transcription lines, if necessary.

Glossary

Braces, unmotivated: Braces that delimit unnecessary groups in the input file. Not permitted in *transcription lines*.

Comment: Comments can be normal \TeX comments that follow a `%`. Usually, however, “comment” refers to invocations of the macros `\begincomment` and `\endcomment`.

```
\begincomment{This is a comment.}
\endcomment
```

Comments appear in the output only if `\drafttrue`. Comments are ignored by `conctex.lsp`.

Commentary: A commentary contains text which should appear within the *transcription lines* in the *output*, but which should not be used for generating the concordance. Commentaries can be coded in several ways.

```
* This is a commentary. *
\begincommentary This is also
  a commentary.\endcommentary
\plain Yet another commentary.
\endplain
```

Evaluated lines: Lines in the *input file*, where `@` is the first non-blank character. Such lines must contain a balanced Lisp expression, which is evaluated by the Lisp interpreter. \TeX ignores lines beginning with `@`.

Homograph identifiers: In \TeX , strings of the form `<<(string)>>`. Used for indicating homographs in the input file. Printed out or not, depending on the value of `\ifdraft`. For the Lisp program, homograph-identifiers can be set in various ways in the lemmatization dictionary.

Ignored lines: Lines in the *input file* that are ignored by \TeX , `conctex.lsp`, or both. Completely blank lines are ignored by `conctex.lsp` and treated normally by \TeX . Lines beginning with `%` are ignored both by \TeX and `conctex.lsp`. If a line contains a `%` in any other position, the rest of the line is discarded by both \TeX and `conctex.lsp`. The character `@` is equivalent to `%` in \TeX . If a line contains an `@` that's not the first non-blank character in the line, `conctex.lsp` discards the rest of the line.

Input file: A file containing text to be typeset for a book containing a facsimile of a manuscript, or

something similar. The input file or files are also used for generating a concordance.

Lemmatization dictionary: A file of Lisp code used for lemmatizing the words in the transcription. The file can contain invocations of the functions `generate-entry`, `add-variants`, and/or `harmless`.

Macro, delimited: A macro within braces, like `{\%}`, `{\rm ...}`, or `{\dh}`

Macro, undelimited: A macro with no enclosing braces, like `\%`, `\rm` or `\indexentry{nouns}{x}%{verbs}{}`.

Output: The typeset result of running \TeX on the *input file* or *files*. Technically, the result of running \TeX is a `dvi` file, however I usually mean either the paper printout or a display on the computer terminal in a program like `xdvi` or `Ghostview`.

Text lines: Lines in the *input file* which are processed by \TeX . They may be *transcription lines* or *commentaries*.

Transcription lines: Lines in the *input file* containing the transcription of the manuscript. They should correspond, for the most part, to the actual lines in the manuscript; however, they may also contain *commentaries*, which may affect the length and hence the line breaking in the output.

Word element: A character which `conctex.lsp` considers to be part of a word. Includes all letters (characters whose `\catcode` is 11), some characters of type “other” (`\catcode` 12), and special character macros.

Word separator: Characters that cannot be part of a word. Currently these are blanks, punctuation, and the character which is represented as ¥ on my terminal (decimal 165, octal 245, hexadecimal A5).

References

- The Bible. Containing the Old and New Testaments. Revised Standard Version.* American Bible Society. New York: 1952.
- Die Bibel. Mit Apokryphen.* Nach der Übersetzung Martin Luthers neubearbeitet. Deutsche Bibelgesellschaft. Stuttgart: 1985.
- KNUTH, Donald E. *The \TeX book.* Addison-Wesley Publishing Company. Reading, Mass.: 1986.
- STEELE, Guy L., *Common Lisp. The Language.* 2nd ed. Digital Press. 1990.
- CLEASBY, Richard and Gudbrand Vigfusson. *An Icelandic-English Dictionary.* 2nd ed. The Clarendon Press. Oxford: 1957.

WILLIAMSON, Hugh. *Methods of Book Design. The Practice of an Industrial Craft.* 3rd ed. Yale University Press. New Haven: 1983.

WINSTON, Patrick Henry and Berthold Klaus Paul Horn. *LISP.* 3rd ed. Addison-Wesley Publishing Company. Reading, Mass.: 1989.

◊ Laurence Finston
Skandinavisches Seminar
Georg-August-Universität
Humboldtallee 13
D-37073 Göttingen, Germany
lfinsto1@gwdg.de