## Indexing, glossaries, and bib2gls

Nicola L. C. Talbot

**Abstract**

The bib2gls command line application [17] combined with the glossaries-extra package [18] provides an alternative indexing method to those provided by the base glossaries package [19]. In addition, since the terms are defined in .bib files, tools such as JabRef [3] can be used to manage large databases.

### 1 Introduction

The LaTeX kernel provides two basic forms of indexing (that is, collating terms and their associated locations in the document). The first form creates a file called \jobname.idx using \makeindex and the information is written to the file using the command \index{⟨*info*⟩}. This writes the line

\indexentry{⟨*info*⟩}{⟨*page*⟩}

to the .idx file, where ⟨*page*⟩ is the page number.

The second form is very similar but uses a different file extension. A file called \jobname.glo is created with \makeglossary and the information is written using the command \glossary{⟨*info*⟩} which writes the line

\glossaryentry{⟨*info*⟩}{⟨*page*⟩}

to the .glo file, where ⟨*page*⟩ is the page number.

In both cases, the page number is obtained from \thepage and the write operation is delayed to ensure the value is correct in the event that the indexing occurs across a page break. These commands date back to the 1980s when processing power and resources were significantly smaller than today. Compilation of draft documents could be speeded up by omitting the indexing, which can be done by commenting out the \make... commands or by inserting \nofiles before them.

The next step is to collate the {⟨*info*⟩}{⟨*page*⟩} information, removing duplicates, concatenating page ranges, hierarchically ordering the terms, and writing the LaTeX code that will typeset the result as an index or glossary. TeX isn't particularly suited for this kind of task. It's much more efficient to use a custom application; makeindex was created for this purpose, which creates a .ind file from the .idx.

### 2 Indexing

The LaTeX kernel doesn't provide any specific commands for reading the file created by the indexing application, but instead defers this task to packages. The first of these was makeidx [5], which is quite trivial. It provides the command \printindex that inputs \jobname.ind if it exists and provides commands for convenient 'see' and 'see also' cross-referencing.

#### 2.1 Indexing example

A simple example that uses the lipsum package [2] for padding follows:

```
\documentclass{article}
\usepackage{lipsum}
\usepackage{makeidx}
\makeindex
\begin{document}
\index{duck}\lipsum*[1]\index{goose}
\par\lipsum[2-4]\par
\index{duck}\index{ant}\lipsum*[5-6]
\index{zebra}\par
\index{goose}\index{aardvark}\lipsum[7-10]\par
\lipsum*[11]\index{dog}\index{ant}\index{goose}
\printindex
\end{document}
```

If the file is called myDoc.tex then[1]

```
latex myDoc
```

will create the file myDoc.idx that contains

```
\indexentry{duck}{1}
\indexentry{goose}{1}
\indexentry{duck}{1}
\indexentry{ant}{1}
\indexentry{zebra}{2}
\indexentry{goose}{2}
\indexentry{aardvark}{2}
\indexentry{dog}{3}
\indexentry{ant}{3}
\indexentry{goose}{3}
```

At this point the output doesn't contain an index, as the file myDoc.ind (which \printindex attempts to read) doesn't exist. This file can be created with

```
makeindex myDoc
```

The document then needs to be rerun to include the new myDoc.ind so the complete document build is

```
latex myDoc
makeindex myDoc
latex myDoc
```

The default behaviour of makeindex is to assume the extension .idx for the input file (if not specified) and use the extension .ind for the output file (which fits the expected extension used in \printindex). In this case, it creates a myDoc.ind containing (except the blank lines around each group are omitted, here and in the following):

```
\begin{theindex}
  \item aardvark, 2
  \item ant, 1, 3
```

[1] latex is used here to denote pdflatex, xelatex, etc., as appropriate.

```
  \indexspace
  \item dog, 3
  \item duck, 1
  \indexspace
  \item goose, 1--3
  \indexspace
  \item zebra, 2
\end{theindex}
```

The `\indexspace` command usually produces a visual separation between letter groups. Neither that command nor the `\begin{theindex}` environment are provided by the LaTeX kernel, so are defined by classes that provide index support.

If you try out this example, you'll find that `\index` doesn't produce any text where it's used in the document. This catches out some new users who expect the indexed term to also appear in the document. Typically, `\index` will be placed either before or after the appropriate word. For example:

```
Aardvarks\index{aardvark} are
\index{nocturnal animal}nocturnal animals.
```

The default output created by `makeindex` can be modified by a style file. For example, if I create a file called `myindexstyle.ist` that contains

```
headings_flag 1
heading_prefix "\\heading{"
heading_suffix "}"
```

and pass this file to `makeindex`:

```
makeindex -s myindexstyle.ist myDoc
```

then the resulting `myDoc.ind` will now contain

```
\begin{theindex}
\heading{A}
  \item aardvark, 2
  \item ant, 1, 3
  \indexspace
\heading{D}
  \item dog, 3
  \item duck, 1
  \indexspace
\heading{G}
  \item goose, 1--3
  \indexspace
\heading{Z}
  \item zebra, 2
\end{theindex}
```

This custom command `\heading` will need to be defined somewhere in my document. A basic example:

```
\newcommand*{\heading}[1]{%
 \item\textbf{#1}\par\nobreak\indexspace\nobreak}
```

Some newer, more sophisticated, classes (such as memoir [8]) and packages (such as imakeidx [1]) provide greater flexibility, making it easier to customize the index format.

The obsolete glossary package [12] provided an analogous version of makeidx designed for use with `\makeglossary` and `\glossary`. This was made more complicated by the need to provide some kind of separation between the term and its description to assist formatting, but it was essentially using the same kind of mechanism as the above indexing example. The memoir and nomencl [9] packages provide similar functions. As with `\index`, `\nomenclature` (as provided by nomencl) and `\glossary` do not produce any text where they're used in the document.

### 2.2 Indexing syntax

The ⟨*info*⟩ argument of both `\index` and `\glossary` needs to be given in the syntax of the indexing application used to process the data.[2] This catches out many new users who may still be learning (La)TeX syntax and don't realise that external tools may have different special characters.

For `makeindex`, the special characters are:

- The 'actual' character used to separate the sort value from the actual term when they're different (default: `@`).
- The 'level' character used to separate hierarchical levels (default: `!`).
- The 'encap' character used to indicate the page number encapsulating command (default: `|`).
- The 'quote' character used to indicate that the following character should be interpreted literally (default: `"`).

These characters can be changed in the `makeindex` style file, if required.

For example, using the defaults,

```
\index{deja vu@\emph{d\'ej\`a vu}}
```

This indicates that the term should be sorted as '`deja vu`' but the term will be written to the output (`.ind`) file as

```
\emph{d\'ej\`a vu}
```

Up to three hierarchical levels are supported by `makeindex`, and also by the standard definition of `\begin{theindex}`:

```
\index{animal!nocturnal!owl}
\index{animal!nocturnal!aardvark}
\index{animal!crepuscular!ocelot}
```

This is converted by `makeindex` to

```
  \item animal
    \subitem crepuscular
      \subsubitem ocelot, 1
    \subitem nocturnal
```

[2] This refers to the kernel definitions of `\index` and `\glossary` which simply take one mandatory argument that's written to the relevant file.

```
  \subsubitem aardvark, 1
  \subsubitem owl, 1
```

(assuming the indexing occurred on page 1).

Each hierarchical level may have a separate sort and actual value. For example (except on one line),

```
\index{debutante@d\'ebutante!1945-1958@1945
\textendash1958}
```

Here, the top-level item has the sort value `debutante` (used by the indexing application to order the top-level entries) with the actual value `d\'ebutante` used for printing in the document's index.

The sub-item has the sort value `1945-1958` (used by the indexing application to order sub-entries relative to their parent entry) with the actual value `1945\textendash 1958` used within the document's index.

If a parent entry is also indexed within the document, it must exactly match its entry within the hierarchy. A common mistake is something like

```
\index{debutante@d\'ebutante}
\index{d\'ebutante!1945-1958@1945\textendash 1958}
```

In this case, `makeindex` treats `d\'ebutante` and `debutante@d\'ebutante` as two separate top-level entries, which results in an index where 'débutante' appears twice, and one entry doesn't have any sub-items while the other does.

The corresponding page number (location) can be encapsulated by a command using the encap special character. This should be followed by the command name without the leading backslash. For example, if on page 2 of my earlier example document I add an encap to the aardvark entry,

```
\index{aardvark|textbf}
```

then the resulting `.ind` file created by `makeindex` will now contain (`makeindex` inserts the \):

```
  \item aardvark, \textbf{2}
```

Other commands could be included, for example,

```
\index{aardvark|bfseries\emph}
```

would end up in the `.ind` file as

```
  \item aardvark, \bfseries\emph{2}
```

but obviously this would lead to the undesired effect of rendering the rest of the index in bold. A better solution is to define a semantic command that performs the required font change, such as

```
\newcommand*{\primary}[1]{\textbf{\emph{#1}}}
```

If the encapsulating command takes more than one argument, the final argument needs to be the page number and the initial arguments need to be added to the encap. For example,

```
\index{aardvark|textcolor{blue}}
```

The makeidx package provides two commands that can be used in this way:

```
\newcommand*\see[2]{\emph{\seename} #1}
\providecommand*\seealso[2]{\emph{\alsoname} #1}
```

(The `\seename` and `\alsoname` macros are language-sensitive commands that produce the text 'see' and 'see also'.) These commands both ignore the second argument, which means that the page number won't be displayed. This provides a convenient way of cross-referencing. For example, if on page 2 I have

```
\index{ant-eater|seealso{aardvark}}
```

then the `.ind` file would contain

```
  \item ant-eater, \seealso{aardvark}{2}
```

From `makeindex`'s point of view, this is just another encapsulating command, so if I add

```
\index{ant-eater}
```

to page 1 and page 8, then this would lead to a rather odd effect in the index:

```
  \item ant-eater, 1, \seealso{aardvark}{2}, 8
```

The page list now displays as '1, *see also* aardvark, 8'.

The simple solution is to place all the cross-referenced terms before `\printindex`. (imakeidx closes the indexing file at the start of `\printindex`, which means that indexing can't take place after it.)

The encap value may start with `(` or `)` to indicate the start or end of an explicit range. If used, these must match. For example, on page 2:

```
\index{aardvark|(textbf}
```

and then on page 10:

```
\index{aardvark|)textbf}
```

results in

```
  \item aardvark \textbf{2--10}
```

If no formatting is needed, `(` and `)` may be used alone:

```
\index{aardvark|(}
...
\index{aardvark|)}
```

Although these parentheses characters have a special meaning at the start of the encap, they're not considered special characters.

If any of the special characters need to be interpreted literally, then they must be escaped with the quote character. For example,

```
\index{n"!@$n"!$}
\index{x@$"|\vec{x}"|$}
```

In the first case above, the special character that needs to be interpreted literally is the level character `!` which appears in both the sort value and the actual value. In the second case, the special character that needs to be interpreted literally is the encap character `|` which appears twice in the actual value. (Of course, replacing `|` with `\vert` avoids the problem.)

This is something that often trips up new users. With experience, we may realise that providing semantic commands can hide the special characters from the indexing application. For example,

```
\newcommand*{\factorial}[1]{#1!}
```

This can take care of the actual value but not the sort value, which still includes a special character:

```
\index{n"!@$\factorial{n}$}
```

The quote character itself also needs escaping if it's required in a literal context:

```
\index{naive@na\""\i ve}
```

From `makeindex`'s point of view the backslash character \ is a literal backslash so `\""` is a backslash followed by a literal double-quote (which has been escaped with `"`).

### 2.3 UTF-8

So far, all examples that include accented characters have used accent commands, such as `\'`, since `makeindex` doesn't support UTF-8. This is essentially down to its age, as it was written in the mid-1980s, before the advent of Unicode. A previous *TUGboat* article [13] highlights the problem caused when trying to use `makeindex` on a UTF-8 file.

Around 2000, `xindy` [4], a new indexing application written in Perl and Lisp, was developed as a language-sensitive, Unicode-compatible alternative to `makeindex`. The native `xindy` format is quite different from the `makeindex` syntax described above and can't be obtained with `\index`. In this case, the special characters are the double-quote `"` used to delimit data and the backslash \ used to indicate that the following character should be taken literally.

In the earlier factorial example, the `makeindex` syntax (used in the `.idx` file) is

```
\indexentry{n"!@$\factorial{n}$}{1}
```

(assuming `\index{n"!@$\factorial{n}$}` occurred on page 1). Whereas in the native `xindy` format this would be written as

```
(indexentry
 :tkey (("n!" "$\\factorial{n}$"))
 :locref "1")
```

or

```
(indexentry
 :key ("n!")
 :print ("$\\factorial{n}$")
 :locref "1")
```

The exclamation mark doesn't need escaping in this case but the backslash does. The `na\""\i ve` example above needs both the backslash and double-quote treated in a literal context:

```
(indexentry
 :tkey (("naive" "na\\\"\\i ve"))
 :locref "1")
```

Of course, in this case UTF-8 is preferable:

```
(indexentry :key ("naïve") :locref "1")
```

This format requires a completely different command than `\index` for use in the document. However, `xindy` is capable of reading `makeindex` syntax. The simplest way of enabling this is by invoking `xindy` through the wrapper program `texindy`. Unfortunately, unlike `makeindex`, there's no way of changing the default special characters. The previous *TUGboat* article on testing indexing applications [13] compares `makeindex` and `xindy`.

A recent alternative that's also Unicode compatible is the Lua program `xindex` [20]. This reads `makeindex` syntax and command line switches are available to change the special characters or to specify the language.

### 2.4 Shortcuts

The `\index` command doesn't generate any text. This can lead to repetition in the code. For example,

```
An aardvark\index{aardvark} is a
nocturnal animal\index{nocturnal animal}.
```

It's therefore quite common to see users provide their own shortcut command to both display and index a term. For example,

```
\newcommand*{\Index}[1]{#1\index{#1}}
%...
An \Index{aardvark} is a
\Index{nocturnal animal}.
```

Complications arise when variations are required. For example, if a page break occurs between 'nocturnal' and 'animal', so that 'nocturnal' is at the end of, say, page 1 and 'animal' is at the start of page 2, then placing `\index` after the term leads to the page reference 2 in the index whereas placing it before leads to the page reference 1. Also `\index` creates a whatsit that can cause interference. Although examples quite often place `\index` after the text, in many cases it's more appropriate to put `\index` first. This shortcut command doesn't provide the flexibility of the placement of `\index` relative to the text.

A problem also arises if the term includes special characters that need escaping in `\index` but not in the displayed text or if the display text needs to be a slight variation of the indexed term. For example, the above definition of `\Index` can't be used in the following:

```
The na\"\i ve\index{naive@na\""\i ve}
geese\index{goose} were frightened by the
flock of ph\oe nixes\index{phoenix@ph\oe nix}.
```

The definition of \Index could be modified to include an optional argument to provide a different displayed term. For example:

```
\newcommand*{\Index}[2][\thedisplayterm]{%
 \def\thedisplayterm{#2}%
 #1\index{#2}}
```

but this 'shortcut' ends up with slightly longer code:

```
The \Index[na\"\i ve]{naive@na\""\i ve}
\Index[geese]{goose} were frightened by the
flock of \Index[ph\oe nixes]{phoenix@ph\oe nix}.
```

An obvious solution to the first case (naïve) is to use UTF-8 instead of LaTeX accent commands combined with a Unicode-aware indexing application (texindy or xindex).

```
The \Index{naïve} geese\index{goose}
were frightened by the flock of
phœnixes\index{phœnix}.
```

This works fine with a Unicode engine (XeLaTeX or LuaLaTeX) but not with inputenc [6], which uses the so-called active first octet trick to internally apply accent commands. This means that the .idx file ends up with

```
\indexentry{na\IeC {\"\i }ve}{1}
\indexentry{goose}{1}
\indexentry{ph\IeC {\oe }nix}{1}
```

The \index mechanism is designed to write its argument literally to the .idx file. This can be seen from the earlier \factorial example where

```
$\factorial{n}$\index{$\factorial{n}$}
```

is written to the .idx file as

```
\indexentry{$\factorial{n}$}{1}
```

Unfortunately, embedding \index in the argument of another command (such as the above custom \Index) interferes with this. For example,

```
\Index{$\factorial{n}$}
```

results in the expansion of \factorial as the indexing information is written to the .idx file:

```
\indexentry{$n!$}{1}
```

The level special character (!) is no longer hidden from the indexing application and, since it hasn't been escaped with the quote character, this leads to an unexpected result in the .ind file:

```
  \item $n
    \subitem $, 1
```

### 2.5 Consistency

With makeindex, invisible or hard to see differences in the argument of \index can cause seemingly duplicate entries in the index. For example (line breaks here are part of the input),

```
\index{debutante@d\'ebutante
!1945-1958@1945\textendash 1958}
%...
\index
{debutante@d\'ebutante!1945-1958@1945\textendash
1958}
```

Here the two entries superficially appear the same but the line break inserted into the first instance results in two different entries in the index. The first entry has a space at the end of its actual value but the second doesn't. This is enough to make them appear different entries from makeindex's point of view. When viewing the .ind file, the difference is only perceptible if the text editor has the ability to show white space.

The simplest solution here is to run makeindex with the -c option, which compresses intermediate spaces and ignores leading and trailing blanks and tabs.

A long document with a large index of hierarchical terms and terms that require a non-identical sort value can be prone to such mistakes. Other inconsistencies can arise through misspellings (which hopefully the spell-checker will detect) or more subtle errors that are missed by spell-checkers.

For example, in English some words are hyphenated ('first-rate'), some are merged into a single word ('firstborn') and some are space-separated ('first aid'). Even native speakers can mix up the separator, and this can result in inconsistencies in a large document. For example,

```
\index{firstborn}
%...
\index{first-born}
%...
\index{first born}
```

From the spell-checker's point of view, there are no spelling errors to flag in the above code. The inconsistencies can be picked up by proof-reading the index, but unfortunately some authors skip the back matter when checking their document.

When using \glossary (rather than \index), which may additionally include a long description, the problem with consistency becomes more pronounced. The example document below illustrates the use of the kernel version of \glossary, with one regular entry and one range entry. All the strings have to be exactly the same.

Also shown here is that since makeindex is a trusted application, it can be run through the shell escape in restricted mode. Finally, a makeindex style file is needed to indicate that data is now marked up with \glossaryentry instead of \indexentry:

```
\documentclass{report}
\begin{filecontents*}{\jobname.ist}
```

```
keyword "\\glossaryentry"
\end{filecontents*}

\IfFileExists{\jobname.glo}
{\immediate\write18{makeindex -s \jobname.ist
                   -o \jobname.gls \jobname.glo}}
{\typeout{Rerun required.}}

\makeglossary
\begin{document}
\chapter{Introduction}
Duck\glossary{duck: a waterbird
with webbed feet}\ldots

\chapter{Ducks}
\glossary{duck: a waterbird
with webbed feet|(}
\ldots
\glossary{duck: a waterbird
with webbed feet|)}

\renewcommand{\indexname}{Glossary}
\makeatletter
\@input@{\jobname.gls}
\makeatother
\end{document}
```

The old glossary package introduced a way of saving the glossary information and referencing it by label to perform the indexing, which helped consistency and reduced document code. The term could then be just indexed by referencing the label with `\useglosentry`, or could be both indexed and displayed with `\gls{`⟨*label*⟩`}`. Special characters still needed to be escaped explicitly, and this caused a problem for `\gls` as the quote character ended up in the document text. Abbreviation handling was performed using a different indexing command, and the package reached the point where it far exceeded its original simplistic design. It was time for a completely new approach, which we turn to now.

## 3 The glossaries package

The glossaries package [19] was introduced in 2007 as a replacement to the now obsolete glossary package. The main aims were to

- define *all* terms so that they can be referenced by label (no document use of `\glossary`);
- internally escape indexing special characters so that the user doesn't need to know about them;
- make abbreviations use the same indexing mechanism for consistency.

The advantage of first defining terms so that they can be referenced by label isn't only to help consistency but also improves efficiency. When a term is defined, partial indexing information is constructed and saved. This is the point where any special characters are escaped, which means that this operation only needs to be done when the term is defined, not every time the term is indexed.

A hierarchical term is defined by referencing its parent by label; thus, the parent's indexing data can easily be obtained and prefixed with the level separator at the start of the child's data.

With the old glossary package, a term had only an associated name, description and sort value, but the new glossaries package provides extra fields, such as an associated symbol or plural form. Unfortunately, when developing the new package I was still thinking in terms of the old package that needed to include the name and description in the indexing information so that it could be displayed in the glossary. Early versions of the glossaries package continued this practice and the 'actual' part of the indexing information included the name, description and symbol, written to the indexing file in the form

`\glossaryentryfield{`⟨*label*⟩`}{`⟨*name*⟩`}{`⟨*description*⟩`}{`⟨*symbol*⟩`}`

This caused a number of problems. First, the name, description and symbol values all had to be parsed for indexing special characters, which added to the document build processing time (especially for long descriptions). Second, long descriptions could cause the indexing information to exceed `makeindex`'s buffer.

The package settings can allow for expansion to occur when terms are defined (for example, if terms are defined in a programmatic context that uses scratch variables that need expanding). This can lead to robust internal commands appearing in the indexing information. To simplify the problem of trying to escape all the `@` characters, the glossaries package uses the question mark character (`?`) as the actual character instead.

While it was necessary with the glossary package to write all this information to the indexing file, it is no longer necessary with the glossaries package as the name, description and symbol can all now be accessed by referencing the corresponding field associated with the term's identifying label. Therefore, newer versions now simply use

`\glossentry{`⟨*label*⟩`}`

for the actual text. Hierarchical entries use

`\subglossentry{`⟨*level*⟩`}{`⟨*label*⟩`}`

for the actual text, where ⟨*level*⟩ is the hierarchical level that's calculated when the term is defined. (This information may be of use to glossary styles that support hierarchical entries.) It's now quicker to construct the indexing information and only the sort value and label need checking for special characters.

For example,

```
\documentclass{report}
\usepackage[colorlinks]{hyperref}
\usepackage[symbols,style=treegroup]{glossaries}
\makeglossaries

\newglossaryentry{waterbird}% label
{name={waterbird},
description={bird that lives in or near water}}

\newglossaryentry{duck}% label
{name={duck},
parent={waterbird},
description={a waterbird with webbed feet}}

\newglossaryentry{goose}% label
{name={goose},
plural={geese},
parent={waterbird},
description={a waterbird with a long neck}}

\newglossaryentry{fact}% label
{name={\ensuremath{n!}},
 description={$n$ factorial},
 sort={n!},
 type=symbols
}
\begin{document}
\chapter{Singular}
\Gls{duck} and \gls{goose}.
\chapter{Plural}
\Glspl{duck} and \glspl{goose}.
\chapter{Other}
\begin{equation}
\gls[counter=equation]{fact} = n \times (n-1)!
\end{equation}
\printglossaries
\end{document}
```

This uses `\makeglossaries` (provided by glossaries), rather than `\makeglossary`, as it's not simply opening one associated file. The glossaries package supports multiple glossaries and all associated files are opened by this command as well as the custom style file for use by the indexing application. In this case, the document has two glossaries: the default main glossary and the symbols list (created with the `symbols` package option).

The glossaries are output using the command `\printglossaries`. This is a shortcut to iterate over all defined glossaries, calling for each

`\printglossary[type=`⟨*label*⟩`]`

where ⟨*label*⟩ identifies the required glossary. It's this command that inputs the file generated by the indexing application. A style is needed that supports hierarchical entries. In this example, I've chosen the `tree` style in the package options but the style can also be set within the optional argument of `\printglossary`. (A list of all styles with example output can be viewed at the glossaries gallery [15].)

As shown here: the hyperref package must be loaded before glossaries. This is an exception to the general rule that hyperref should be loaded last.

The commands `\gls`, `\Gls`, `\glspl` and `\Glspl` all reference a term by its label and simultaneously display and index the term. The variations provide a way of displaying the plural form (`\glspl` and `\Glspl`) and to convert the first letter to upper case (`\Gls` and `\Glspl`). In this case, the 'waterbird' entry isn't explicitly indexed in the document but it's included in the indexing information for its child entries 'duck' and 'goose', which are indexed.

The above example creates two indexing files with extensions `.glo` (for the default glossary) and `.slo` (for the symbols list), that both use `makeindex` syntax. Each file contains the indexing information for a particular glossary. Both files require the `.ist` style file that's also created during the document build.

The lines are quite long but are all in the form (line breaks for clarity)

`\glossaryentry`
`{`⟨*data*⟩`|`⟨*encap*⟩`}`
`{`⟨*location*⟩`}`

The ⟨*encap*⟩ and ⟨*location*⟩ information can vary with each indexing instance but the ⟨*data*⟩ part is constant for each term, and it's this part that's created when the term is defined.

When the 'waterbird' term is defined, the ⟨*data*⟩ part is determined to be

```
waterbird?\glossentry{waterbird}
```

The sort part here is `waterbird` and the actual part is `\glossentry{waterbird}`. This is stored internally and accessed when the child entries are defined. For example, when the 'duck' entry is defined, its ⟨*data*⟩ information is set to (line break for clarity and not included in ⟨*data*⟩)

```
waterbird?\glossentry{waterbird}!
duck?\subglossentry{1}{duck}
```

The hierarchical level numbering starts with 0 for top-level entries, so the duck entry has the level set to 1 since it has a parent but no grandparent.

The factorial example has this ⟨*data*⟩ part set:

```
n"!?\glossentry{fact}
```

Note that the special character occurring in the sort value has been escaped. This has to be done only once, when the entry is defined.

The location number defaults to the page number but may be changed, as in the reference to the `fact` entry, which switches to the `equation` counter:

```
\begin{equation}
\gls[counter=equation]{fact} = n \times (n-1)!
\end{equation}
```

Since the report class is in use, this is in the form ⟨*chapter*⟩.⟨*equation*⟩ (`3.1` in this case).

Location formats only have limited support with `makeindex`, which requires a bare number (0, 1, 2, ...), Roman numeral (i, ii, iii, ... or I, II, III, ...), basic Latin letter (a, ..., z or A, ..., Z) or a simple composite that combines these forms with a given separator (such as A-4 or 3.1). The separator must be consistent, so you can't have a mixture of, say, A:i and B-2 and 3.4.

In this case, the separator is a period or full stop character, which is the default setting in the custom style file created by `\makeglossaries`, so `makeindex` will accept '3.1' as a valid location.

Unfortunately, the glossary style may need to know which counter generated each location. This is especially true if the hyperref package is in use and the location numbers needs to link back to the corresponding place in the document. The hyperlink information can't be included in the indexed location as it will be rejected as invalid by `makeindex`. The only other part of the indexing information that can vary without `makeindex` treating the same term as two separate entries is within the encap, so the glossaries package actually writes the encap as

`setentrycounter[`⟨*h-prefix*⟩`]{`⟨*counter*⟩`}\`⟨*csname*⟩

where ⟨*counter*⟩ is the counter name and ⟨*csname*⟩ is the name of the actual encapsulating command. This defaults to `glsnumberformat` but may be changed in the optional argument of commands like `\gls`.

The first example document at the start of this article demonstrated `makeindex`'s implicit range formation, where the location list for the 'goose' entry (which was indexed on pages 1, 2 and 3) was compressed into `1--3`. This compression can only occur if the encap is identical for each of the indexing instances within the range.

The hypertarget will necessarily change for each non-identical indexed location. This means that if the actual target is included in the encap it will interfere with the range formation. Instead, only a prefix is stored (⟨*h-prefix*⟩) which can be used to reconstruct the hypertarget. This assumes that `\theH`⟨*counter*⟩ is defined in the form ⟨*h-prefix*⟩`\the`⟨*counter*⟩. Now the encap will be identical for identical values of ⟨*h-prefix*⟩. If the hypertarget can't be reconstructed from the location by simply inserting a prefix then it's not possible to have hyperlinked locations with this indexing method.

In the above example, the report class has been loaded along with hyperref so `\theHequation` is defined as

```
\theHsection.\arabic{equation}
```

This means that the indexing of the term in equation 3.1 occurs when `\theHequation` expands to 3.0.1 (the section counter is 0) so ⟨*h-prefix*⟩ can't be obtained since there's no prefix that will make ⟨*prefix*⟩`3.1` equal to `3.0.1`. This results in a warning from the glossaries package:

```
Hyper target `3.0.1' can't be formed by
prefixing location `3.1'. You need to modify
the definition of \theHequation otherwise
you will get the warning: "`name{equation.3.1}'
has been referenced but does not exist"
```

(and hyperref does indeed generate that warning once the glossary files have been created). The only solution here is to either remove the location hyperlink or redefine `\theHequation` so that a prefix can be formed.

### 3.1 `xindy`

Although the glossaries package was originally designed for use with just `makeindex`, version 1.17 added support for `xindy`. It made sense to use `xindy`'s native format as it's more flexible; also, `texindy` only accepts the default `makeindex` special characters so it won't accept `?` as the actual character.

The default setting assumes the `makeindex` application will be used for backward compatibility. The `xindy` package option will switch to `xindy` syntax. Again the partial indexing data is constructed when each entry is defined, but now the special characters that need escaping are `"` and `\`.

The previous example can be converted to use `xindy` by modifying the package options:

```
\usepackage[symbols,style=treegroup,xindy]
  {glossaries}
```

The package also needs to know which counters (aside from the default `page` counter) will be used for locations. In our example, since one of the terms is indexed with the `equation` counter, this needs to be indicated:

```
\GlsAddXdyCounters{equation}
```

(The argument should be a comma-separated list if you are indexing other counters as well.)

As with `makeindex`, it's not straightforward to add the information needed to convert the location into a hyperlink. Now the prefix and location are provided using

`:locref "{`⟨*h-prefix*⟩`}{`⟨*location*⟩`}"`

and the counter and encapsulating format are merged into the attribute value:

`:attr "⟨counter⟩⟨format⟩"`

This is why with the `xindy` package option set it's necessary to specify which non-default counters and formats you want to use — so that corresponding commands can be provided.

Unlike `makeindex`, which will only accept very specific types of numbering, with `xindy` you can have your own custom numbering scheme, provided that you define a location class that specifies the syntax. This is obviously a far more flexible approach but the downside is a far greater chance that the location might include `xindy`'s special characters that will need escaping, and now the escaping must be done every time an entry is indexed, not just when the entry is defined.

Suppose for example, I have a little package (that loads etoolbox [7] and tikzducks [11]) that provides the robust command `\ducknumber{⟨n⟩}`to display ⟨n⟩ little ducks in a row,

```
\newcount\duckctr
\newrobustcmd{\ducknumber}[1]{%
 \ifnum#1>0\relax
   \duckctr=0\relax
   \loop
    \advance\duckctr by 1\relax
    \tikz[scale=0.3]{\duck;}%
   \ifnum\duckctr<#1
   \repeat
 \fi
}
```

It also provides `\duckvalue{⟨counter⟩}` if the value needs to be obtained from a counter:

```
\newcommand*{\duckvalue}[1]{%
 \ducknumber{\value{#1}}}
```

Now let's suppose I want the page numbering in my document to be represented by ducks:

```
\renewcommand{\thepage}{\duckvalue{page}}
```

so, for example, on page 5, five little ducks are displayed in the footer. Now let's suppose that I index a term on this page. The location will expand to

```
\ducknumber{5}
```

This would be rejected as invalid by `makeindex`, but what about `xindy`? With an appropriate location class `xindy` would accept this, but it would interpret `\d` as the literal character 'd'. The resulting code it would write to the designated output file would be

```
ducknumber{5}
```

so you'd end up with 'ducknumber5' typeset in your document.

The backslash must be escaped but there's a conflict between expansion and TEX's asynchronous output routine. With the glossaries package, the location is obtained by expanding the command `\theglsentrycounter`, and the corresponding hypertarget value (if supported) is obtained by expanding `\theHglsentrycounter`. These two commands can be fully expanded when trying to determine the prefix. If the value of the `page` counter is currently wrong, then it's equally wrong for both values and it should still be possible to obtain the prefix.

When it comes to the actual task of preparing the location so that it's in a suitable format for `xindy`, there's no sense in converting `\theglsentrycounter` into `\\theglsentrycounter` as clearly there's no way for `xindy` to extract the page number from this. On the other hand, if `\theglsentrycounter` is fully expanded (and then detokenized and escaped), the page number could end up incorrect if it occurs across a page break.

The normal way around this problem (used by `\protected@write`) is to locally let `\thepage` to `\relax` so that it isn't expanded until the actual write operation is performed, but if this method is used the location will end up as `\\thepage` which will prevent `xindy` from obtaining the correct value.

It's necessary for `\thepage` to be expanded before the write operation in order to escape the special characters but at the same time, the actual value of `\c@page` shouldn't be expanded until the write operation is actually performed.

Essentially, for the duck numbering example, on page 5 `\thepage` needs to be converted into

```
\\ducknumber{\the\c@page}
```

where `\\` are two literal (catcode 12) characters and `\the\c@page` is left to expand when the write operation is performed.

The glossaries package gets around this problem with a nasty hack that locally redefines some commands. For example, `\@alph\c@page` expands to `\gls@alphpage`. This command is skipped when the special characters are escaped but expands to the original definition of `\@alph\c@page` when the write operation is actually performed.

This action is only performed when the `page` counter is being used for the location. Other counters will need to be expanded immediately to ensure that they are the correct value.

As this hack can cause problems in some contexts, if you know that your locations will never expand to any content that contains `xindy` special characters, then it's best to switch off this behaviour with the package option `esclocations=false`.

This is an inherent problem when converting from one syntax (LATEX in this case) to another

(`xindy` or `makeindex`). Each syntax has its own set of special characters (required to mark up or delimit data) that may need to be interpreted literally.

### 3.2 Using TEX to sort and collate

Some users who aren't familiar with command line tools have difficulty integrating them into the document build and prefer a TEX-only solution that doesn't require them. In general, it's best to use tools for the specific task they were designed for. Indexing applications are designed for sorting and collating data. TEX is designed for typesetting. Each tool is optimized for its own particular intended purpose. It is possible to sort and collate in TEX but it's much less efficient than using a custom indexing application. However, for small documents it may suit some users to have everything done within TEX, so version 4.04 of the glossaries package introduced a TEX-only method.

The example document given on page 53 can be converted to use this method simply by replacing `\makeglossaries` with `\makenoidxglossaries` and `\printglossaries` with `\printnoidxglossaries`. As with `\printglossaries`, this is a shortcut command that iterates over all defined glossaries, doing

`\printnoidxglossary[type=`⟨*label*⟩`]`

In this case, the command doesn't input a file but sorts the list of entry labels and iterates over them to display the information using the required glossary style. The label list only includes those entries that have been indexed in the previous LATEX run. This information is obtained from the `.aux` file. Each time an entry is indexed using commands like `\gls`, a line is written to the `.aux` file in the form

`\gls@reference{`⟨*type*⟩`}{`⟨*label*⟩`}{`⟨*location*⟩`}`

where ⟨*type*⟩ identifies the glossary, ⟨*label*⟩ identifies the entry and ⟨*location*⟩ is in the form

`\glsnoidxdisplayloc{`⟨*h-prefix*⟩`}{`⟨*counter*⟩`}{`⟨*encap*⟩`}{`⟨*number*⟩`}`

This has the advantage that there is no conversion from one syntax to another and there's no restriction on ⟨*number*⟩ (as long as it's valid LATEX code). The disadvantages are that there's no range support and sorting is slow and uses character code comparisons. (See my earlier *TUGboat* article comparing indexing methods [13].)

With this method, each entry has an associated internal field labelled `loclist`. When the `.aux` file is parsed, each location is added to this field using one of etoolbox's internal list commands. This list is iterated over in order to display the locations.

## 4 The glossaries-extra package

The glossaries-extra package [18] was created in 2015 as a compromise between the conflicting requirements of users who wanted new features and users who complained that the glossaries package took a long time to load (because it had so many features). New features, especially those that require additional packages, necessarily add to the package load time.

The glossaries-extra package automatically loads the base glossaries package, but there are some differences in the default settings, the most noticeable being the abbreviation handling. The base package only allows one abbreviation style to be used throughout the document. The extension package defines a completely different mechanism for handling abbreviations that allows multiple styles within the same document.

As with the base package, the default indexing application is still assumed to be `makeindex` but the extension package provides two extra methods (although from LATEX's point of view they both use the same essential code).

The new command

`\printunsrtglossary[`⟨*options*⟩`]`

works fairly similarly to `\printnoidxglossary`, in that it iterates over a list of labels, but the list contains all the labels defined in the given glossary (rather than just those that have been indexed) and no sorting is performed by TEX.

As with the other methods, there's a shortcut command that iterates over all glossaries:

`\printunsrtglossaries`

For example,

```
\documentclass{report}
\usepackage[colorlinks]{hyperref}
\usepackage[symbols,style=treegroup]
  {glossaries-extra}

\newglossaryentry{waterbird}{name={waterbird},
 description={bird that lives in or near water}}

\newglossaryentry{duck}{name={duck},
 parent={waterbird},
 description={a waterbird with webbed feet}}

\newglossaryentry{goose}{name={goose},
 plural={geese},
 parent={waterbird},
 description={a waterbird with a long neck}}

\newglossaryentry{fact}{name={\ensuremath{n!}},
 description={$n$ factorial},
 sort={n!},
 type=symbols
}
```

```
\begin{document}
\printunsrtglossaries
\end{document}
```

No indexing is performed in this document. With the other methods provided by the base package this would result in empty glossaries, but with this method all defined entries are shown (and only one LaTeX call is required to display the list). The 'goose' entry appears after 'duck' but only because 'goose' was defined after 'duck'.

The glossary style I've chosen here (`treegroup`) shows the letter group headings. This is something that's usually determined by the indexing applications according to the first character of the sort value. The heading information is then written to indexing output file (read by `\printglossary`) at the start of a new letter block.

The 'noidx' method checks the first letter of the sort value at the start of each iteration, and if it's different from the previous iteration a new heading is inserted. The 'unsrt' method also does this unless the `group` key has been defined, in which case the letter group label is obtained from the corresponding field (if it's set).

This letter group formation can lead to strange results if the entries aren't defined in alphabetical order [16]. For example,

```
\documentclass{article}
\usepackage[style=treegroup]{glossaries-extra}

\newglossaryentry{ant}{name={ant},
 description={small insect}}

\newglossaryentry{aardvark}{name={aardvark},
 description={animal that eats ants}}

\newglossaryentry{duck}{name={duck},
 description={waterbird with webbed feet}}

\newglossaryentry{antelope}{name={antelope},
 description={deer-like animal}}

\begin{document}
\printunsrtglossaries
\end{document}
```

This produces the document shown in figure 1. (The vertical spacing below the letter headings is too large, but that is the default result; the point here is the undesired second 'A' group.)

If the `group` key has been defined but not explicitly set then an empty headerless group is assumed. If the above example is modified so that it defines the `group` key:

```
\glsaddstoragekey{group}{}{\grouplabel}
```

## Glossary

**A**

**ant** small insect
**aardvark** animal that eats ants

**D**

**duck** waterbird with webbed feet

**A**

**antelope** deer-like animal

**Figure 1**: Example glossary with letter groups

## Glossary

**ant** small insect
**aardvark** animal that eats ants
**duck** waterbird with webbed feet
**antelope** deer-like animal

**Figure 2**: Example glossary with empty group

but without modifying the entry definitions to set this key then no letter groups are formed (see figure 2).

The `record` package option automatically defines the `group` key. Each group value should be a label. The corresponding title can be set with

`\glsxtrsetgrouptitle{`⟨*label*⟩`}{`⟨*title*⟩`}`

For example,

```
\documentclass{article}
\usepackage[style=treegroup,record]
{glossaries-extra}
\glsxtrsetgrouptitle{antrelated}{Ants and
Ant-Eaters}
\glsxtrsetgrouptitle{waterbirds}{Waterbirds}
\glsxtrsetgrouptitle{deerlike}{Deer-Like}
\newglossaryentry{ant}{name={ant},
 group={antrelated},
 description={small insect}}
\newglossaryentry{aardvark}{name={aardvark},
 group={antrelated},
 description={animal that eats ants}}
\newglossaryentry{duck}{name={duck},
 group={waterbirds},
 description={waterbird with webbed feet}}
\newglossaryentry{antelope}{name={antelope},
 group={deerlike},
 description={deer-like animal}}
\begin{document}
\printunsrtglossaries
\end{document}
```

# Glossary

**Ants and Ant-Eaters**

**ant** small insect
**aardvark** animal that eats ants

**Waterbirds**

**duck** waterbird with webbed feet

**Deer-Like**

**antelope** deer-like animal

**Figure 3**: Example glossary with custom groups

This now produces the glossary shown in figure 3. (Alternatively, use the `parent` key for a hierarchical structure or the `type` key to separate the logical blocks into different glossaries [16].)

`\printunsrtglossary` uses an iteration handler that supports the `loclist` internal field used with the 'noidx' method. If this field is set, the locations will be displayed but, as with the 'noidx' method, no ranges are formed and the elements of the `loclist` field must conform to a specific syntax. However, the handler will first check if the `location` field is set. If it is, that will be used instead.

The `location` key isn't provided by default but is defined by the `record` option, so locations can also be provided when a term is defined. For example,

```
\newglossaryentry{ant}{name={ant},
 group={antrelated},
 location={1, 4--5, 8},
 description={small insect}}
```

This may seem cumbersome to do manually but it's the underlying method used by `bib2gls` [17].

## 5 Glossaries and `.bib`: `bib2gls`

Some years ago I was asked if it was possible to provided a GUI (graphical user interface) application to manage files containing many entry definitions. This article has only mentioned defining entries with `\newglossaryentry` but there are other ways of defining terms with the glossaries package (and some additional commands provided with glossaries-extra). I already have several GUI applications that are quite time-consuming to develop and maintain, and the proposed task seemed far too complex, so I declined.

More recently, a question was posted on StackExchange [10] asking if it was possible to store terms in a `.bib` file, which could be managed in an application such as JabRef [3], and then converted into a `.tex` file containing commands such as `\newglossaryentry`. This was a much better proposition as the graphical task could be dealt with by JabRef and the conversion tool could be a command line application.

I added the `record` option and the commands like `\printunsrtglossary` to glossaries-extra to assist this tool. The `record` option not only creates new keys (`group` and `location`) but also makes references to undefined entries trigger warnings rather than errors. This is necessary since the entries won't be defined on the first LaTeX call. The option also changes the indexing behaviour. As with the 'noidx' method, the indexing information is written to the `.aux` file so that the new tool could find out which entries are required and their locations in the document. In this case, the `.aux` entry is in the form

`\glsxtr@record{`⟨*label*⟩`}{`⟨*h-prefix*⟩`}{`⟨*counter*⟩`}{`⟨*encap*⟩`}{`⟨*location*⟩`}`

As with the 'noidx' method there is no conversion from one syntax to another when the indexing takes place, so there is no need to worry about escaping special indexing characters.

It later occurred to me that, without the constraints of the `makeindex` or `xindy` formats, it's possible to save the hypertarget so that it doesn't have to be reconstructed from ⟨*h-prefix*⟩ and ⟨*location*⟩. In glossaries-extra version 1.37 I added the package option `record=nameref`, which writes more detailed indexing information to the `.aux` file (and support for this new form was added to `bib2gls` v1.8). This means that the earlier `makeindex` example on page 53 can be rewritten in such a way that the equation location now has a valid hyperlink.

TeX syntax can be quite hard to parse programmatically. Regular expressions don't always work. I have a number of applications that are related to TeX in some way and need to parse either complete documents or code fragments. The most complicated of these was a Java GUI application used to assist production editors. The document code submitted by authors often contained problematic code that needed fixing, which was both tedious and time-consuming, so I tried to develop a system that parsed the original source provided by the authors and created new files with the appropriate patches and comments alerting the production editors of a potential problem, where necessary. The files were also flattened (that is, `\input` was replaced by the contents of the referenced file) to reduce clutter.

I realised that the TeX parsing code used in this application would also be useful in some of my other Java applications so, rather than producing unnecessary duplication, I split the code off into a separate library, `texparserlib.jar` [14]. Rather than testing the code in big GUI applications that take a long

time to set up and run, I added a small application called `texparserapp.jar` to the `texparser` repository together with a selection of sample files to test the library.

The production editor GUI application not only needed to parse the `.tex` and `.bib` files supplied by the authors but also needed to gather information from `.aux` files. Some of this information is displayed in the graphical interface and it looks better if LaTeX commands like `\'` or `\c` are converted to Unicode when showing author names. It used to also be a requirement for production editors to produce HTML files containing the abstract. I originally used TeX4ht for this but an update caused a conflict, and since only the abstract needed converting and MathJax could be used for any mathematical content, I decided that the TeX parser code should not only provide LaTeX to LaTeX methods but also LaTeX to HTML — with the caveat that the conversion to HTML was not intended for complete documents but for code fragments supplemented by information obtained from the `.aux` file.

The GUI application was used not only to prepare workshop proceedings but also to prepare a related series of books that contained reprints. Since writing the TeX parser library the requirements for the proceedings have changed, which make the production editing task easier, and the publisher for the related series has also changed and the new publisher provides their own templates. The application has now largely become redundant although it can still be used to prepare volumes for the proceedings.

Since I already had this library that was designed to obtain information from `.aux` and `.bib` files, it made sense to use it for my new tool. This meant that the new tool also had to be in Java. The library methods that can convert LaTeX code fragments to HTML provide a useful way of obtaining an appropriate sort value from the `name` field as accent commands can be converted to Unicode characters. Command definitions provided in `@preamble` can also be interpreted (provided they aren't too complex). Any HTML markup is stripped and leading and trailing white space is trimmed. This means that there should rarely be any need to set the `sort` field when defining an entry.

Sorting can be performed according to a valid language tag, such as `en` (English) or `en-GB` (British English) or `de-CH-1996` (Swiss German new orthography). Java 8 has support for the Unicode Common Locale Data Repository (CLDR) which provides collation rules, so `bib2gls` can support more languages than `xindy` (although, unlike `xindy`, it doesn't support Klingon).

There are other sort methods available as well, including sorting according to Unicode value (case-sensitive or case-insensitive) or sorting numerically (assuming the sort values are numbers) or sorting according to use in the document (determined by the ordering of the indexing information contained within the `.aux` file).

For example, suppose the file `entries.bib` contains the following:

```
% Encoding: UTF-8
@entry{waterbird,
  name={waterbird},
  description={bird that lives in or near water}}
@entry{goose,
  name={goose},
  parent={waterbird},
  description={waterbird with a long neck}}
@entry{duck,
  name={duck},
  parent={waterbird},
  description={waterbird with webbed feet}}
```

and suppose the file `symbols.bib` contains

```
% Encoding: UTF-8
@preamble{"\providecommand{\factorial}[1]{#1!}
\providecommand{\veclength}[1]{|#1|}"}
@symbol{nfact,
  name={\ensuremath{\factorial{n}}},
  description={$n$ factorial}}
@symbol{lenx,
  name={\ensuremath{\veclength{\vec{x}}}},
  description={length of $\vec{x}$}}
```

The document code

```
\documentclass{report}
\usepackage[colorlinks]{hyperref}
\usepackage[symbols,style=treegroup,
            record=nameref]
  {glossaries-extra}
\GlsXtrLoadResources[
  src=entries,% entries.bib
  sort=en-GB]
\GlsXtrLoadResources[
  src=symbols, % symbols.bib
  type=symbols,% glossary
  sort=letter-nocase]

\begin{document}
\chapter{Singular}
\Gls{duck} and \gls{goose}.
\chapter{Plural}
\Glspl{duck} and \glspl{goose}.
\chapter{Other}
\begin{equation}
\gls[counter=equation]{nfact} = n \times (n-1)!
\end{equation}
The length of $\vec{x}$ is \gls{lenx}.\par
\printunsrtglossaries
\end{document}
```

Unlike the `makeindex` and `xindy` methods, which require one call per glossary, with this approach only one `bib2gls` call is required, regardless of the number of glossaries. For example, if the document code is in `myDoc.tex`, then the build process is

```
pdflatex myDoc
bib2gls myDoc
pdflatex myDoc
```

Letter groups are not formed by default. To get them, specify the `-g` switch:

```
bib2gls -g myDoc
```

`bib2gls` creates one `.glstex` output file per instance of `\GlsXtrLoadResources`, but you don't necessarily need one `\GlsXtrLoadResources` per glossary. You may be able to process multiple glossaries within one instance of this command, or a single glossary may require multiple instances.

The `.glstex` file contains the glossary definitions (using provided wrapper commands for greater flexibility) in the order obtained from the provided sort method. In the above example, the entries in the first `.glstex` file are defined in the order obtained by sorting the values according to the `en-GB` rule. The entries in the second `.glstex` file are defined in the order obtained by sorting the values according to the `letter-nocase` rule (that is, case-insensitive Unicode order).

If the `sort` key isn't provided (which it generally isn't), its value is taken from the designated fallback field. In the case of `@entry` this is the `name` field and in the case of `@symbol` this is the entry's *label*. So in the above example, the symbols are sorted as first '`lenx`' and second '`nfact`'.

The fallback field used for `@symbol` entries can be changed. For example, to switch to the `name` field:

```
\GlsXtrLoadResources[
  src=symbols,
  type=symbols,
  symbol-sort-fallback=name,
  sort=letter-nocase
]
```

Since the `name` field contains commands, the TEX parser library is used to interpret them. The transcript file (`.glg`) shows the results of the conversion. The `nfact` entry ends up with just two characters, '`n!`' but the `lenx` entry ends up with four characters: vertical bar (Unicode 0x7C), lower case 'x' (Unicode 0x78), combining right arrow above (Unicode 0x20D7) and vertical bar (Unicode 0x7C). The order is now: $n!$ (`nfact`), $|\vec{x}|$ (`lenx`).

## References


[1] Claudio Beccari and Enrico Gregorio. The imakeidx package, 2018. `ctan.org/pkg/imakeidx`.

[2] P. Happel. The lipsum package, 2019. `ctan.org/pkg/lipsum`.

[3] JabRef: Graphical frontend to manage BibTEX databases, 2018. `jabref.org`.

[4] R. Kehr and J. Schrod. `xindy`: A general-purpose index processor, 2018. `ctan.org/pkg/xindy`.

[5] The LaTeX Team. The makeidx package, 2014. `ctan.org/pkg/makeidx`.

[6] The LaTeX Team, F. Mittelbach, and A. Jeffrey. The inputenc package, 2018. `ctan.org/pkg/inputenc`.

[7] P. Lehman and J. Wright. The etoolbox package, 2018. `ctan.org/pkg/etoolbox`.

[8] L. Madsen and P. R. Wilson. The memoir class, 2018. `ctan.org/pkg/memoir`.

[9] L. Netherton, C. V. Radhakrishnan, et al. The nomencl package, 2019. `ctan.org/pkg/nomencl`.

[10] The Pompitous of Love. Is there a program for managing glossary tags?, 2016. `tex.stackexchange.com/questions/342544`.

[11] samcarter. The tikzducks package, 2018. `ctan.org/pkg/tikzducks`.

[12] N. Talbot. The glossary package, 2006. `ctan.org/pkg/glossary`.

[13] N. Talbot. Testing indexes: `testidx.sty`. *TUGboat* 38(3):377–399, 2017. `tug.org/TUGboat/tb38-3/tb120talbot.pdf`.

[14] N. Talbot. `texparserlib.jar`: A Java library for parsing (LA)TEX files, 2018. `github.com/nlct/texparser`.

[15] N. Talbot. Gallery of all styles provided by the glossaries package, 2019. `dickimaw-books.com/gallery/glossaries-styles`.

[16] N. Talbot. Logical glossary divisions (type vs group vs parent), 2019. `dickimaw-books.com/gallery/logicialdivisions.shtml`.

[17] N. Talbot. bib2gls: Command line application to convert `.bib` files to `glossaries-extra.sty` resource files, 2019. `ctan.org/pkg/bib2gls`.

[18] N. Talbot. The glossaries-extra package, 2019. `ctan.org/pkg/glossaries-extra`.

[19] N. Talbot. The glossaries package, 2019. `ctan.org/pkg/glossaries`.

[20] H. Voß. `xindex`: Unicode compatible index generation, 2019. `ctan.org/pkg/xindex`.



◇ Nicola L. C. Talbot
School of Computing Sciences
University of East Anglia
Norwich Research Park
Norwich NR4 7TJ
United Kingdom
`N.Talbot (at) uea dot ac dot uk`
`http://www.dickimaw-books.com`