Spindex — Indexing with Special Characters

Laurence Finston

1. Introduction

Books in the field of philology, among others, often contain many special characters: letters like ð and þ, ligatures like æ and œ, phonetic symbols like ɧ and ŋ and even more unusual ones. If these books require indexes, words with these special characters must be sorted alphabetically. However, to the best of my knowledge, the available indexing programs are only able to sort words in English, or at best in a handful of European languages. Spindex (for "Special Index") is a package that can sort arbitrary special characters alphabetically. It can also be adapted for use with languages that do not use the Latin alphabet.

TEX has no built-in routines for alphabetical sorting, so it is necessary to use the sorting routines belonging to the operating system, a programming language, or another program. Spindex is a combination of TEX macros in the file `spindex.tex` and a program written in Common Lisp in the file `spindex.lsp`. It is intended for use with `plain` TEX, but it is possible (with some difficulty) to use it with LATEX, too.

The first section of this article explains Spindex for the user who just wants to use it for making an index, and doesn't care about how it works. The following section explains some of the principles behind the TEX macros and the Lisp program.

2. Using Spindex

In order to use Spindex, the file `spindex.lsp`, containing the Lisp program, must be in your working directory, and `spindex.tex`, containing the definition of the TEX macro `\indexentry` and additional TEX code, must be either in your working directory or in a directory in TEX's load path as defined in your `texmf.cnf` file (if you don't know what this is, ask your local TEX wizard, or just put the file in your working directory). Your input file must include the line `\input spindex` before you use `\indexentry` for the first time.

When you use `\indexentry`, it causes TEX to write a file of Lisp code called `entries.lsp`. When TEX is done with your input file, you invoke the Lisp interpreter and give `spindex.lsp` to it as input. If you're using the Gnu Lisp interpreter, which is what I use, you type

```
gcl<spindex.lsp
```

The program in `spindex.lsp` loads `entries.lsp` and creates a TEX file containing the index called `index.tex`. Now you can run TEX on `index.tex`. Below I describe how to automate this process and include `index.tex` in your original input file.

2.1. The macro `\indexentry`. An index entry is created using `\indexentry`, which has six arguments, all of which except for `#1` may be empty (i.e., `{}`). TEX does not have true optional arguments, but it is possible to define macros so that they check whether an argument is empty or not, simulating the effect of optional arguments. The consequence of this is that six sets of braces must always follow `\indexentry` whether there's anything in them or not.

The first argument, `#1`, is `name`, which is used for alphabetizing the entries, and it is usually what is written to the index. It is the only required argument. An occurrence of `\indexentry` with only the `name` argument is the simplest possible kind. For example,

```
\indexentry{nouns}{}{}{}{}{}
```

on page 54

⟹

nouns . . . . 54.

In most cases, `\indexentry` will be typed into the input file directly after the word or phrase that it refers to:

```
a noun\indexentry{nouns}{}{}{}{}{} is
a word that refers ...
```

produces the following output:

a noun is a word that refers ...

Putting `\indexentry` directly after the word or phrase it refers to prevents a page break between them, which would cause an incorrect page number to appear in the index. However, `\indexentry` can also stand alone, as in the examples below. Note that `\indexentry` has no effect on the output file. All it does is write information to `entries.lsp`, which is used for making the index. However, I use a conditional called `\ifdraft` for editing purposes that makes `\indexentry` write a marginal hack whenever `\drafttrue`, i.e., whenever `\ifdraft` expands to `\iftrue`.

a noun is a word that refers ... `*nouns*`

For the final draft, I set `\draftfalse`, and the marginal hacks disappear.

Argument `#2` is `text`, and will usually be empty. If it's not empty, it's what's written to the index, but the entry is still alphabetized according to `name`.

```
\indexentry{A}{A (the letter A)}{}{}{}{}
```

$\Longrightarrow$

```
    A (the letter A) . . . . . . . . . . . . . 96.
```

but "␣(the letter A)" does not affect the alphabetization of the entry.

The `text` argument can also be used for putting comments into the index at a particular place.

```
\indexentry{nouns}{*Comment*}{}{}{}{}
\indexentry{prepositions}{}{}{}{}{}
\indexentry{adverbs}{}{}{}{}{}
```

$\Longrightarrow$

```
    adverbs . . . . . . . . . . . . . . . . . 87.
    *Comment* . . . . . . . . . . . . . . . . 87.
    prepositions . . . . . . . . . . . . . . . 87.
```

Note that "*Comment*" is put where "nouns" would go. The `text` argument only has an effect when an entry is created. After that it's ignored, so if you want a `text`, you must make sure it's set the first time. It would be easy to change this, but I felt that it was safer to program it this way. Most of the time `text` will not be used. It is only for special cases like these.

The best way to set `text` is to use dummy entries at the beginning of your input file where the page number is suppressed using argument `#3`. A comment, like the one in the previous example, also shouldn't have a page number and leaders attached. Suppressing the page number can also be useful for editing, when you're not sure whether to include a particular occurrence of an entry in the index. It doesn't matter what appears in `#3`; if it's non-empty, this occurrence of `\indexentry` will not cause the current page number to be added to the list of page numbers for this entry.

```
\indexentry{verbs}{}{np}{}{}{}
```

$\Longrightarrow$

```
    verbs
```

I like to use "np" (for "no page") in `#3`, but it can be anything within reason.[1] If an entry has no page numbers, no leaders are printed. Suppressing the page number in one invocation of `\indexentry` doesn't affect another invocation on the same page.

```
\indexentry{verbs}{}{np}{}{}{}
\indexentry{verbs}{}{}{}{}{}
```

$\Longrightarrow$

```
    verbs . . . . . . . . . . . . . . . . 123.
```

Argument `#4` is for a cross-reference. A cross-reference can be an arbitrary string or it can correspond to another entry. Here's an entry with a cross-reference that refers to an arbitrary string.

[1] An undefined control sequence or a macro with insufficient arguments will cause an error.

```
\indexentry{ships}{}{}{transport}{}{}
```

$\Longrightarrow$

```
    ships . . . . . . . . . . . . . . . . . . 75.
    See also: transport
```

Here's one with a cross-reference that refers to another entry.

```
\indexentry{ships}{}{}{boats}{}{}
\indexentry{boats}{}{}{}{}{}
```

$\Longrightarrow$

```
    boats . . . . . . . . . . . . . . . . . . 54.
    ships . . . . . . . . . . . . . . . . . . 54.
    See also: boats
```

Doesn't look much different, does it? But when a cross-reference refers to an entry that had a `text` (`#2`) argument, there is a difference.

```
\indexentry{boats}%
     {boats (lat. naves)}{}{}{}{}
\indexentry{ships}{}{}{boats}{}{}
```

$\Longrightarrow$

```
    boats (lat. naves) . . . . . . . . . . . 54.
    ships . . . . . . . . . . . . . . . . . . 54.
    See also: boats (lat. naves)
```

The cross-reference uses the `text` of an entry, if it exists. If there are multiple cross-references, they are alphabetized according to what is actually printed, i.e., the `texts`, if they exist, whereas the entries in the index are always alphabetized according to `name`.

Spindex allows 3 levels of nesting – headings, subheadings and subsubheadings. Argument `#5` is the `heading`, if the entry is a subheading or a subsubheading, and `#6` is the `subheading`, if the entry is a subsubheading. This is how you make a subheading entry:

```
\indexentry{transitive}{}{}{}{verbs}{}
```

$\Longrightarrow$

```
    verbs
        transitive . . . . . . . . . . . . . 54.
```

Here's one for a subsubheading entry:

```
\indexentry{active}{}{}{}{verbs}%
     {transitive}
```

$\Longrightarrow$

```
    verbs
        transitive
            active . . . . . . . . . . . . . 49.
```

A subheading or subsubheading entry will create an entry for its `heading` and/or `subheading`, if these don't already exist.

Here's a slightly tricky example (the line `\hbox{}\eject` is only there to end page 57).

```
\pageno=57
\indexentry{monosyllabic}{}{}{}%
     {adverbs}{temporal}
\hbox{}\eject
\indexentry{adverbs}{sbrevda}{}{}{}{}
```

$\Longrightarrow$

```
adverbs . . . . . . . . . . . . . . . . . 58.
    temporal
        monosyllabic . . . . . . . . . . . 57.
```

Do you see why "sbrevda" is not written to the index? The first invocation of `\indexentry`, for "adverbs, temporal, monosyllabic", caused entries for "adverbs" and "adverbs, temporal" to be created automatically. When `\indexentry` was invoked for "adverbs" in its own right, on page 58, the text argument was ignored, because the entry for "adverbs" had already been created. The best way to deal with this problem is by using a dummy entry, like this:

```
\pageno=1
\indexentry{adverbs}{sbrevda}{x}{}{}{}
\hbox{}\eject
\pageno=57
\indexentry{monosyllabic}{}{}{}%
     {adverbs}{temporal}
\hbox{}\eject
\indexentry{adverbs}{}{}{}{}{}
```

$\Longrightarrow$

```
sbrevda . . . . . . . . . . . . . . . . . 58.
    temporal
        monosyllabic . . . . . . . . . . . 57.
```

Here I use "x" to suppress the page number for the dummy entry. Subsequent invocations of `\indexentry` for "adverbs", like the one on page 58, needn't specify the `text` argument, since it's ignored.

Sometimes it might be desirable to put sub- or subheadings in order, but not in alphabetical order, if another ordering principle seems more appropriate.

```
\pageno=1
\indexentry{light}{light, visible}%
     {xxx}{}{}{}
\indexentry{wavelengths}{}{}{}%
     {light}{}
\hbox{}\eject
\indexentry{f}{violet}{}{}{light}%
     {wavelengths}
\hbox{}\eject
\indexentry{d}{green}{}{}{light}%
     {wavelengths}
\hbox{}\eject
\indexentry{b}{orange}{}{}{light}%
     {wavelengths}
\hbox{}\eject
\indexentry{c}{yellow}{}{}{light}%
     {wavelengths}
\hbox{}\eject
\indexentry{light}{}{}{}{}{}
\hbox{}\eject
\indexentry{a}{red}{}{}{light}%
     {wavelengths}
\hbox{}\eject
\indexentry{e}{blue}{}{}{light}%
     {wavelengths}
```

$\Longrightarrow$

```
light, visible . . . . . . . . . . . . . . 6.
    wavelengths . . . . . . . . . . . . . . 1.
        red . . . . . . . . . . . . . . . . 7.
        orange . . . . . . . . . . . . . . 4.
        yellow . . . . . . . . . . . . . . 5.
        green . . . . . . . . . . . . . . . 3.
        blue . . . . . . . . . . . . . . . 8.
        violet . . . . . . . . . . . . . . 2.
```

The subsubsubheadings (the colors of visible light) are alphabetized according to their `names`, i.e., "a", "b", "c", etc. This has the effect of putting them in order according to their wavelengths. Since there are no other subsubheadings, this causes no problems. Some items may have a conventional order that takes precedence over the alphabet.

```
\indexentry{Bears, the Three}{}{}%
     {Goldilocks}{}{}
  \indexentry{c}{Baby}{}{}%
     {Bears, the Three}{}
  \indexentry{c}{Baby}{}{}%
     {Bears, the Three}{}
  \indexentry{a}{Papa}{}{}%
     {Bears, the Three}{}
  \indexentry{b}{Mama}{}{}%
     {Bears, the Three}{}
```

$\Longrightarrow$

```
Bears, the Three . . . . . . . . . . . . 23.
See also: Goldilocks
    Papa . . . . . . . . . . . . . . . . 23.
    Mama . . . . . . . . . . . . . . . . 23.
    Baby . . . . . . . . . . . . . . . . 23.
```

Cross-references can refer to subheadings and subsubheadings, too:

```
\indexentry{schooners}{}{}{}{ships}%
     {sailing}
\indexentry{rigging}{}{}%
     {ships-sailing-schooners}%
     {}{}
```

⟹

```
rigging . . . . . . . . . . . . . . . . . 54.
See also: ships, sailing, schooners
ships
   sailing
      schooners . . . . . . . . . . . . . 54.
```

A cross-reference that refers to a heading entry simply uses the `name` argument from that entry.

```
\indexentry{carnivores}{}{}{mammals}{}{}
\indexentry{mammals}{}{}{}{}{}
```

⟹

```
carnivores . . . . . . . . . . . . . . . 25.
See also: mammals
mammals . . . . . . . . . . . . . . . . . 25.
```

It doesn't matter if the entry being used as a cross-reference has a `text`; you use the `name` anyway, but the `text` is printed to the index file.

```
\indexentry{fish}%
     {fish ({|it|pisces})}%
     {}{}{}{}
\indexentry{oceans}{}{}{fish}{}{}
```

⟹

```
fish (pisces) . . . . . . . . . . . . . 100.
oceans . . . . . . . . . . . . . . . . 100.
See also: fish (pisces)
```

When a subheading entry is used as a cross-reference, its `heading` and `name` arguments, separated by a hyphen, are used in the `cross-reference` argument of the entry that refers to it.

```
\indexentry{wolves}{}{}{bears-brown}{}{}
\indexentry{brown}{}{}{}{bears}{}
```

⟹

```
bears
   brown . . . . . . . . . . . . . . . . 371.
wolves . . . . . . . . . . . . . . . . . 371.
See also: bears, brown
```

When a subsubheading entry is used as a cross-reference, its `heading`, `subheading` and `name` arguments, separated by hyphens, are used in the `cross-reference` argument of the entry that refers to it.

```
\indexentry{American}%
     {American (Eastern)}%
     {}{}{bears}{brown}
\indexentry{wolves}{}{}%
     {bears-brown-American}{}{}
```

⟹

```
bears
   brown
      American (Eastern) . . . . . . . . 41.
wolves . . . . . . . . . . . . . . . . . 41.
See also: bears, brown, American (Eastern)
```

The syntax of cross-references is:

$\langle$cross-reference$\rangle \longrightarrow \langle$arbitrary string$\rangle$
   | $\langle$entry reference$\rangle$
$\langle$entry reference$\rangle \longrightarrow$ `heading`$\langle$suffix$\rangle$
$\langle$suffix$\rangle \longrightarrow \langle$empty$\rangle$ | `-subheading`
   | `-subheading-subsubheading`

Only one cross-reference can appear in any given occurrence of `\indexentry`.

Of course, subheading and subsubheading entries can themselves have cross-references, and their page numbers can be suppressed, too:

```
\indexentry{fish}{}{}{}{}{}
\indexentry{freshwater}{}{np}%
          {angling}{fish}{}
\indexentry{sturgeon}{}{}{caviar}%
          {fish}{freshwater}
```

⟹

```
fish . . . . . . . . . . . . . . . . . . 14.
   freshwater
   See angling
      sturgeon . . . . . . . . . . . . . 14.
      See also caviar
```

So far, all of the examples have been of entries with only one page number. Here's an example with multiple page numbers.

```
\pageno=5
\indexentry{trains}{}{}{}{}{}
\hbox{}\eject
\pageno=10
\indexentry{trains}{}{}{}{}{}
\hbox{}\eject
\pageno=15
\indexentry{trains}{}{}{}{}{}
\hbox{}\eject
\pageno=25
\indexentry{trains}{}{}{}{}{}
\hbox{}\eject
```

⟹

```
trains . . . . . . . . . . . 5, 10, 15, 25.
```

If an entry occurs on consecutive pages, page ranges are printed to the index instead of the individual page numbers.

```
trains
   diesel . . . . . . . . . . . . . . 62–98.
```

```
    electric . . . . . . . . . . . . 105–210.
    steam . . . . . . . . . . . . . . . 5–10.
```

Sometimes, the last number in a page range is abbreviated.

```
  ships . . . . . . . . . . . . . . . 104–23.
    sailing . . . . . . . . . . . . 1004–200.
    steam . . . . . . . . . . . . . 1239–98.
```

The rules for abbreviating page numbers are described on page 269.

If an entry has no page numbers, but it does have a cross-reference, "*See*" is printed instead of "*See also*".

```
\indexentry{adjectives}{}%
     {suppress page number!}%
     {pronouns}{}{}
```

⟹

adjectives
  See pronouns

If there are two cross-references, they are separated by "*and*", and if there are three or more, the last two are separated by "*and*" and the others are separated with a semi-colon.

```
\indexentry{schooners}{}{}{}{ships}{}
\indexentry{ships}{}{}{boats}{}{}
\indexentry{ships}{}{}{transport}{}{}
\indexentry{ships}{}{}{fishery}{}{}
\indexentry{rigging}{}{}%
          {ships-schooners}{}{}
\indexentry{rigging}{}{}{boats}{}{}
```

⟹

```
rigging . . . . . . . . . . . . . . . . . 54.
See also: boats and ships, schooners
ships . . . . . . . . . . . . . . . . . . 54.
See also boats; fishery and transport
      schooners . . . . . . . . . . . . . 54.
```

If an entry has no page numbers, no cross-references and no sub- or subsubheadings, it will be printed to the index, but `spindex.lsp` will issue a warning.

If more than one index is desired, for instance an index of names and an index of subjects, it would not be difficult to add a seventh argument to indicate to which index an entry belongs.

2.2. Coding special characters and macros. By now, you're probably convinced that Spindex has plenty of bells and whistles, but the capabilities described so far don't offer any significant advantage over the available indexing packages. The real power of Spindex is its ability to perform alphabetical sorting on arbitrary special characters.

It is not possible to use the normal coding for special characters, like `\dh` for ð, `\th` for þ, `\ae` for æ, and `\o` for ø, in `\indexentry`'s arguments. If your computer can represent characters like "æ" on its screen, and you've defined `\catcode`\æ=\active` and `\letæ=\ae`, you can't use "æ" in an `\indexentry` either. Nor can you use `~` as a tie. Instead, special characters are coded by leaving out the \ and surrounding what remains with `||`, like this: `|dh|` for ð, `|th|` for þ, etc. Active characters, like `~`, if they are used in `\indexentry` at all, must use a similar coding using only non-active characters. To use special characters that are only available in math mode, just surround the coding with `$$`, e.g., `$|aleph|$`. Using `||` is actually better, since using the normal codings could result in a lot of nested braces, which would make the input file difficult to read, especially since `\indexentry` already has 6 sets of braces. (Incidentally, Spindex includes an Emacs-Lisp function for writing `\indexentry` which queries for the arguments and puts them inside the braces automatically.)

Here are some examples of using special characters in `\indexentry`.

```
\indexentry{|th|eir}{}{}{}{s|'a|}{}
\indexentry{s|ae|tninger}{}{}%
     {S|"a|tze}{}{}
\indexentry{$|aleph|$}%
     {$|aleph|$ --- The letter aleph}%
     {}{}{}{}
\indexentry{|poll|}%
     {|poll| -- Polish |poll|}%
     {}{}{}{}
```

⟹

```
ℵ — The letter aleph . . . . . . . . . . 54.
ł – Polish ł . . . . . . . . . . . . . . 54.
sá
   þeir . . . . . . . . . . . . . . . . . 54.
sætninger . . . . . . . . . . . . . . . . 54.
See also: Sätze
```

`||` can be used to code anything, in particular, any control sequence, not just special characters. For example:

```
\indexentry{{|it|verbs}}{}{}{}{}{}
```

⟹

```
verbs . . . . . . . . . . . . . . . . . . 19.
```

You could achieve the same effect with

```
\indexentry{verbs}{{|it|verbs}}{}{}{}{}
```

but there is a difference. If

```
\indexentry{verbs}{}{}{}{}{}
```

and

```
\indexentry{{|it|verbs}}{}{}{}{}{}
```

were both used in an input file, they would create two different entries, printed on different lines, one in the current font (probably roman) and one in italic, but the entries would be identical with respect to alphabetization. Their order in the index file would correspond to the order of the invocations of `\indexentry` in the input file. In most cases, it will be easier to put a font change in the `text` argument, but in special circumstances it might be better to have it in the `name` argument instead.

2.2.1. Customizing `spindex.lsp`. There is a huge number of special characters available and each project will have its own special requirements. Even when the same characters are used, their order may differ. For these reasons, it is necessary for the user to customize `spindex.lsp` for each set of requirements. This is not difficult. In `spindex.lsp` you will find a list that looks like this.

```
(a b c d dh e f g h i j k l m
  n o p q r s t u v w x y z
  ae oslash acirc thorn)
```

These are the characters that will be assigned a unique integer value, in ascending order, for alphabetical sorting. The exact items in this list will depend on the user's requirements. A function called set-char-values assigns the integer values to variables with names based on the items in this list, i.e., a-value, b-value, ..., thorn-value. Usually, more than one character will occupy the same position in the alphabet, so not all of the characters used will require their own value. Some share a value with a character in the list, for example, according to some alphabetization conventions, "á", "à", and "ā" will all use a-value. All of the uppercase letters share a value with their corresponding lowercase letters. In some languages, ligatures like "æ" and "œ" are treated as "a e" and "o e" respectively, so they are assigned a list of two values, i.e., (a-value e-value) and (o-value e-value). In Danish, however, "æ" has its own position toward the end of the alphabet, so if a user needs an index sorted according to Danish conventions, set-char-values will have to assign an integer value to a symbol for "æ".

Each ordinary character and special coding that may appear as an argument in `\indexentry` must be accounted for in the function letter-function in `spindex.lsp`. This is how the code in letter-function looks for an ordinary character:

```
((or (equal local-string "a")
     (equal local-string "A"))
 (setq current-int-list ‘(,a-value)))
```

Here's how the code looks for a special character:

```
((or (equal local-string "thorn")
     (equal local-string "th"))
 (setq current-int-list ‘(,thorn-value))
 (setq current-tex-code "{\th}"))
```

This tells `spindex.lsp` that `|th|` and `|thorn|` are valid special codings, that they are assigned the value thorn-value, and that they are to be replaced with `{\th}` when `spindex.lsp` writes the index file. Note that the names of the symbols need not correspond to the coding used in `\indexentry`: "þ" is coded as `\th` in TeX and can be coded as `|th|` or `|thorn|` in `\indexentry`. However, in the character list, the symbol associated with "þ" is called thorn. In other cases, the name of a symbol is not permitted to be the same as the coding in TeX and `\indexentry`. For instance, the coding for "ø" is `\o` and can be coded as `|o|` in `\indexentry`. However, the symbol in the character list may not be o, because this is already used for "o". So the symbol in the character list is called oslash. If a character like "ä", coded as `\"a` in TeX and `|"a|` in `\indexentry`, should be assigned its own value, the symbol name would have to be something like aumlaut instead of `"a`, since the `"` would cause a fatal error in `spindex.lsp`. Spindex includes detailed instructions for customizing the Lisp program.

2.3. Overview of `\indexentry`'s arguments

- Argument `#1` (`name`). Only required argument. Used for alphabetizing entries at all levels (heading, subheading and subsubheading). Printed to index file unless `#2` (`text`) is non-empty.
- Argument `#2` (`text`). Printed to index file if non-empty, but entry is alphabetized according to `name`. Also used when a cross-reference refers to this entry. Can be used for comments and other special purposes.
- Argument `#3` is used for suppressing the page number. Any string containing only characters of `\catcode`=11 ("letter") and/or `\catcode`=12 ("other") can be used safely.
- Argument `#4` (`cross-reference`). Can be an arbitrary string or refer to another entry at any level, using a special syntax described above. Entries at any level can have cross-references (see page 257).
- Argument `#5` (`heading`). Will be empty if the entry is a heading. If the entry is a subheading or a subsubheading, this argument refers to the heading entry, of which this entry is a sub- or subsubheading. Used for making a Lisp symbol.

- Argument #6 (`subheading`). Will be empty if the entry is a heading or a subheading. If the entry is a subsubheading, this argument refers to the subheading entry, of which this entry is a subsubheading. Used for making a Lisp symbol.

2.4. Running Spindex. The `\indexentry` macro may write a marginal hack, but otherwise it has no effect on the file in which it is used. It simply writes a file of Lisp code that's used to generate another TEX file. Spindex does not in itself make any connection between the two TEX files. The user can (and must) decide what to do with them.

I use a combination of a UNIX shell script and a TEX driver file to control running TEX and Lisp. This is a rather complicated topic, since I also use them to control other things, like generating the table of contents, the bibliography, page references, etc. I plan on describing this technique in a subsequent article, but here is a simple example just for the index.

```
#### This is the shell script run_driver

if [[ -f index_switch.tex ]]
then
rm index_switch.tex
fi

tex driver

if [[ -f index_switch.tex ]]
then
gcl<"spindex.lsp"
tex driver
else
echo "There were no index entries"
fi
```

```
%%% This is the TeX driver file
%%% driver.tex

\newif\iffirstrun
\newread\indexin
\openin\indexin=index_switch
\ifeof\indexin
\firstruntrue
\else
\firstrunfalse
\let\suppressindex=t
\fi
\closein\indexin

\input spindex

\input input_file

\iffirstrun
\message{This is the first run,
     not inputting index}%
\else
\message{This is the second run,
     inputting index}%
\vfil\eject
\input index
\fi
\bye
```

The shell script `run_driver` runs TEX on the file `driver.tex`. If `\indexentry` isn't used, then `run_driver` is finished. Otherwise, it runs `spindex.lsp` to create the index file. Then it runs TEX on `driver.tex` again. This time, no file of Lisp code is written; instead, `driver.tex` inputs the index file and TEX exits.

2.5. "Faking" an index. Since `entries.lsp` and `index.tex` are both ordinary ASCII files, it's possible to edit them as one would edit any TEX file or Lisp program. Since they are automatically generated and old versions are overwritten, this would only make sense for polishing a final draft. But it is possible. More practical is a dummy TEX file that contains invocations of `\indexentry` but no text to be typeset, like the examples above. Explicit page breaks and numbering must be specified. This is an example of an index produced using a dummy file:

ℵ — The letter aleph . . . . 23.
alphabets
  Polish . . . . 12–16.
Danish words . . . . 122.
ð – The letter italic *ð*
See: þ (The letter thorn)
ð – The letter bold face **ð** . . . . xx.
ł – Polish ł . . . . 24.
See also: alphabets, Polish
ŋ (a phonetic symbol) . . . . 18.
nouns . . . . . viii–xxi, 11, 121–23, 146–49.
See also: verbs
parts of speech . . . . x–xiv, 12.
See also: nouns *and* verbs
Sätze
  übergeordnete . . . . 12.
  untergeordnete . . . . 13.
sætninger . . . . 24.
See also: Danish words *and* Sätze
verbs . . . . 12.
  intransitive . . . . 121.
  transitive . . . . 12.
  See also: verbs, intransitive
    active (except deponentia) . 3, 12–27. 120–22.
    See also: nouns; Sätze *and* øllebrø**ð**
    passive . . . . viii.

```
words
    abstractions
        åbenhed
        See: Danish words
This is a comment where yyy would be.
øllebrøð  . . . . . . . . . . . . . . . . . 13.
See also: Danish words
åndsarbejde . . . . . . . . . . . . . . . . 17.
See also: Danish words
þ (The letter thorn)  . . . . . . . . . . . 12.
```

and this is the beginning of the dummy file that produced it:

```
%% This is dummy_index.tex
\input spindex
\input ipamacs
\font\ipatenrm=wsuipa10
\def\ipa{\ipatenrm}
\pageno=3
\indexentry{yyy}{This is a %
     comment where yyy would be.}%
     {np}{}{}{}
\indexentry{active}{active %
     (except deponentia)}%
     {}{nouns}{verbs}{transitive}
\hbox{}\eject
\pageno=122
\indexentry{active}{}{}%
     {|o|llebr|o||dh|}%
     {verbs}{transitive}
\hbox{}\eject
\pageno=121
\indexentry{active}{}{}{S|"a|tze}%
     {verbs}{transitive}
\hbox{}\eject
\pageno=120
\indexentry{active}{}{}{S|"a|tze}%
     {verbs}{transitive}
```

The complete dummy file contains a total of 73 `\indexentry` commands.

2.6. Getting Spindex. Spindex will be available on an ftp server under the normal conditions applying to free software. If you are interested, please contact me via email and I will tell you where to get it. The program `spindex.lsp` was written using the Gnu Lisp interpreter, which is free. The program itself should work without any trouble with a different Common Lisp interpreter; only two non-essential functions use the operating system interface, which always depends on the particular Lisp interpreter you're using. Getting these two functions to work with a different interpreter should require only minor adjustments.

3. Programming Spindex

3.1. Why not LaTeX? Spindex is designed for use with `plain` TeX. It's possible to use it with LaTeX, too, as mentioned above, but there are some difficulties involved. I find that LaTeX works well as long as one of its pre-defined formats can be used without significant changes. However, if modifications are necessary, I find that programming a format with `plain` TeX is much easier and gives better results. It's always a little risky to write macros when using a large package like LaTeX that already contains a lot of macros. In LaTeX especially, it's difficult to figure out exactly what macro or assignment is causing a certain effect, or even to understand the macro definitions. Many packages also change the `\catcode` of characters, which can cause serious problems. For instance, if you use a package that sets `\catcode`\|=\active`, Spindex will fail.

The program in `spindex.lsp` functions independently of TeX or LaTeX and only one change is necessary to make `\indexentry` work in LaTeX: `\pageno` must be replaced by `\thepage`. The actual text of the index entries, the headings, subheadings, subsubheadings, page numbers and cross-references, will be the same whether you use TeX or LaTeX. However, `spindex.lsp` also writes formatting commands to the index file, and these must be compatible with the format and the output routine being used. The version of `spindex.lsp` that I'm making available writes formatting commands appropriate to the simple `plain` TeX format and output routine that are included in `spindex.tex`. The formatting is performed by a combination of the code written to `index.tex` by `spindex.lsp` and the definitions in `spindex.tex`. Since the formatting commands written to `index.tex` are defined in a general way, it's possible to make significant changes just by changing the definitions in `spindex.tex`, without making any changes to the Lisp program. However, if the user wants `spindex.lsp` to write different formatting commands, it's easy to modify it.

Using Spindex with LaTeX will require some experimentation to get it to produce the kind of formatting desired. Anyone who wishes to do this may feel free. There are many LaTeX formats and I rarely use any of them, so I have no interest in doing this experimenting. This is a task best left to a LaTeX programmer who really uses the formats.

3.2. Why Lisp? While it is possible to get TeX to jump through hoops, I usually find it easier to let TeX do what it does best, typesetting, and use a conventional programming language for things like storing and manipulating data, alphabetizing, writing files, etc. While C seems to be the language of choice for front-end programs for TeX, Lisp offers a number of significant advantages, partly due to Lisp code being interpreted rather than compiled. It's possible to have TeX write executable Lisp code directly, so that it is unnecessary to write routines for reading data from files, and Lisp code is easier to test and debug than program code that must be compiled. Lisp also has many functions for sorting and manipulating strings and, of course, lists, Lisp's characteristic data type. In addition, the structure of the program in `spindex.lsp` depends on Lisp's ability to use undeclared variables, which is not possible in C. The program `spindex.lsp` is not very long, and it runs fast, at least on the installation I'm using (a Dec Alpha computer running Digital UNIX). I use the Gnu Lisp Interpreter, which is free and works well. Unfortunately, it does not conform to the newest standard described in Guy L. Steele's *Common Lisp. The Language*, 2nd ed., 1990, but that hasn't turned out to be a problem.

3.3. The TeX macro `\indexentry`. Spindex uses the conditionals (`\newif`s) `\ifdraft` and `\ifindex` and the control sequences `\suppressindex` and `\firstindexentry`. We've already seen `\ifdraft`; it's used for telling `\indexentry` whether to write a marginal hack or not. The conditional `\ifindex` and the control sequence `\suppressindex` are used for telling TeX whether to make an index or not. The file `spindex.tex` contains the lines

```
\indextrue
%\indexfalse
```

one of which should be commented out, depending on whether you want an index or not. There's another way of suppressing the index, though, without changing `spindex.tex`. The input file can contain the line `\let\suppressindex=t` or `\def\suppressindex{}` before the line `\input spindex`. Then, if `\indextrue`, `\indexfalse` is set instead.

```
\ifindex
\ifx\suppressindex\undefined
\message{\noexpand\indextrue. %
     Will make an index, if there %
     are any entries.}
\else
\indexfalse
\fi\fi
\ifindex
\else
\message{\noexpand\indexfalse. %
     Won't make an index, %
     even if there are entries.}
\fi
```

Then, the definition of `\indexentry` is put inside a conditional using `\ifindex`.

```
\ifindex
\def\indexentry#1#2#3#4#5#6{...}\else
\def\indexentry#1#2#3#4#5#6{\relax}\fi
```

If `\ifindex` expands to `\iffalse` (`\ifindexfalse`), `\indexentry` simply eats its 6 arguments. The control sequences `\firstindexentry` and `\suppressindex` are used as Boolean variables. They can expand to a single token or be undefined, and are used in conditional constructions. Their specific values, if any, are not really important, so I like to use `n` and `t`, like nil and `t` in Lisp. The TeX driver file `driver.tex` uses `\suppressindex` the second time TeX is run on it in order to prevent `\indexentry` from overwriting `entries.lsp`.

The line `\let\firstindexentry=t` appears in `spindex.tex`. Assuming `\indextrue`, if the control sequence `\firstindexentry` expands to `t` (i.e., the first time `\indexentry` is invoked), it calls the macro `\beginindex`, which performs certain actions that only need to be performed once. It opens a file called `index_switch.tex` and writes something to it. It doesn't matter what it writes — all `index_switch.tex` has to do is exist. It's used for running Spindex with the UNIX shell script and the TeX driver file described on page 261. TeX cannot directly access shell variables or execute commands in a shell, and a shell script cannot directly influence TeX when it's running. However, both can write and test for the existence of files, so I use `index_switch.tex` to communicate between `run_driver` and `driver.tex`.

We're done with `index_switch.tex` now, so the output stream is closed and freed to be reallocated, if necessary. Now `\beginindex` opens the file which will contain the Lisp code for the index entries. In this article I call it `entries.lsp`, but actually it can have any name within reason. Then it says `\let\firstindexentry=n`, so these actions won't be performed again.

Next, `\indexentry` takes arguments `#2`–`#6` and puts them in boxes. It checks the width of the boxes and behaves appropriately, simulating the effect of true optional arguments. This is a useful trick that does not appear in *The TeXbook*. It's not as neat as a look-ahead mechanism using `\futurelet` or `\afterassignment` and `\let`, but it's a lot easier to code. Here's a simple example of this technique:

```
\setbox2=\hbox{#2}%
\ifdim\wd2>0pt
\message{There's something in %
     argument 2}%
\else
\message{Argument 2 is empty}%
\fi
```

Above I state that six sets of braces must always follow `\indexentry`. Strictly speaking, of course, this isn't true, but TEX will consider the six tokens or groups that follow `\indexentry` to be its arguments, so leaving out the braces (or characters with `\catcode=1` and `\catcode=2`) is hardly practical. The `\indexentry` macro writes code to `entries.lsp` based on what's in its arguments. Argument `#1` is required, so `\indexentry` doesn't need to put it in a box. It writes

(generate-entry @⟨*name*⟩@

The `@` symbol is used as a string delimiter instead of `"` in order to make it possible to use `"` in `\indexentry`'s arguments: `|"a|` for "ä", `|"o|` for "ö", etc. This means that `@` "as is" in an argument to `\indexentry` will cause a fatal error. But `|@|` works. The other arguments are put into boxes.

```
\setbox2=\hbox{#2}%
\setbox3=\hbox{#3}%
\setbox4=\hbox{#4}%
\setbox5=\hbox{#5}%
\setbox6=\hbox{#6}%
```

Then,

```
\ifdim\wd2>0pt
\write\index{\space\space\space %
     :text @#2@}%
\fi
```

causes

:text @⟨*text*⟩@

to be written to `entries.lsp` if `#2` is non-empty, and similarly for the other four arguments, except that `#3` (for suppressing the page number) is treated a little differently, since the page number is printed by default:

```
\ifdim\wd3=0pt
\write\index{\space\space\space %
     :page-no \the\pageno}%
\fi
```

$\Longrightarrow$

:page-no ⟨*page number*⟩

if `#3` is empty. After the arguments `#2` through `#6` are tested for existence and the code (if any) is written to `entries.lsp`, `\indexentry` writes a closing parenthesis to match (generate-entry @⟨*name*⟩@. Here are some examples:

```
\indexentry{nouns}{}{}{}{}{}
```

$\Longrightarrow$

```
(generate-entry @nouns@
   :page-no 1
   )
```

```
\indexentry{masculine}{masc.}%
     {}{}{nouns}{}
```

$\Longrightarrow$

```
(generate-entry @masculine@
   :text @masc.@
   :heading @nouns@
   :page-no 1
   )
```

```
\indexentry{a-stems}{}{x}{verbs}%
     {nouns}{masculine}
```

$\Longrightarrow$

```
(generate-entry @a-stems@
   :heading @nouns@
   :subheading @masculine@
   :cross-ref @verbs@
   )
```

```
\indexentry{s|ae|tninger}{}{}%
     {S|"a|tze}{}{}
```

$\Longrightarrow$

```
(generate-entry @s|ae|tninger@
   :cross-ref @S|"a|tze@
   :page-no 24
   )
```

The `\write` commands in `\indexentry` are the reason why it can't use the normal coding for macros in its arguments, i.e., the coding using backslashes, like `\th`, `\oe` and `\it`. A `\write` command will expand an expandable macro, and write an unexpandable one as is, but with a following space. There's more about this topic in section 3.6.

After TEX is done with the input file, and all of the index entries have been processed, the output stream `\index` associated with the file `entries.lsp` should be closed. I redefine `\bye` so that it calls the function `\endindex`, which is defined like this:

```
\ifindex
\def\endindex{\closeout\index}
\else
\def\endindex{\relax}
\fi
```

3.4. The Lisp program `spindex.lsp`. This program loads the file of Lisp code, `entries.lsp`, which was written by the `\indexentry` commands. This file consists of invocations of the Lisp function generate-entry, which uses `\indexentry`'s `name` argument, and its `heading` and `subheading` arguments, if present, to access a symbol (or variable). Since the names of these symbols depend on the arguments to `\indexentry`, they can be different each time Spindex is run and therefore cannot be declared in `spindex.lsp`. This may appear to be dangerous, but it isn't. Lisp has very few reserved words. Most of its internal variables begin and end in `*`, like `*package*`. If an index entry is made with a `name` that duplicates the name of a Lisp function, like car, this will not cause an error (or even a problem), because each Lisp symbol has a function cell and a value as a variable, and the interpreter can tell from the context which is meant. Also, safety routines can be written to catch dangerous names before the string is used to create a symbol. There is one for entries beginning and ending in asterisks, "T" and "NIL". The Gnu Lisp interpreter has named constants that don't begin and end in `*`, but it will signal an error if an attempt is made to change their values. However, they are represented internally in uppercase letters, and the symbols created by generate-entry probably won't be, so it's unlikely that these constants will cause any problems. If they do, it's still possible to write safety routines to take care of them.

3.4.1. Generating the entries. The name, heading and subheading arguments to generate-entry are all strings and undergo some manipulation before they are used as the names of Lisp symbols. Therefore, some characters may appear in arguments to `\indexentry` which would normally cause problems in Lisp, for instance, an index entry like "Lincoln,␣Abraham" is legal, whereas commas and spaces may not normally appear in symbol names in Lisp. If there is no heading argument, the entry is a heading, and the name of the symbol is name. If heading (but not subheading) is non-empty, the entry is a subheading, and heading and name are joined with a hyphen: heading-name. If heading and subheading are both non-empty, the entry is a subsubheading, and heading, subheading and name are joined with a hyphen, e.g.,

```
\indexentry{transitive}{}{}{}{verbs}{}
```

maps to the symbol name

|verbs-transitive|

and

```
\indexentry{active}{}{}{}{verbs}%
    {transitive}
```

maps to the symbol name |verbs-transitive-active|.

The use of `||` surrounding the symbol name in `spindex.lsp` is independent of the use of `||` to delimit special character codings in `\indexentry`'s arguments. In Lisp, `|`⟨*characters*⟩ `|` has the effect of escaping all of the characters inside `||`, so that characters can be used in the name of a Lisp symbol that would normally not be allowed. This also makes it possible to have symbol names with lowercase letters. Lisp normally ignores case and converts lowercase letters in symbol names to uppercase letters internally. But this would mean that

```
\indexentry{a}{a (the letter a)}{}{}{}{}
```

and

```
\indexentry{A}{A (the letter A)}{}{}{}
```

would map to the same Lisp symbol and therefore not create two different entries, and the `text` "A (the letter A)" would be ignored, because `text` is only used when an entry is created, as explained above. So all lowercase letters are escaped as well as space, comma, and indeed everything except for uppercase letters, which are not escaped, and `{` and `}`, which are ignored.[2] However, this special meaning of `|` in Lisp means that an index entry for "þat" and one for "that", created by

```
\indexentry{|th|at}{}{}{}{}{}
```

and

```
\indexentry{that}{}{}{}{}{}
```

would both map to a Lisp symbol called |that|, since the `||` in `|th|at` would be interpreted by Lisp simply as escape characters. In order to prevent this, `||` in an `\indexentry` are converted to `|!` and `!|` so that the two invocations of `\indexentry` above map to two different symbols, |!th!at| and |that|. The exclamation points have no effect on alphabetization or on the output to `index.tex`, since sorting and output both use the original, unconverted name argument.

Now generate-entry accesses the symbol (using read-from-string) and checks to see if it's bound. If it isn't, it means that this is the first occurrence of this entry. In this case, a structure of type "entry" (defined by defstruct entry) with the slots name,

[2] The way characters or groups of characters are handled can be modified according to the user's requirements.

text, sort-string, page-nums, cross-refs, cross-ref-cons, subheadings and subsubheadings is created and the symbol is bound to it. The information in generate-entry's other arguments is stored in the appropriate slots. If the symbol is bound, i.e., the entry already exists, the page number and cross-reference information in generate-entry's arguments may be added to the appropriate slots in the structure, unless it's already there due to previous invocations of `\indexentry`.

It's easier to "fake" an index using the function generate-entry than it is to use a dummy input file. If one wants to type in the code for invocations of generate-entry, there's no need to use `\indexentry` at all, for instance, to make an index for a book that's already been printed or that's not made using TeX. In this case, it would make sense to redefine generate-entry so that it could take lists of strings and integers for its cross-ref and page-num keyword arguments. Then generate-entry need only be invoked once for each entry.

3.4.2. The sort strings. The name argument is used to make a string to be stored in the sort-string slot of the entry structure. This is what makes it possible for Spindex to alphabetize special characters.

Lisp's sorting routine for characters and strings, like C's and UNIX' sorting routines, can sort the 256 characters of an 8-bit character encoding according to a code table based on the ASCII code table. For sorting strings using only English words this is adequate, but most of the special characters likely to appear in an index do not appear in the ASCII code table (or in Lisp's), and most of the characters that *do* appear in the code table are unlikely to appear in an index. Since uppercase letters (positions 65–90) and lowercase letters (positions 97–122) are treated identically for purposes of alphabetization, and it makes no sense to sort numerals or punctuation marks according to their position in the code table, only 26 positions are relevant and 229 are wasted.

Spindex makes it possible to use all 256 positions, or as many of them as necessary, by assigning integer values to a set of variables, i.e., a-value = 1, b-value = 2, etc. Each letter or special character is associated with a list of one or more of these values. The characters a, b and þ are associated with the lists (a-value), (b-value) and (thorn-value) respectively On the other hand, in some languages the ligature "æ" is treated as "a e", so it's associated with the list (a-value e-value). This is the reason for associating characters with lists rather than single integers.[3]

Some characters should be sorted as if they were other characters. All of the uppercase characters should be treated the same as their corresponding lowercase characters, and in some styles of alphabetization "á", "à", "ā", etc. should be treated like "a", so that the list associated with "á" (coded as `\'a` in TeX and `|'a|` in `\indexentry`) should be (a-value). On the other hand, in Icelandic, "á" follows a in the alphabet (likewise for the other vowels), so "á" would need to have a unique value aacute-value such that a-value < aacute-value < b-value. While `spindex.lsp` can assign integer values only from 0 to 255, in practice many more characters can be accommodated, because some characters receive the same values and others use combinations of values assigned to other characters.

The string which was the `name` argument to `\indexentry` is read character by character, except that a | causes everything up to the next | (a special coding) to be treated as a unit. The function letter-function returns lists of integers to the function generate-info, which creates a new string using the characters from the code table *that have these values.* So, the sort-string for an `\indexentry` "nouns" might look like `"^P^Q^W^P^U"` (consisting of non-printing characters in Lisp's printed representation). It doesn't matter what the sort-string looks like because the user never even needs to know it exists, and the characters which are assigned will vary according to the content of the character list described on page 260. The sort-string for "transitive" might look like

```
"^V^T^A^P^U
 ^V
 ^X^F"
```

where i-value is assigned the integer 10 corresponding to the newline character, as in Fig. 1. The function set-char-values keeps track of how many there are and signals an error if they exceed 256. Spindex can be made to perform alphabetical sorting for languages using non-Latin alphabets if the user makes an appropriate list, or an index can

[3] It would be possible to change the indexing program so that the characters could be associated either with a single integer or a list of integers. If I revise `spindex.lsp` I will probably make this change, but only for aesthetic reasons.

be reversed or scrambled by changing the order of the characters (if anyone wanted to do this).

After the sort string has been generated, it is stored in the entry structure's sort-string slot. Then generate-entry makes a cons cell and puts the sort string into the car and the symbol itself into the cdr.

```
\indexentry{verbs}{}{}{}{}{}
```

$\Longrightarrow$

```
("^X^F^T^B^U" . |verbs|)
```

If the entry is a heading, this cons cell is put into an association list, or alist, called sort-list. If the entry is a subheading, the cons cell is put into an alist in the subheadings slot of the heading entry of which it is a subheading; if it's a subsubheading, it's put into an alist in the subsubheadings slot of the subheading entry of which it is a subsubheading. Got that?[4]

If a subheading is created before its heading exists, e.g.,

```
\indexentry{transitive}{}{}{}{verbs}{}
```

without a preceding

```
\indexentry{verbs}{}{}{}{}{}
```

|verbs| must be created in order for |verbs-transitive| to be stored with its sort string in |verbs|'s subheadings slot. This is accomplished by means of a recursive call to generate-entry. If

```
\indexentry{active}{}{}{}{verbs}%
     {transitive}
```

is invoked before

```
\indexentry{transitive}{}{}{}{verbs}{}
```

|verbs-transitive| is generated by a recursive call to generate-entry, and |verbs|, too, if it doesn't exist already. The page number is suppressed for entries that are generated automatically in this way, and there is no way to specify a text for them. This is another reason for putting dummy entries at the beginning of your input file for specifying texts.

3.4.3. Page numbers. By default, the macro \indexentry writes the page numbers to the file entries.lsp. When an entry is created, if the page number has not been suppressed, a list containing the page number is stored in the entry structure's page-nums slot. For each additional call to \indexentry the page number (if it hasn't been suppressed) is simply added onto the list, unless that page number is already in the list due to a previous invocation of \indexentry on that page. It would be possible to change this in order to keep track of the number of occurrences per page. This is unnecessary for an index, but it might be useful for some other application. Usually, the page numbers will occur in order in the page number list, however, spindex.lsp sorts the list before writing the page numbers to index.tex, so they will be in the correct order even if the user explicitly changes the page number in the input file with \pageno=⟨*integer*⟩ in such a way that the pages are numbered out of order.

3.4.4. Cross-references. A cross-reference (argument #4 to \indexentry) can refer to another entry (at any level) or it can be an arbitrary string. Whichever it is, it is stored as is (the string is not converted) in a list with all the other cross-references for this entry in the cross-refs slot of the entry structure.

When a heading entry is first created, its text argument (or if text is empty, its name argument) is used to make a cons cell that is stored in that entry's cross-ref-cons slot. This is used when this entry is used as a cross-reference in another entry. A subheading entry uses a string consisting of the text or name of its heading, a comma, a space, and its own text or name. A subsubheading entry uses a string consisting of the text or name of its heading, a comma, a space, the text or name of its subheading, a comma, a space, and its own text or name. This string is stored in the cdr of the cons cell, and given to generate-info, which returns a sort-string, which is stored in the car of the cons cell. Cross-references, unlike entries, are always alphabetized according to what is actually printed.

An index entry is illustrated in Fig. 1.

3.4.5. Output. After spindex.lsp has loaded the file entries.lsp, it puts the cons cells in sort-list (the alist containing the heading entries) into alphabetical order according to their cars, i.e., the sort-strings, with

```
(setq sort-list
   (sort sort-list #'string<
                 :key #'car))
```

Now the heading entries are in alphabetical order and the function export-entries simply pops each cons cell off of sort-list, evaluates the symbol in the cdr to get the entry structure, extracts the information for each entry and writes it to the TEX file index.tex (as with entries.lsp, any name

[4] The subsubheading slot of a heading entry, the subheading slot of a subheading, and both of these slots in a subsubheading will always be nil.

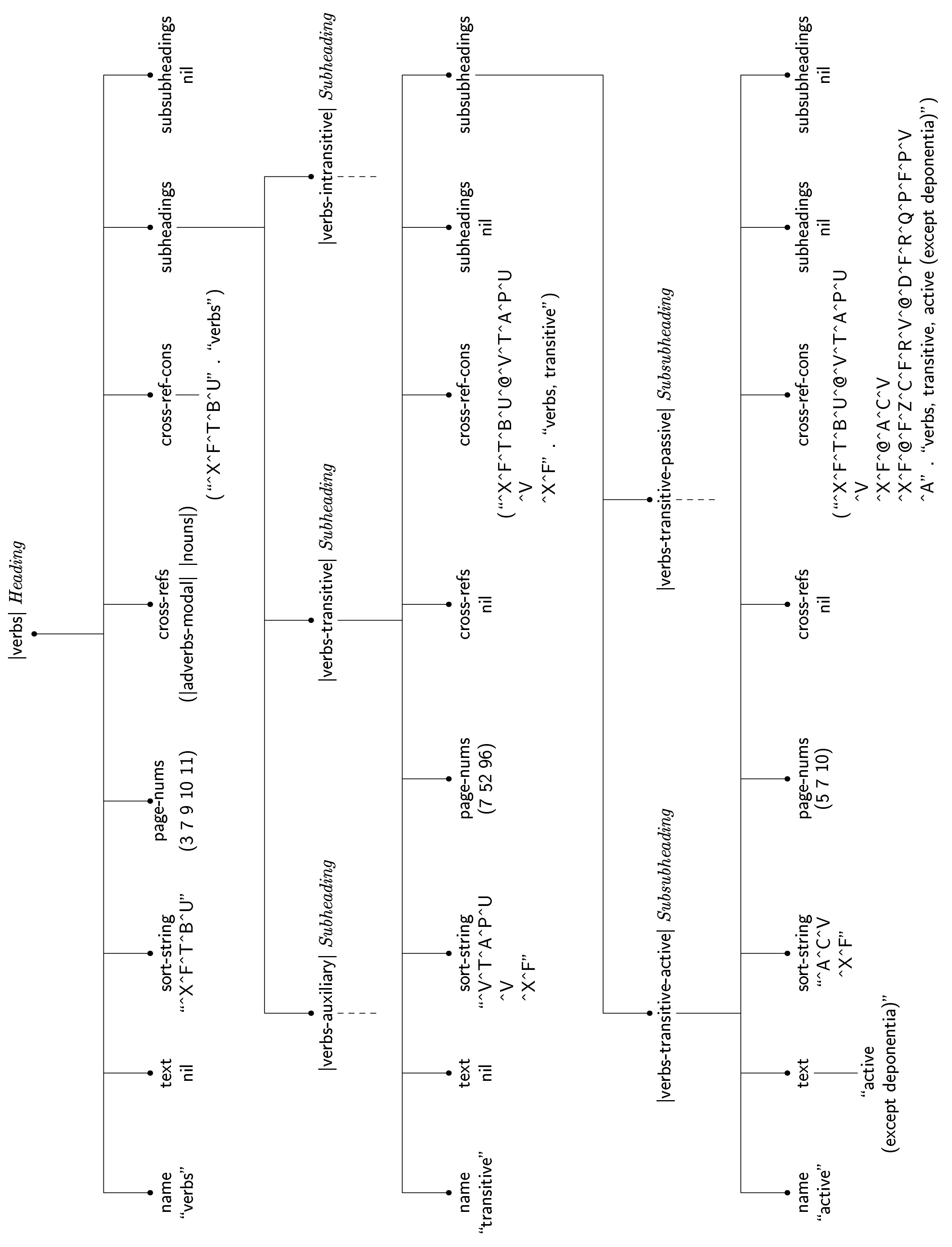
|verbs| *Heading*
name "verbs"
text nil
sort-string "^X^F^T^B^U"
page-nums (3 7 9 10 11)
cross-refs (|adverbs-modal| |nouns|)
cross-ref-cons ("^X^F^T^B^U" . "verbs")
subheadings
subsubheadings nil
|verbs-auxiliary| *Subheading*
|verbs-transitive| *Subheading*
|verbs-intransitive| *Subheading*
name "transitive"
text nil
sort-string "^V^T^A^P^U ^V ^X^F"
page-nums (7 52 96)
cross-refs nil
cross-ref-cons ("^X^F^T^B^U^@^V^T^A^P^U ^V ^X^F" . "verbs, transitive")
subheadings nil
subsubheadings
|verbs-transitive-active| *Subsubheading*
|verbs-transitive-passive| *Subsubheading*
name "active"
text "active (except deponentia)"
sort-string "^A^C^V ^X^F"
page-nums (5 7 10)
cross-refs nil
cross-ref-cons ("^X^F^T^B^U^@^V^T^A^P^U ^V ^X^F^@^A^C^V ^X^F^@^F^Z^C^F^R^V^@^D^F^R^Q^P^F^P^V ^A" . "verbs, transitive, active (except deponentia)")
subheadings nil
subsubheadings nil

Fig. 1. A heading entry with sub- and subsubheadings.

within reason can be chosen). Headings are not indented, subheadings are indented to the value of `\parindent` and subsubheadings are indented to twice this value. The function generate-info converts the text or name string of each entry into TeX coding, which is written to `index.tex`. When export-entries processes a heading entry, and the subheading slot is non-nil, then the alist in the slot is sorted and export-entries is called recursively. If a subheading entry's subsubheading slot is non-nil, then the alist it contains is sorted and export-entries is called recursively. If there are page numbers associated with an entry, leaders are printed and then the page numbers, separated by commas and followed by a period. It is possible, if unusual, that an `\indexentry` could appear in the front matter, and that the page number would therefore be negative. In this case, export-entries will cause that page number to be printed as a lowercase Roman numeral. If no page numbers are associated with an entry, either because they have all been suppressed, or because an entry was only generated automatically by a sub- or subsubheading entry and `\indexentry` was never called for it in its own right, no leaders are printed. If there are page numbers and cross-references, the cross-references are printed on the following line, indented to the same degree as the entry, preceded by the text "*See also*". If there are cross-references but no page numbers, the cross-references are preceded by the text "*See*". If there are two cross-references, they are separated by the word *and*. If there are more than two, the final two are separated by the word *and* and the others by a semi-colon. Of course, the strings *See*, *See also*, and *and* can be changed for books in languages other than English.

If an entry has no page numbers, no cross-references and there are no sub- or subsubheadings, a warning message is issued. Non-consecutive pages are simply written to `index.tex` and separated by commas. Page ranges are printed as the first and last number in the range, separated by an en-dash (–), whereby the last number may be abbreviated according to the following scheme:

- Let a and b be integers such that $0 < a < b$. a and b represent the beginning and end of a page range.
- If $a < 10^2$, b is not abbreviated: 1–9, 27–100.
- Else if $\lfloor a/10^2 \rfloor = \lfloor b/10^2 \rfloor$ and $(b \bmod 10^2) \geq 10$, b is abbreviated to $(b \bmod 10^2)$: 100–12, 254–99, 1104–29.
- Else if $a \geq 10^3$, $\lfloor a/10^3 \rfloor = \lfloor b/10^3 \rfloor$ and $(b \bmod 10^3) \geq 10^2$, b is abbreviated to $(b \bmod 10^3)$: 1003–125, 2006–194.
- Else if $a \geq 10^4$, $\lfloor a/10^4 \rfloor = \lfloor b/10^4 \rfloor$ and $(b \bmod 10^4) \geq 10^3$, b is abbreviated to $(b \bmod 10^4)$: 10234–1045, 23245–5321.

And similarly for integer $n \geq 5$:

- If $a \geq 10^n$, $\lfloor a/10^n \rfloor = \lfloor b/10^n \rfloor$ and $(b \bmod 10^n) \geq 10^{n-1}$, b is abbreviated to $(b \bmod 10^n)$, up to $b =$ TeX's maximum legal integer (*The TeXbook*, p. 118), namely $2^{31} - 1 = 2147483647 =$ octal $17777777777 =$ hexadecimal 7FFFFFFF: 170234–81045, 1623245–935321, 2037892089–147483647.[5]

Otherwise b is not abbreviated: 102–109, 198–205, 1002–1009, 19052–21088. In particular, page ranges with Roman numerals are never abbreviated: cv–cxii, and page ranges starting with a Roman and ending with an Arabic numeral are impossible. The program in `spindex.lsp` also includes an option for disabling abbreviation.

A possible improvement to Spindex would be to allow page indications followed by *ff*, and underlined and italic page numbers, as in *The TeXbook* and *The METAFONTbook*. This would require changes to `\indexentry` and `spindex.lsp`, but it wouldn't be too difficult. If there is sufficient interest, I will program an option for different styles of page numbering.

If there is more than one cross-reference, they must be sorted alphabetically before they are written to `index.tex`. The same technique is used as for sorting the entries themselves. For an arbitrary string, generate-info generates a sort-string and puts it and the original string into a cons cell. If the cross-reference refers to another entry, the function do-cross-refs gets the cons cell stored in the cross-ref-cons slot of that entry. All of the cons cells are put into a list and sorted according to their cars, i.e., their sort-strings. Then, their cdrs (the original strings) are converted to normal TeX coding by generate-info and written to `index.tex`. If it's an arbitrary string, a warning is issued, that this cross-reference doesn't correspond to an entry.

The formatting of `index.tex` depends on the code written by `spindex.lsp` on the one hand, and

[5] Actually, the Lisp routine that performs the abbreviation can abbreviate integers up to the value of most-positive-long-float using the Gnu Lisp interpreter. On the computer I'm using, it's $1.7977 * 10^{308}$.

on the TeX format used on the other. None of the formatting is hard-wired into the program. The index file can be a complete TeX input file, it can input other TeX files, or it can be input by another TeX file. If the TeX code written to `index.tex` is formulated in a general way, and parameters are set and macros defined in another file, then the same `index.tex` can produce output according to a wide range of different formats without making any changes to the Lisp program. However, it's not difficult to change the TeX code written by export-entries, if the user prefers to do the formatting this way. I do not recommend changing the routines for the page numbers and cross-references, though, unless you know what you're doing.

3.5. Fine points of alphabetization. The function set-char-values assigns values to characters ≥ 1 and < 256. There are, however, two other possible values, nil and 0. If a character is assigned a value of nil, nothing is added to the sort-string and it is ignored for purposes of alphabetization. The value 0 acts as a word separator and is assigned to ␣. This corresponds to one style of alphabetization, namely alphabetization by word, so that an entry "abc xyz" will appear before an entry "abcdef". If nil is assigned to ␣, then the entries will be alphabetized by letter and spaces will be ignored, so "abcdef" will appear before "abc xyz". Other characters, like hyphen, can also act as word separators by assigning them the value 0 (in this case, it's necessary to be careful with em- and en-dashes in arguments to `\indexentry`). Codings using `||` that contain only hyphens and/or spaces (and contain at least one character), are valid and are assigned the value nil, so they can be used when the hyphens and spaces shouldn't act as word separators. The coding `|tie|` is for a `~` that is assigned the value 0 and therefore acts as a word separator. `|tie-nil|` is the coding for a `~` that does not act as a word separator. Characters like $, *, {, }, ?, !, ;, ., :, etc. are assigned the value nil, so they can appear in index entries and do not affect alphabetization. Some codings, like control sequences for font switching or formatting, can also be assigned the value nil, so that the `|it|` in `\indexentry{{|it|abc}}{}{}{}{}{}` does not affect alphabetization. Curly braces in an argument are ignored both for purposes of alphabetization and for accessing symbols, so that `{abc}` and `abc` will map to the same symbol. The coding `|it|abc` will also map to the same symbol as `{|it|abc}`, but the former should not be used because the switch to italic will be global in `index.tex`. Likewise, the user should not type `{|it|␣ abc}` because spaces, even spaces following control sequences, are not ignored for purposes of alphabetization (unless ␣ is assigned the value nil), and `{|it|␣ zzz}` would appear in the index before `{|it|abc}`.

Since some characters are assigned the same values, it's possible for entries that print differently to have identical sort-strings. The two entries

```
\indexentry{a}{a (the letter a)}{}{}{}{}
```

and

```
\indexentry{A}{A (the letter A)}{}{}{}{}
```

will have identical sort-strings, namely `"^A"` (assuming a-value = 1). It is impossible to ensure that lowercase letters will always be sorted before or after uppercase letters in situations like this. The order of these entries in the index will be determined by which of them appeared first in the input file. To ensure a particular order of entries of this type (and to ensure that a `text` argument is not ignored) it is safest to use dummy `\indexentry`s with suppressed page numbers at the beginning of the input file.

Indexes generally do not need to do numerical sorting. If the numerals are all assigned the value nil in letter-function, then entries that differ only with respect to the numerals contained in their `names` can be put into order by using dummy entries at the beginning of the input file. However, if a particular application requires it, it should be possible to write a routine that will perform true numerical sorting.

3.6. Some limitations. In its current form, Spindex allows three levels of nesting. It is not considered correct form for indexes to have deeper nesting than this, however, it might be desirable for a special purpose, not necessarily for an index. Spindex could be adapted for deeper nesting by adding an argument for each level to `\indexentry`. However, `\indexentry` already has 6 arguments, and it might be desirable to use the remaining three arguments for some other purpose. It is possible to get around TeX's limit of 9 arguments to a macro, but it's easier if one doesn't have to. Macros with lots of arguments encourage typing mistakes and make the input file difficult to read. Modifying `spindex.lsp` would be less of a problem; for each additional level of nesting the entry structures would need an additional slot, and export-entries would need to be called recursively that many more times.

It would be easy to remove the limitation to 256 positions for alphabetical sorting. Let n be an integer such that $n > 0$ and let α be the set of characters processed by set-char-function. Each character $\in \alpha$ is associated with a single position and assigned a list of n integers. Let β be the set of legal characters $\notin \alpha$ which are assigned lists of n integers, such that each character $\in \beta$ shares a position with a character $\in \alpha$. Let γ be the set of legal characters which are assigned nil. These characters are ignored for purposes of alphabetization, i.e., they are associated with no position. Let δ be the set of legal characters which are associated with lists of integers of length $> n$. The lists assigned to the characters $\in \delta$ may differ in length. For each character $d \in \delta$, let the length of its list be l_d such that l_d is a multiple of n. Then, each character $d \in \delta$ will be associated with x positions such that $x = l_d/n$. Let $\lambda = \alpha \cup \beta \cup \gamma \cup \delta$. Thus λ is the set of legal characters. A string S of length l_S consisting of characters in λ will be associated with y positions where y is the sum of the positions associated with the individual characters in S. Let Z be the sort string derived from S and l_Z its length. Then $l_Z = y * n$. Let p be the number of available positions, then $p = 256^n$. As n increases arithmetically, l_Z increases geometrically and p increases exponentially. If $n = 2$, $p = 256^2 = 65,536$, and for $n = 3$, $p = 256^3 = 16,777,216$. In this way, Spindex can theoretically accommodate infinitely many positions, however, I suspect that increasing n too much would soon cause the Lisp program to run *very slowly* and eventually exhaust the capacity of the computer.

In the format I use, when `\drafttrue`, `\indexentry` causes a marginal hack to be printed next to the line where `\indexentry` appeared in the input file. The marginal hack is printed in the typewriter font `cmtt10`, so an `\indexentry` with `||` like

```
\indexentry{|th|is}{}{}{}{}{}
```

will produce a marginal hack like `|th|is`. If I change the font to roman (`cmr10`), the marginal hack will look like —th—is, because the character — is in same position in `cmr10` as `|` is in `cmtt10` (`"7C`). So I'm limited to using a typewriter font if I want my marginal hacks to look right. Also, two `\indexentry`s on one line will cause the second marginal hack to overwrite the first, causing an unsightly mess. Fixing this would be so complicated that I've decided not to bother, since it's only for rough drafts anyway, and a single line will rarely have multiple invocations of `\indexentry` (except for dummy entries). I'd probably have to define a new class of insertions and I'm not sure it would be possible to get the marginal hacks lined up properly.

Another limitation is that the user can't use normal TeX coding for the special characters and other control sequences in `\indexentry`. Using `||` has advantages, but it would be nice to be able to use normal TeX coding, too.

It is possible to fix this problem, and to have the marginal hack printed in roman type, but the benefit does not justify the increased complexity of `\indexentry`'s definition. However, the solution may be interesting and useful for some other purpose.

To simplify matters, I will use the macro `\next` to illustrate. The following facts are involved:

1. `|` is an ordinary character, `\catcode` = 12.
2. `\write` will expand macros like `\"o`, `\th`, `\it`, the active character `~`, and other active characters like æ if such are defined, and put a space after each unexpanded macro, like `\oe`.
3. Changing the `\catcode` of a character used in an argument to a macro has no effect on that character once it's been read and tokenized.
4. `\write` is not executed immediately. It is put into a whatsit and expansion takes place upon `\shipout`. The macros in the text written by `\write` are therefore expanded according to the definitions in force at the time of the `\shipout`, not when `\write` is invoked (*The TeXbook* p. 227).
5. A delayed `\write` must be used (not an `\immediate\write`) in order to write the page number to the opened file.

The problems can be solved in the following way:

```
%%%% This is next.tex

\newwrite\nextout
\immediate\openout\nextout=next.output
\newlinechar=`\^^J

\def\verticalstroke{|}
\def\foo{foo outside}

\catcode`\|=\active
\let|=\verticalstroke

\def\next{\begingroup
\def\foo{foo inside \noexpand\next}
\def|{vertical inside \noexpand\next}
\catcode`\|=\active
\def\subnext##1##2{%
\immediate\write\nextout%
{This is arg1 inside \noexpand\subnext,
^^J but outside the group:^^J##1}
\immediate\write\nextout%
{This is arg2 inside \noexpand\subnext,
^^J but outside the group:^^J##2}
\begingroup
\def\foo{foo inside}%
26. \def|{vertical inside}%
27. \immediate\write\nextout{This is arg 1
28.      inside \noexpand\subnext,^^J
29.      and inside the group:^^J##1}%
30. \immediate\write\nextout{This is arg 2
31.      inside \noexpand\subnext,^^J
32.      and inside the group:^^J##2}%
33. %%
34. \write\nextout{This is arg 1 at
35.      \noexpand\shipout:^^J
36.      ##1}%
37. \write\nextout{This is arg 2 at
38.      \noexpand\shipout:^^J
39.       ##2}%
40. %% This is for a delayed write of
41. %% the local definitions of the macros
42. %% to \nextout
43. \edef\anext{\write\nextout{^^J%
44.      This is arg 1 at
45.      \noexpand\shipout,^^J
46.      but with the local definition:^^J
47.      ##1}}
48. \anext
49. \edef\anext{\write\nextout{This is arg 2
50.      at \noexpand\shipout,^^J
51.      but with the local definition:^^J
52.      ##2}}%
53. \anext
54. \write\nextout{^^JThis is \noexpand
55.   \catcode\noexpand`\noexpand\|:
56.   \the\catcode`\|}%
57. %% This works
58. \endgroup\endgroup}%
59. \subnext}
60. %% This keeps <macro name> inside \next
61. %% from being written to \nextout
62. %%\endgroup}%
63. %%\expandafter\endgroup\subnext}
64.
65. \catcode`\|=12
66.
67. \next{|}{\foo}
68.
69. \closeout\nextout
70.
71. \end
```

This writes the following text to the file `next.output`

```
This is arg1 inside \subnext ,
 but outside the group:
vertical inside \next
This is arg2 inside \subnext ,
 but outside the group:
foo inside \next
This is arg 1 inside \subnext ,
 and inside the group:
vertical inside
This is arg 2 inside \subnext ,
 and inside the group:
foo inside
This is arg 1 at \shipout :
 |
This is arg 2 at \shipout :
 foo outside

This is arg 1 at \shipout ,
 but with the local definition:
 vertical inside
This is arg 2 at \shipout ,
 but with the local definition:
 foo inside

This is \catcode `\|: 12
```

The `\catcode` of `|` must be set to `\active` outside the definition of `\next`, so that `\def|{...}` will not cause an error. It is set back to 12 (other) after the definition of `\next`. Here, `\subnext` is defined inside of `\next`, but that isn't necessary; it could be defined outside of it, as long as `\catcode`\|=\active` when `\subnext` is defined.

What appear to be arguments to `\next` in line 67 actually are not. Rather, they are arguments to `\subnext`, which therefore must be the last thing in the definition of `\next` before the closing `}`.

Before `\subnext` reads its arguments, `\next` changes the `\catcode` of `|` to `\active`, so it can be defined as a macro. In this example, `|` first expands to `vertical inside \next` and then to `vertical inside` when `\subnext` is expanded. It could also be made to expand to `$\vert$` for a marginal hack, or anything else. At `\shipout`, though, it expands to `|`, i.e., `the character |`. The definition `\def\verticalstroke` in line 7 is necessary to make this possible: because `\catcode`\|=\active`, `\def|{|}` will cause infinite recursion when TeX tries to expand `|`. The definition `\def|{^^7C}` will also fail, because `^^7C` and `|` are equivalent. The `|` in the `\write` command was active when it was tokenized, so it is expanded upon `\shipout` using its global definition, even though `|` is no longer active at this time.

Following this, in lines 40–53, delayed `\write`s are performed using the local definition of `|` and `\foo`. This is accomplished by a trick explained in the answer to Exercise 21.10 of *The TeXbook*:

```
\edef\anext{\write\nextout{##1}}
\anext
```

(a simplified version of the code in line 43–48), causes `|` to be expanded within the definition of `\anext`, before the `\write` command is put into its whatsit. It is, however, necessary to redefine `\anext` for each argument that is to be written to `\nextout`. Even by taking the definition of `\subnext` out of `\next` (this possibility is mentioned above), which would allow the use of arguments in `\anext`'s definition (arguments to macros whose definitions are as deeply nested as the definition of `\anext` is here are not possible, since TEX does not allow parameters like `###1`), and writing

```
\edef\anext##1{{\write\nextout{##1}}%
\anext#1
\anext#2
\anext#3
```

won't work — `vertical outside` and `foo outside` will be written to `\nextout`, apparently because the local definitions of `|` and `\foo` are not accessible inside of `\anext`, but I really don't know the reason.

Macros need not be redefined before the arguments are read. By using grouping, it's possible to have `\subnext` expand the macros in three different ways (or as many as TEX's memory allows), depending on the time of expansion, as in the example above. However, if delayed `\write` commands are used, and the token lists are not expanded beforehand using an `\edef`, it is important to make sure that all macros in the text to be written are defined at the time of `\shipout`. If a macro is only defined within a group, and the group has ended when `\shipout` occurs, it will cause an "undefined control sequence" error.

The group begun in `\next` ends at the end of `\subnext`. If `\endgroup` was placed after `\subnext` is called at the end of `\next`, it would be interpreted as `\subnext`'s first argument. It also doesn't work to write `\expandafter\endgroup\subnext` in line 59 (and remove one of the `\endgroups` in line 58). This will have the effect that `vertical inside \next` and `foo inside \next` are never printed to `next.output`, since these definitions will be inaccessible to `\subnext`. I admit, I don't know why this is. It seems that TEX temporarily "forgets" it's in this group while it's expanding `\subnext`.

4. Final remarks

Spindex runs TEX on an input file which writes information to a file of Lisp code. A Lisp program inputs this file and writes another TEX file. This is only one possibility of using TEX and an auxiliary program in combination. Spindex needs to run TEX initially in order to generate page number information by means of TEX's output routine. This may not be necessary for other applications, so another auxiliary program might operate directly on the TEX input file. Another possibility is storing data in files of Lisp code and using a Lisp program to generate TEX input files. Of course, auxiliary programs can be written in other languages, like C, Fortran, Pascal, etc.

Auxiliary programs like Spindex depend on the fact that TEX input files are ASCII files. The value of this feature of TEX doesn't seem to be recognized as much as it ought to be. It would be impossible, or at the very least impractical, for an amateur (like me) to implement an indexing program for a word-processing package that stores its typesetting data in a format that people can't read. The trend in software is clearly in favor of menu-driven, point-and-shoot programs with colorful graphics and sound effects. While programs of this sort are superficially easier to use than packages like TEX and METAFONT, they discourage creativity on the part of the user, at least with respect to programming extensions to the programs themselves.

LATEX presents a similar problem. The more macros you use, the more likely it is that a macro you write will cause an unforeseen problem, especially if you don't understand how the macros you're using work. Large packages offer functionality, which is not always needed, and you pay for it with increased run-time and a loss of flexibility. I used LATEX when I first started writing auxiliary programs, but I found that I spent most of my time trying to make it stop doing things that I didn't want. For this reason (among others), I recommend using `plain` TEX, and the other formats and macros documented in *The TEXbook*, as the basis for programming extensions to TEX.

I've used some of the other possible combinations of TEX and auxiliary programs in other packages, which I plan to document in subsequent articles. Many of the techniques described in this article are of general applicability, not just for indexing. I hope that Spindex may inspire other TEX users to try writing an auxiliary program of their own.

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