Just give me a Lollipop (it makes my heart go giddy-up)*

Victor Eijkhout

Department of Computer Science
University of Tennessee at Knoxville
Knoxville TN 37996-1301
`eijkhout@cs.utk.edu`

Abstract

The Lollipop format is a meta-format: it does not define user macros, but it contains the tools with which a style designer can easily implement such user macros. This article will show some of the capabilities of Lollipop and will give the reader a small peek behind the scenes of the implementation.

TeX is intended to support higher-level languages for composition

Donald Knuth

1 Introduction

One of the reasons that TeX is not widely accepted outside the scientific world is that the effort needed to create new visual designs, or even to make minimal modifications of a given design ("this article is a bit too long, but since we have rather generous margins, why don't we put the title in the margin next to the abstract, instead of over it") is disproportionally large. In Eijkhout and Lenstra (1991) it was argued that one way of solving this problem would be to implement powerful tools that a style designer could use to program macros without ever programming in TeX itself. In effect, the style designer "needs only say what she wishes done" (Perlis) and the meta-format creates the macros that do this. This article describes such a meta-format: Lollipop[1].

Now, for those who wondered at the title of this article, the first half refers to an epigram by Alan Perlis, to be found on page 365 of *The TeXbook*; the second half derives from a sixties ditty by Millie Small. All other etymologies are erroneous, and severely frowned upon.

2 The Structure of Lollipop

The Lollipop format tries to provide tools that make programming macros as hard as using them. I will not discuss the use of the resulting macros in detail, but will focus on implementational matters.

2.1 Working with Lollipop.

In order to process a document in Lollipop there has to be a 'style definition' for that document. This definition, a sequence of Lollipop macro calls, can be in the document itself, it can be `\input`, or it can be contained in a format. The latter option of loading a style definition in Lollipop and dumping the result as a new format is encouraged for two reasons. First of all, it indicates better the separation between the work of the style designer and that of the user. Secondly—especially on old computers (say of the order of a 286)—processing the style definition for a complicated document can easily take one or two minutes.

2.2 The basic Lollipop macros.

The Lollipop format is partly a macro collection – and some of the more interesting utilities will be discussed below – and partly a tool box for defining macros. The tools are four macros for defining

- headings (`\DefineHeading`): the main characteristics of a heading are that it has a title, and that it should stay attached to the following text;
- lists (`\DefineList`): a list is characterized by the fact that it has items;
- text blocks (`\DefineTextBlock`): a text block is basically just a group, however, it is so general that lists and headings are really special cases of text blocks; and
- page grids (`\DefinePageGrid`): a page grid is (a macro that installs) an output routine.

Each of these macros[2] can have a large number of options.

---

*Presented at TUG '92, July 27–30, Portland, Oregon,USA.

[1] The Lollipop format is available for anonymous ftp from `cs.utk.edu`.

[2] There is in fact a fifth macro `\DefineExternalItem`, closely related to `\DefineList`; it will be treated below.

2.3 An example of the use of Lollipop.

Although a large number of examples would be necessary to give a representative sample of the possibilities of the Lollipop tools, here is one example to give the reader the basic idea. The following macro defines a heading `\SubSection`.

```
\DefineHeading:SubSection counter:i
    whitebefore:18pt whiteafter:15pt
    Pointsize:14 Style:bold
    block:start SectionCounter literal:,
          SubSectionCounter literal:.
          fillupto:levelindent title
    external:Contents title external:stop
    Stop
```

(The terms '`block`', '`external`' et cetera are called 'options'.) This definition specifies that subsections have a counter that counts in lowercase roman numerals, that there should be a certain amount of white space above and below it, and that it should be formatted in 14 point bold as the section counter, a comma, the subsection counter, a full stop, filling these counters up to the `\levelindent` width (to be explained below), and following this by the title. Also it specifies that the title should go to the contents file.

This macro `\DefineHeading` must be a pretty complicated object, don't you think? Well, here is the full definition[3]:

```
\@GenericConstruct{Heading}
\def\@DefineHeading{
    \@DefineStopCommand{\relax}
    \csarg\edef{\@name}{\@GEN@OPEN
        \the\@main@options@list
        \@GEN@CLOSE}
    }
```

where the auxiliary macro `\csarg` is defined as

```
\def\csarg#1#2{\expandafter#1%
    \csname#2\endcsname}
```

2.4 Definition of the `\Define` macros.

Since the `\Define...` macros are so much alike – many options are common to all of them – I let all of them be defined automatically by the same macro `\@GenericConstruct`. This defines `\DefineHeading` as a macro that will process a list of options (this part contains the common work for all constructs), and then call `\@DefineHeading` to do the actual definition.

A call `\DefineHeading:Section` will expand first of all to a call

```
\def\@name{Section}
```

As can be seen in the example above, this macro is then used to define `\Section` with an `\edef`. This `\edef` unpacks the token list `\@main@option@list` that has been constructed during option processing. Also, the macros `\@GEN@OPEN` and `\@GEN@CLOSE` contain lots of conditionals that may or may not cause code to be included in the definition of `\Section` depending on values of parameters that were set during option processing. This is explained further below.

2.5 Options.

Clearly, a large responsibility rests on processing the options. For instance, in the example above the option '`counter`' has to allocate the appropriate counter, but also set the test `\has@countertrue`.

Options can be general, such as the '`counter`' option (here `\xp` is `\expandafter`):

```
\@GenericOption{counter}{\has@counteryes
    \NewCounter:\@name
    \xp\add@mark@item\xp{\@name Counter}
    \CounterRepresentation:\@name=#1
    }
```

or they can be specific, such as the option controlling white space between items in a list:

```
\@ListOption{whitebetween}{....}
```

Generic options are defined as follows:

```
\def\@GenericOption#1{
    \append@to@list
        {@GenericOptions}{\\#1;}
    \csarg\def{Option@#1}##1##2}
```

for instance, for '`counter`' a macro `\Option-@counter` is defined. The definition

```
\@GenericConstruct{List}
```

causes the definition of `\@ListOption`:

```
\def\@GenericConstruct#1{
    \csarg\def{@#1Option}##1%
        {\csarg\def{#1@##1}####1####2}
```

so that the '`whitebetween`' option causes the definition of a macro `\@List@whitebetween`.

Now let's say we are defining a heading, and we find the option '`foo`'. We then check whether a macro `\Heading@foo` is defined. If so, we execute it; if not we check for the existence of a more general macro `\Option@foo`. This is executed if it exists, and if it doesn't, we check whether `\foo` is a defined control

[3] Several pieces of code in this article have been simplified. Others however, such as the following, have been left intact to convey to the reader the idea that Lollipop is a sophisticated format.

sequence. If it is, we include the command `\foo` in the `\@main@options@list`, so that it will later be part of the definition of the heading we are defining; if it is not, we generate an error message.

3 The Basic Tools

In this section I will give a short overview of the capabilities of the four basic macros. First the block structure macros used in all of them are explained briefly.

3.1 Block structure.

Text blocks and lists are obvious candidates for environment macros that do grouping, so that values of `\leftskip`, `\parindent`, and whatever more can stay local. As I've argued in (Eijkhout, 1990) such macros can also handle spacing above and below the environment. Thus, Lollipop has two macros

```
\def\@GEN@OPEN{\outer@start@commands
    \begingroup \inner@start@commands}
\def\@GEN@CLOSE{\inner@end@commands
    \endgroup \outer@end@commands}
```

that induce grouping, and that perform the various actions needed at the boundaries of an environment. This also includes such common actions as handling counters and titles, placing marks, and defining labels for symbolic referencing.

For instance, if the macro currently being defined (if this is `\Section`, the macro `\@name` has that value) has a counter, that should be incremented. Therefore the macro `\@GEN@OPEN` contains a line

```
\ifhas@counter
    \noexpand\StepCounter:\@name
\fi
```

Recall that these macros are called inside an `\edef`, so `\Section` macro contains the line

```
\StepCounter:Section
```

only if the macro has indeed a counter.

In general, a macro `\foo` opening the environment will contain the code generated by `\@gen@open`, while a corresponding command `\foostop` contains the `\@gen@close` code.

Headings

Maybe somewhat surprisingly, a heading can be considered as an environment, namely as one where the heading command contains both the opening and closing commands of the environment. Titles are treated below.

Text blocks

Text blocks are environments that can span several paragraphs. They have explicit open and close commands. Text blocks are, for instance, a way of having a chunk of text be indented and perhaps labeled. As an example, here is the specification of the examples in (Eijkhout, 1992): they are indented, and the word 'Example' is set in italic over them.

```
\DefineTextBlock:example
    breakbefore:500 breakafter:1
    PushListLevel
    noindent begingroup Style:italic
        literal:Example endgroup
    par nobreak Indent:no
    text
    Stop
```

The option '`text`' indicates where the text of the block fits in the specification. Any options appearing after this option will result in code in the macro that closes the environment. For instance, here is a possible way of defining left-aligning display equations:

```
\DefineTextBlock:DisplayEq
    whitebefore:abovedisplayskip
    whiteafter:belowdisplayskip
    whiteleft:levelindent
    literal:$ displaystyle text
    literal:$
    Stop
```

The closing macro will be defined as

```
\def\DisplayEqstop{ ...
    $
    \endgroup ... }
```

where the dollar corresponds to the one after the '`text`' option.

Lists

The main point of interest about lists is the formatting of the item labels. The two main choices are

```
    item:left ... item:stop
```

for left-aligning, and

```
    item:right ... item:stop
```

for right-aligning labels. The label can for instance contain an '`itemCounter`', or an '`itemSign`', or even both. The item sign and the representation of the item counter are dependent on the level, and can be set by the designer.

An interesting option is '`description`'. If this option is used, all text following `\item` to the end of line will be taken as the label text. The LaTeX style description list can be implemented as

```
    item:left Style:bold
      description Spaces:2
      item:stop
```

which is used as

```
\item The label
 and the next line is again normal
```

Abbreviated closing

Both lists and text blocks have an explicit closing command. Since such phenomena are properly nested, the format can very well figure out what to close if I tell it to close the current block. Therefore, the macro `\>` closes whatever list or text block is opened last, and `\>]` closes all lists and text blocks that are currently open.

Page grids

Definition of output routines is much easier in Lollipop than in plain TeX, but still it is the hardest part of working with Lollipop. Hence I will not go into full detail.

The most important option for page grids is '`text`'. It indicates that a page will use part of `\box255`. If this option does not appear, we are defining an output routine that does not use `\box255`. For such output routines the option `\nextpagegrid` is crucial: it tells TeX what output routine to take when the current one has output a page.

For instance, if left and right hand pages have a different layout, we could implement them as separate output routines:

```
\DefinePageGrid:leftpage
    nextpagegrid:rightpage
    ...
\DefinePageGrid:rightpage
    nextpagegrid:leftpage
    ...
```

The '`text`' option usually appears inside

```
    band:start text band:end
```

and it can occur several times there. For instance

```
    band:start text
      white:20pt text
      band:stop
```

defines a two column layout with a gutter width of 20 point.

Some of the options for page grid (height and width for instance) have a global significance, but for others it is recorded whether they appear before or after the '`text`' options. Depending on this, they become part of the header or the footer of the page.

When the output routine is invoked, Lollipop assembles any header or footer, and computes the remaining space for text. If this is not equal to the size of `\box255`, `\vsize` is reset, and `\box255` is thrown back to the main vertical list. This mechanism is an easy way to get pages with the same size if the size of the header or footer can vary.

Definition of output routines is in fact so easy in Lollipop that for title pages of chapters it is easier to write a special page grid, than to mess around with a lot of macros. Thus the line

```
\Chapter The second chapter\par
```

may look to the user as calling a macro, whereas in fact it installs a new output routine for the chapter title page. The way the title is handled is explained below.

4 Titles and References

The perceptive reader may have noticed in the definition of `\DefineHeading` above that the macro defined is not declared with a parameter. How then are titles handled?

Well, since in Lollipop not only headings, but also lists, text blocks, and page grids can have titles (but need not; every once in a while a heading without a title can be convenient, and output routines with titles are surprisingly useful, as I indicated above), the option '`title`' controls whether a construct actually has a title by setting a switch `\ifhas@title` to true. Definition of the actual heading macro then executes a line

```
\ifhas@title \@Titelize{\@name}\fi
```

where `\@Titelize` is a macro that takes a macro, and redefines it with an argument.

This redefinition trick can even be performed twice: if the macro has a counter, this should be referenceable. For some reason I decided against the LaTeX approach of using `\label` commands: any command that can be referenced in Lollipop[4] accepts an optional parameter with the label key. For instance

```
\Section[definition:section] Notations
    and Definitions\par
```

gives the key '`definition:section`' to a section.

5 Indentation Levels

If lists of various types are used in a nested fashion, each next level is indented with respect to the previous one by a certain amount. Specifying these amounts can be done quite flexibly in Lollipop, and it is also made easy for the designer to have other indented material line up with these implied left margins (Braahms, Eijkhout and Poppelier, 1989).

[4] Not explained in this article is that the way something is referenced is also easily determined by the user. This makes it possible for instance to refer to chapters by name instead of by number.

On each level, a control sequence `\levelindent` indicates the amount by which the next level will be indented. Thus, letting `\parindent` be set equal to `\levelindent` at the start of a text block, will give nicely aligning indentations no matter at what level the block appears.

The value of `\levelindent` is determined by looking at the level number (say that this is 3), and checking whether a macro `\levelindentiii` exists. If so, the value of this is taken, if not, some default fraction of the value of `\basicindent` is taken. The style designer can set this `\basicindent`, and adjust individual levels by

```
\LevelIndent:3=25pt
```

or similar calls.

Lists are indented to the next level automatically, but in order to provide this functionality for other objects there exists an explicit

```
\PushListLevel
```

command. There is even a `\PopListLevel` command that has various uses. For instance, it can be used to realise 'suspended lists': the effect of

```
\item Some text\par
{\PopListLevel
\noindent Some text.\par}
\item Again an item
```

is that the 'some text' aligns with the text outside the list, instead of with the items in the list.

Popping and pushing list levels is also essential for correct formatting of external files; see below.

6 Marks

TeX's marks are a means of communication between routines that supply certain information (values of counters, titles), and the output routine. Since there is no way for the output routine to tell the rest of the macros which ones should pass information through marks, in Lollipop *everyone* puts their information (that is, titles and counter values) in marks. The output routine then selects with a simple call, for instance

```
\LastPlaced:SectionTitle
```

the value of `\SectionTitle` in the `\botmark`.

Let's look at the implementation of this. There is a list `\mark@items` that has the names of everything that goes in a mark. For instance, defining a heading `\Section` causes calls

```
\add@mark@item{SectionTitle}
\add@mark@item{SectionCounter}
```

These allocate token lists

```
\mark@toks@SectionTitle
\mark@toks@SectionCounter
```

which are to contain the title and the counter value, and which get their value from a command such as

```
\refresh@mark@item
   {SectionTitle}{The title}
```

whenever `\Section` is called. Everytime a mark item is refreshed, a new mark is placed on the page which contains the values of *all* mark token lists. The output routine then simply picks from this structure whatever information it needs.

7 External Files

Formats such as LaTeX usually supply facilities for a table of contents, and maybe lists of figures and tables, but what if an author needs in addition a list of notations, one of definitions, and one of authors referenced? Lollipop takes the drastic approach, and provides none of these.

But it makes it easy for you to define them yourself.

7.1 User interface.

An external file is characterized by in an internal name, and a file name extension:

```
\DefineExternalFile:Contents=toc
```

This command does some initialization such as a call to `\newwrite`, and it creates a switch so that the user can specify with

```
\WriteContents:no
```

that the file is not to be overwritten in this run. A global switch

```
\WriteExtern:no
```

prohibits all external writes.

Next, commands such as `\Section` have to specify that they want to write out information. This is done with the option '`external`'. Usually, all that is written out is the title

```
    external:contents title external:stop
```

but other information can be written out too.

The hard part about external files is specifying how their information is to be typeset. Telling that a file needs to be loaded is simple:

```
\LoadExternalFile:Contents
```

For every command that writes information to an external file, the style definition needs to contain a call

```
\DefineExternalItem:Section file:Contents
    item:left Style:roman SectionLabel
        item:stop
    title Spaces:2 Style:italic Page
    Stop
```

where '`SectionLabel`' is the counter that was written out automatically for `\Section`, and '`title`' is whatever information was specified with the '`external`' option.

In effect, this defines the layout of a list that has only one item. Now we see another use for pushing indentation levels: contents items for subsections may need to be indented, but since they are a separate list on the outer level, we need to push them explicitly to the correct indentation:

```
\DefineExternalItem:SubSection
    file:Contents
    PushIndentLevel Style:roman
    item:right SectionLabel literal:.
        SubSectionLabel Spaces:1 item:stop
    title Spaces:2 Style:italic Page
    Stop
```

Note that a composite label is made here out of the section and subsection numbers.

7.2 Implementation.

External files are handled in much the same way they are treated in LaTeX: all information is written to the main auxiliary file, and this is loaded at the end of the run, in order to update the other external files.

Writing out titles and such means that these are subject to the usual expansion of `\write`. The LaTeX approach of letting the user put `\protect` commands has proved over time to be too error-prone, so I've decided to inhibit all expansion in titles.

8 Extendability of Lollipop

For each macro package, the question comes up 'but what if I want something that it cannot do?' The option mechanism of Lollipop can cope with this quite easily. Any option that is undefined is interpreted as a control sequence. Thus the style designer can incorporate arbitrary macros.

For instance, the title pages of *TEX by Topic* (Eijkhout, 1992) have quite elaborate headings, for which I programmed a separate macro `\ChapterHead`, which uses the (automatically generated) macro `\ChapterTitle`.

```
\def\ChapterHead
   {\hbox{ ...
         \PointSize:24 \Style:roman
         \ChapterTitle
    ...}
```

The macro `\Chapter` then uses this `\ChapterHead`:

```
\DefinePageGrid:Chapter
    NextPageGrid:textpage HasTitle:yes
    ...
    ChapterHead
    ... Stop
```

The undefined option '`ChapterHead`' generates a call to the macro `\ChapterHead`.

9 Goodies

It goes without saying that Lollipop has a sophisticated font selection scheme, a verbatim mode, and other assorted niceties. However, since these facilities are rather pedestrian, if rather useful, I will not discuss them here.

10 Conclusion

Lollipop is a long, complicated format. An article about it can only give a taste of its philosophy. I hope this piece has given the reader an idea of how macros can be generated automatically, according to the wishes of a style designer. People wanting to use Lollipop can get the software and a user's guide; people wanting to understand it will have for a while only this article and the code to go on.

As yet there is no real experience with Lollipop. I have used it myself for two books, but I am the author. I find it very easy to use, but if something goes wrong the errors can be mystifying in the extreme.[5] Error messages are still a major concern. Recall that macros are automatically defined and redefined, by macros that are themselves never explicitly defined. Still, I hope that the dynamic approach will catch enough user mistakes already in the definition stage for this format to be of value to non-TEXnicians wishing to use TEX.

References

[1] Braams, Johannes, Victor Eijkhout, and Nico Poppelier, "The development of national LaTeX styles", *TUGboat*, **10**(3), pages 401–406, 1989.

[2] Eijkhout, Victor, *A paragraph skip scheme*, *TUGboat*, **11**(4), pages 616–619, 1990.

[3] Eijkhout, Victor, *TEX by Topic*, Addison-Wesley, 1992.

[4] Eijkhout, Victor and Andries Lenstra. "The document style designer as a separate entity", *TUGboat*, **12**(1), pages 31–34, 1991.

[5] Perlis, Alan, "Epigrams on Programming", *ACM Sigplan Notices*, **17** (9), pages 7–13, 1982.

[5] And the macros themselves can become pretty big. While debugging, I discovered that TEX will 'only' `\show` the first 1000 characters of a macro...

Index Preparation for TEX Related Documents*

David Salomon

11835 Sapota Dr.,
Lakeside, CA 92040, USA
`dxs@ms.secs.csun.edu`

June 1992

A beta release of the `MakeIndex` program has recently become available for the Macintosh computer, and I immediately started using it to prepare the indexes of two new books. `MakeIndex` is easy to use with LATEX but, since I like to work with `plain` TEX, I have developed all the necessary macros from scratch. They are presented here for the benefit of anyone who wants a professionally looking index.

A few words about the use of `MakeIndex` are in order. Either LATEX or TEX is used to create a raw index file (the IDX file) with entries such as `\indexentry{abc}{24}`, `\indexentry{xyz}{108}`. `MakeIndex` then reads the file, sorts the items, merges multiple occurences of the same item on the same page, and creates the final, IND, file with entries such as '`\item abc, 11, 24, 101`' '`\item xyz 15, 76, 108`'. The user has to define macro `\item` to typeset an index item in any desired way, and the entire index is then typeset by `\input doc.ind`. `MakeIndex` supports sub- and subsub items (indicated by a '!'), and has many other features (see appendix 1).

It is generally agreed that index preparation is a complex job, and that it should not be fully automated. Certain things are best done manually. My original intent was, therefore, to develop simple macros that are easy to read, understand and modify for specific needs, yet can do most of the work. My experience so far indicates that they typically do more than 90% of the work, so only a small part of the total effort of index preparation needs to be done manually.

The macros described here are divided into two groups, those that write index items on the IDX file, and those that read the final, IND file, and typeset the final index. Following the tradition of *The TEXbook* (p. 423), the circumflex is turned into an active character and is used to indicate index items in the source document. A typical example is `^{abc}`. The macros have to take into account the following:

1. Some index items are 'silent', they should be written on the IDX file but should not appear in the document. The macros accordingly accept either 1 or 2 arguments of the forms `^{abc}` and `^[xyz]{abc}` (but not `^{abc}[xyz]`). The string `xyz` is not typeset but becomes part of the index item on the IDX file.
2. Many documents about TEX use the notation |`abc`| for verbatim listings. The user should thus be allowed to write `^`|`abc`| (but not `^`|`abc`|`{..}`). In order that they not interfere with the sorting, the vertical bars should be removed from the IDX file, and be reinserted in the IND file.
3. The notation |`\abc`| presents another special case. The user should be allowed to write `^`|`\abc`|, and both the vertical bars and the '\' should be removed before sorting. However, because of the special way macro `\getpar` below reads its argument, the removed items cannot simply be reinstated. The solution is to convert such an index item, in the IND file, to the form `\bs abc\\`. When the IND file is `\input`, macro `\bs` adds the '\' and typesets its argument verbatim.
4. In a document about TEX, certain index items are preceded by a '\'. A good example is the item `\TeX`. This item should be sorted without the '\', to appear among the Ts. The '\' should thus be removed from this item in the IDX file prior to sorting, and should be appended back to the same item on the IND file before typesetting it.
5. Certain characters, such as '`$`' and '`%`', have special meaning in TEX. Since an index item may contain such characters, their special meaning should be temporarily suppressed when writing an item on the IDX file.

Point 5 above is handled by the `\sanitize` macro which changes the catcodes of most special characters to 'other'. Since '`^`' is used to indicate an index item, the catcodes of `^`, `^^K` and `^^A` are not changed. `\sanitize` should be expanded before the arguments of the index macros are scanned. The only macros with arguments are `\inxA#1` & `\inxB[#1]#2`. Note below how `\begingroup` and `\endgroup` are used to localize the effect of `\sanitize`.

*Presented at the 9^e NTG meeting, June 4, 1992, Amsterdam.

As a result, things such as '`\insert`', '`#52`' and '`52%`' can easily be written on the index file, while '`J\^orgen`' cannot.

Table 1 shows examples of index items. For each item, the way it is specified in the source file is shown, as well as the resulting text in the document, and the way the item is finally typeset in the index.

<table>
<tr><th>Source</th><th>Typeset in document</th><th>Typeset in index</th></tr>
<tr><td>^[abc]</td><td>abc</td><td>abc</td></tr>
<tr><td>^[xyz]{abc}</td><td>abc</td><td>xyzabc</td></tr>
<tr><td>^[xyz]{ abc}</td><td>abc</td><td>xyz abc</td></tr>
<tr><td>^|abc|[1]</td><td>abc</td><td>abc</td></tr>
<tr><td>^|\abc|[1]</td><td>\abc</td><td>\abc</td></tr>
<tr><td>^[|\abc |]{xyz}</td><td>xyz</td><td>\abc xyz</td></tr>
<tr><td>^{\abc}</td><td>replacement text of \abc</td><td>same</td></tr>
<tr><td>^[|\abc|!xyz]{}[2]</td><td>nothing</td><td>\abc, xyz[3]</td></tr>
</table>

[1]Note that there is no need for braces in this case
[2]Main argument should be empty.
[3]This is a subitem.

Table 1

To handle the different cases above, a small utility program, `IdxInd`, has been written. It should be run on the IDX file before `MakeIndex` reads it, and on the IND file after it is created by `MakeIndex`. The entire process involves the following steps:

- The document is typeset by TEX and file `doc.idx` is prepared.
- `IdxInd` reads file `doc.idx` and creates file `tmp.idx`. This is called the first pass.
- `MakeIndex` is run on `tmp.idx`. It creates the sorted file `tmp.ind`.
- The first and last lines of the IND file are `\begin{index}` and `\end{index}`. They should be removed manually, since they are only used by LATEX.
- `IdxInd` is run on `tmp.ind` and creates file `doc.ind`. This is the second pass.
- File `doc.ind` is `\input` and the indexing macros (the second set below) used to typeset the final index. Any special index items will show up in this step and will require manual intervention. Also, bad page breaks in the index pages will have to be corrected by the user at this stage, either by changing the penalties used in the macros, or by inserting penalties at strategic points in file `doc.ind`.

Table 2 shows how different types of index items appear in the source file and in the four index files involved.

<table>
<tr><th>type</th><th>in source file</th><th colspan="4">on index file</th></tr>
<tr><th></th><th></th><th>doc.idx</th><th>tmp.idx</th><th>tmp.ind</th><th>doc.ind</th></tr>
<tr><td>1</td><td>^[abc]{}</td><td>abc</td><td>abc</td><td>abc</td><td>abc</td></tr>
<tr><td>2</td><td>^|abc|</td><td>|abc|</td><td>abc*</td><td>\item abc*,..</td><td>\item |abc|</td></tr>
<tr><td>3</td><td>^|\abc|</td><td>|\abc|</td><td>abc#</td><td>\item abc#,..</td><td>\item \bs abc\\</td></tr>
<tr><td>3 (with sub item)</td><td>^[|\abc|!xyz]{}</td><td>|\abc|!xyz</td><td>abc#!xyz</td><td>\item abc#,..
\subitem xyz,..</td><td>\item \bs abc\\,..
\subitem xyz,..</td></tr>
<tr><td>4</td><td>{^\abc}</td><td>\abc</td><td>abc&</td><td>\item abc&..</td><td>\item \abc</td></tr>
<tr><td>4 with) sub item)</td><td>^[\abc!xyz]{}</td><td>\abc!xyz</td><td>abc&!xyz</td><td>\item abc&..
\subitem xyz,..</td><td>\item \abc
\subitem xyz,..</td></tr>
</table>

Table 2

Note that one should not have, in the same document, items such as `^[\abc]{}` and `^|\abc|`. They would be changed by pass 1 to `abc&` and `abc#`, respectively, and would therefore be considered different by `MakeIndex`. Instead of `^[\abc]{}` the author should write `^[|\abc|]{}`. It is also permissible to write `^[|\abc||bold]{}` (with three vertical bars). In such a case, `IdxInd` will only remove the first two bars (and the backslash).

The last example in table 2 is of a main item `|\abc|` and a subitem `xyz`. The entire item must be silent (placed in square brackets), and the pair of braces that follows should be empty.

When `IdxInd` removes a pair of vertical bars (or vertical bars and a '\', or just a '\') it stores a special code following the modified item, so that the removed characters can later be restored. In table 2 the codes shown are an '*', a '#' or an '&'. In reality, the program uses ASCII codes 04, 03 and 02, respectively, for this purpose. Those are the codes of control characters, and so don't show up in print.

Here is the first set of macros, those that write the index entries on the IDX file. Note the use of `\string\indexentry`. Using `\noexpand\indexentry` also works, but writes a space following `\indexentry`.

```
\newwrite\inx
\immediate\openout\inx=\jobname.idx

\def\Caret{\ifmmode\def\next{^}\else\let\next=\indexT\fi\next}
\catcode`\^=\active \let^=\Caret

\def\makeother#1{\catcode`#1=12\relax}
\def\sanitize{\makeother\ \makeother\\\makeother\$\makeother\&%
\makeother\#\makeother\_\makeother\%\makeother\~\makeother\|}

\def\indexT{\futurelet\new\macX}
\def\macX{\ifx\new[\let\Next=\inxB\else\let\Next=\inxA\fi
\begingroup\sanitize\Next}
\def\inxA#1{#1\finidx{#1}}
\def\inxB[#1]#2{#2\finidx{#1#2}}
\def\finidx#1{\gdef\wridx{\write\inx{\string\indexentry{#1}{\folio}}}%
\endgroup\wridx}
```

The document, with index items, should follow, ending with `\closeout\inx`.

Special index items: Control sequences and items in angle brackets should be handled in a special way. The control sequence `\abc` should be indexed as `^[\abc]{}` (silent). An item such as `<xyz>` should be indexed as `^[<xyz>]{}` (silent) or `^[<xyz>]{}\lr{xyz}` where `\lr` is a macro producing the brackets.

Since in practice we don't want the brackets and backslashes to participate in the sorting, we should move them to the right in the IDX file (thus `^[abc\]{}` `^[xyz><]{}`), run `MakeIndex`, and move them back to the left in the IND file. For most documents, such cases are rare, and can be handled manually. For documents on TEX, a program should be written to do this. For the index of *The TEXbook*, a special code was written on the raw index file (pp. 423–424) to indicate special items. After sorting, a special program was run to interpret the codes and add brackets, backslashes, etc.

The index itself is prepared in double-column format. The macros for double columns appear in *The TEXbook*, pp. 416–417 and should be personalized. Typically, the values of `\hsize` & `\vsize` should be changed, and the `\shipout` in macro `\onepageout` may have to be modified.

The second set of macros appears below. These are the indexing macros, which are deliberately kept simple, to make them easy to read and modify. They should be sufficient for most needs. Note the following:

- An item such as `^[auxiliary spaces|see{ties}]{}` should appear only once in the document (since no page numbers will be listed for it anyway). The macros shown here will work even if `^{auxiliary spaces}` appears several times, but the `see` must appear in the first occurence.
- An item such as `^[auxiliary spaces|seealso{ties}]{}` should, similarly, appear just once in the document, at the last time `auxiliary spaces` is indexed.

```
\newif\ifpagebreak\pagebreaktrue
\def\item{\begingroup\obeylines\getpar%
                \global\pagebreaktrue}
{\obeylines
\gdef\getpar#1
{\par\ifpagebreak\bigbreak\else\nobreak\fi
\hangafter=1 \hangindent=15pt #1\par
\global\leftskip=0pt \endgroup}}
\def\goodfil{\hfil\penalty-9\hfilneg}
               %similar to \filbreak [353]
\def\see#1#2{{\it see }#1} % If there is
                   % more than one \see or
% \seealso per item, manual intervention
% is necessary.
\def\seealso#1#2{#2; \goodfil{\it see
                               also }#1}
\def\bold#1{{\bf#1}}
\def\indexspace{\par \vskip 10pt plus 5pt %
                         minus 3pt\relax}
\def\subitem{\par\pagebreakfalse%
                      \leftskip=7.5pt\item}
\def\subsubitem{\par\pagebreakfalse%
                       \leftskip=15pt\item}
```

Assuming that the double-column macros (from *The TEXbook*, pp. 416–417) are available, the index can now be produced by:

```
\pageno=<whatever>
\line{\bf\hfil Index\hfil\hfil}\vskip.1in
A quotation can be placed here

\begindoublecolumns
\tolerance=6000 \parindent=0pt
\input mybook.ind
\enddoublecolumns
\bye
```

Exercise: An index is normally divided into 26 groups, each of items starting with the same letter. Modify

`\indexspace` to typeset the letter preceding each group.

Answer: If we can assume that all 26 letter groups are present, this is easy. The modified macro is:

```
\newcount\Letter\Letter='101
\def\indexspace{\par\vskip...%
                         \advance\Letter1%
             {\bf\char\Letter}\vskip...}
```

and an additional `\indexspace` command has to be placed manually in the IND file before the first group. If some groups may be missing, the simplest solution is to add a parameter to `\indexspace`:

```
\def\indexspace#1{\par\vskip...
                {\bf#1}\vskip...}
```

and supply the argument manually. A general solution may use the following idea: Macro `\indexspace` sets a flag (a `\newif` variable) to indicate that a letter should be typeset before the next item. Macro `\item` should test the flag. If the flag is true, `\item` should isolate the first character of its argument, typeset it (with appropriate vertical spacing) and clear the flag.

Appendix 1: MakeIndex

`MakeIndex` expects an IDX input file with entries of the form `\indexentry{entry}{page number}`. The following are the main options that it recognizes:

- A '!' can be used for sub items and subsub items. There are no subsubsub items. The IDX file entry `\indexentry{abc!opq!xyz}{23}` is converted by `MakeIndex` to the following three entries in the IND file:
  ```
  \item abc
  \subitem opq
  \subsubitem xyz, 23
  ```
  The user should define macros `\item`, `\subitem` and `\subsubitem` to typeset the index in any desired way.
- A vertical bar is used to indicate a command. The IDX file entry
  `\indexentry{abc|bold}{23}`
  is converted by `MakeIndex` to
  `\item abc \bold{23}` on the IND file. Again, the user should define a macro `\bold` to typeset the page number in the desired way. Other common examples are
  `\indexentry{abc|see{xyz}}{23}` and
  `\indexentry{abc|seealso{xyz}}{23}`.
- An '@' is used to specify a print entry that's different from the sort entry. The IDX file entry `\indexentry{eleven@xi}{23}` will result in '`\item xi, 23`' placed among the E's in the IND file.

Appendix 2: IdxInd

This small utility is written in Modula 2 for the Metrowerks PSE compiler on the Macintosh computer, and is freely available from the author. It has two separate parts, pass 1 and pass 2. In pass 1 it reads an IDX file, scans each record for special characters, replaces them with ASCII codes 02, 03 or 04, and writes the record on a new IDX file, to be processed by `MakeIndex`. After `MakeIndex` creates the IND file, pass 2 is run, to restore the special characters.

The program looks for a pair of vertical bars and for a backslash. It distinguishes three cases:

- An IDX file entry contains a pair of vertical bars. This corresponds to a type 2 record in table 2. The bars are removed, and an ASCII 02 in inserted at the back of the entry.
- The entry contains a pair of vertical bars, the first of which is immediately followed by a backslash. This corresponds to type 3 in table 2. The three characters are removed, and an ASCII 03 is inserted.
- A backslash but no vertical bars. This corresponds to type 4 in table 2. Again, the backslash is removed, and an ASCII 04 is inserted.

Pass 2 performs two tasks. The first is to look for the special ASCII characters and restore the vertical bars and/or backslash. The second task is to look for long, multi-line records, and convert them to a single line. An index item that appears on many pages in the document, may end up as a long record, with many page numbers, in the IND file. Such an entry is broken, by `MakeIndex`, into several lines. It may typically look like:

```
\item abc 3, 11, 18, ...
125, 153, 167,...
180, 242, 394, ...
```

Our simple macros expect each line to start with `\item`, `\subitem` or something similar. The three lines in the above example have thus to be united into a single record, and this is done in pass 2 by removing the carriage return characters at the end of the first two lines.