Expanded Plain TEX

May 1994
For version 2.6.

Karl Berry
Steven Smith

1 Introduction

The *Eplain* macro package expands on and extends the definitions in plain TEX. This manual describes the definitions that you, as either an author or a macro writer, might like to use. It doesn't discuss the implementation; see comments in the source code ('`xeplain.tex`') for that.

Eplain is not intended to provide "generic typesetting capabilities, as do LaTEX (written by Leslie Lamport) or Texinfo (written by Richard Stallman and others). Instead, it provides definitions that are intended to be useful regardless of the high-level commands that you use when you actually prepare your manuscript.

For example, Eplain does not have a command `\section`, which would format section headings in an "appropriate" way, such as LaTEX's `\section`. The philosophy of Eplain is that some people will always need or want to go beyond the macro designer's idea of "appropriate". Such canned macros are fine—as long as you are willing to accept the resulting output. If you don't like the results, or if you are trying to match a different format, you are out of luck.

On the other hand, almost everyone would like capabilities such as cross-referencing by labels, so that you don't have to put actual page numbers in the manuscript. The author of Eplain is not aware of any generally available macro packages that (1) do not force their typographic style on an author, and yet (2) provide such capabilities.

Besides such generic macros as cross-referencing, Eplain contains another set of definitions: ones that change the conventions of plain TEX's output. For example, math displays in TEX are, by default, centered. If you want your displays to come out left-justified, you have to plow through *The TEXbook* to find some way to do it, and then adapt the code to your own needs. Eplain tries to take care of the messy details of such things, while still leaving the detailed appearance of the output up to you.

Finally, numerous definitions turned out to be useful as Eplain was developed. They are also documented in this manual, on the chance that people writing other macros will be able to use them.

You can send bug reports or suggestions to `tex-eplain@cs.umb.edu`. The current version number of Eplain is defined as the macro `\fmtversion` at the end of the source file '`eplain.tex`'. When corresponding, please refer to it.

To get on this mailing list yourself, email '`tex-eplain-request@cs.umb.edu`' with a message whose body contains a line

`subscribe` *you@your.preferred.address*

2 Installation

The simplest way to install Eplain is simply to install the file '`eplain.tex`' in a directory where TEX will find it. What that directory is obviously depends on your operating system and TEX installation. I personally install '`eplain.tex`' in a directory '`/usr/local/lib/texmf/tex/plain`'.

If you want, you can also create a format ('`.fmt`') file for Eplain, which will eliminate the time spent reading the macro source file with `\input`. You do this by issuing a sequence of Unix commands something like this:

```
prompt$ touch eplain.aux
prompt$ initex
This is TeX, ...
**&plain eplain
(eplain.tex)
*\dump
... messages ...
```

You must make sure that '`eplain.aux`' exists *before* you run '`initex`'; otherwise, warning messages about undefined labels will never be issued.

You then have to install the resulting '`eplain.fmt`' in some system directory or set an environment variable to tell TEX how to find it. I install the format files in '`/usr/local/lib/texmf/ini`'; the environment variable for the Web2C port of TEX to Unix is `TEXFORMATS`.

Some implementations of TEX (including Web2C) use the name by which TEX is invoked to determine what format to read. For them, you should make a link to the '`virtex`' program named '`etex`', and then install the format file with the name '`etex.fmt`'. This lets users invoke TEX as '`etex`' and get the format file read automatically, without having to say '`&eplain`'.

For convenience, the file '`etex.tex`' in the distribution directory does `\input eplain` and then `\dump`, so that if you replace '`eplain`' with '`etex`' in the example above, the format file will end up with the right name.

The `install` target in the '`Makefile`' does all this properly for Unix systems and Web2C. You may have to change the pathnames.

Under emtex, '`eaj@acpub.duke.edu`' says that

```
tex386 -i ^&plain eplain \dump
```

produces a format file.

3 Invoking Eplain

The simplest way to use Eplain is simply to put:

```
\input eplain
```

at the beginning of your input file. The macro file is small enough that reading it does not take an unbearably long time—at least on contemporary machines.

In addition, if a format ('`.fmt`') file has been created for Eplain (see the previous section), you can eliminate the time spent reading the macro source file. You do this by responding `&eplain` or `&etex` to TeX's '`**`' prompt. For example:

```
initex
This is TeX, ...
**&eplain myfile
```

Depending on the implementation of TeX which you are using, you might also be able to invoke TeX as '`etex`' and have the format file automatically read.

If you write something which you will be distributing to others, you won't know if the Eplain format will be loaded already. If it is, then doing `\input eplain` will waste time; if it isn't, then you must load it. To solve this, Eplain defines the control sequence `\eplain` to be the letter `t` (a convention borrowed from Lisp; it doesn't actually matter what the definition is, only that the definition exists). Therefore, you can do the following:

```
\ifx\eplain\undefined \input eplain \fi
```

where `\undefined` must never acquire a definition.

Eplain consists of several source files:

'`xeplain.tex`'
: most of the macros;

'`arrow.tex`'
: commutative diagram macros, see Chapter 5 [Arrow theoretic diagrams], page 28 (written by Steven Smith);

'`btxmac.tex`'
: bibliography-related macros, see Section 4.3 [Citations], page 6;

'`texnames.sty`'
: abbreviations for various TeX-related names, see Section 4.19 [Logos], page 26 (edited by Nelson Beebe).

The file '`eplain.tex`' is all of these files merged together, with comments removed.

All of these files except '`xeplain.tex`' can be input individually, if all you want are the definitions in that file.

Also, since the bibliography macros are fairly extensive, you might not want to load them, to conserve TeX's memory. Therefore, if the control sequence `\nobibtex` is defined, then the bibliography definitions are skipped. You must set `\nobibtex` before '`eplain.tex`' is read, naturally. For example, you could start your input file like this:

```
\let\nobibtex = t
\input eplain
```

By default, `\nobibtex` is undefined, and so the bibliography definitions *are* made.

Likewise, define `\noarrow` if you don't want to include the commutative diagram macros from '`arrow.tex`', perhaps because you already have conflicting ones.

If you don't want to read or write an '`aux`' file at all, for any kind of cross-referencing, define `\noauxfile` before reading '`eplain.tex`'. This also turns off all warnings about undefined labels.

Eplain conflicts with AMSTEX (more precisely, with '`amsppt.sty`') The macros `\cite` and `\ref` are defined by both.

If you want to use AMSTEX's `\cite`, the solution is to define `\nobibtex` before reading Eplain, as described above.

If you have '`amsppt.sty`' loaded and use `\ref`, Eplain writes a warning on your terminal. If you want to use the AMSTEX `\ref`, do `\let\ref = \amsref` after reading Eplain. To avoid the warning, do `\let\ref = \eplainref` after reading Eplain and before using `\ref`.

4 User definitions

This chapter describes definitions that are meant to be used directly in a document. When appropriate, ways to change the default formatting are described in subsections.

4.1 Diagnostics

Plain TEX provides the `\tracingall` command, to turn on the maximum amount of tracing possible in TEX. The (usually voluminous) output from `\tracingall` goes both on the terminal and into the transcript file. It is sometimes easier to have the output go only to the transcript file, so you can peruse it at your leisure and not obscure other output to the terminal. So, Eplain provides the command `\loggingall`. (For some reason, this command is available in Metafont, but not in TEX.)

It is also sometimes useful to see the complete contents of boxes. `\tracingboxes` does this. (It doesn't affect whether or not the contents are shown on the terminal.)

You can turn off all tracing with `\tracingoff`.

You can also turn logging on and off globally, so you don't have to worry about whether or not you're inside a group at the time of command. These variants are named `\gloggingall` and `\gtracingall`.

Finally, if you write your own help messages (see `\newhelp` in *The TEXbook*), you want a convenient way to break lines in them. This is what TEX's `\newlinechar` parameter is for; however, plain TEX doesn't set `\newlinechar`. Therefore, Eplain defines it to be the character `^^J`.

For example, one of Eplain's own error messages is defined as follows:

```
\newhelp\envhelp{Perhaps you forgot to end the previous^^J%
   environment? I'm finishing off the current group,^^J%
   hoping that will fix it.}%
```

4.2 Rules

The default dimensions of rules are defined in chapter 21 of the *The TEXbook*. To sum up what is given there, the "thickness" of rules is 0.4pt by default. Eplain defines three parameters that let you change this dimension: `\hruledefaultheight`, `\hruledefaultdepth`, and `\vruledefaultwidth`. By default, they are defined as *The TEXbook* describes.

But it would be wrong to redefine `\hrule` and `\vrule`. For one thing, some macros in plain TEX depend on the default dimensions being used; for another, rules are used quite heavily, and the performance impact of making it a macro can be noticeable. Therefore, to take advantage of the default rule parameters, you must use `\ehrule` and `\evrule`.

4.3 Citations

Bibliographies are part of almost every technical document. To handle them easily, you need two things: a program to do the tedious formatting, and a way to cite references by labels, rather than by numbers. The BibTEX program, written by Oren Patashnik, takes care of the first item; the citation commands in LaTEX, written to be used with BibTEX, take care of the second. Therefore, Eplain adopts the use of BibTEX, and virtually the same interface as LaTEX.

The general idea is that you put citation commands in the text of your document, and commands saying where the bibliography data is. When you run TEX, these commands produce output on the file with the same root name as your document (by default) and the extension '`.aux`'. BibTEX reads this file. You should put the bibliography data in a file or files with the extension '`.bib`'. BibTEX writes out a file with the same root name as your document and extension '`.bbl`'. Eplain reads this file the next time you run your document through TEX. (It takes multiple passes to get everything straight, because usually after seeing your bibliography typeset, you want to make changes in the '`.bib`' file, which means you have to run BibTEX again, which means you have to run TEX again...) An annotated example of the whole process is given below.

If your document has more than one bibliography—for example, if it is a collection of papers—you can tell Eplain to use a different root name for the '`.bbl`' file by defining the control sequence `\bblfilebasename`. The default definition is simply `\jobname`.

See the document *BibTEXing* (whose text is in the file '`btxdoc.tex`', which should be in the Eplain distribution you got) for information on how to write your `.bib` files. Both the BibTEX and the Eplain distributions contain several examples, also.

The `\cite` command produces a citation in the text of your document. The exact printed form the citation will take is under your control; see Section 4.3.1 [Formatting citations], page 7. `\cite` takes one required argument, a comma-separated list of cross-reference labels (see Section 4.9 [Cross-references], page 13, for exactly what characters are allowed in such labels). Warning: spaces in this list are taken as part of the following label name, which is probably not what you expect. The `\cite` command also produces a command in the `.aux` file that tells BibTEX to retrieve the given reference(s) from the `.bib` file. `\cite` also takes one optional argument, which you specify within square brackets, as in LaTEX. This text is simply typeset after the citations. (See the example below.)

Another command, `\nocite`, puts the given reference(s) into the bibliography, but produces nothing in the text.

The `\bibliography` command is next. It serves two purposes: producing the typeset bibliography, and telling BibTEX the root names of the `.bib` files. Therefore, the argument to `\bibliography` is a comma separated list of the `.bib` files (without the '`.bib`'). Again, spaces in this list are significant.

You tell BibTEX the particular style in which you want your bibliography typeset with one more command: `\bibliographystyle`. The argument to this is a single filename *style*, which tells BibTEX to look for a file *style*`.bst`. See the document *Designing BibTEX styles* (whose text is in the '`btxhak.tex`') for information on how to write your own styles.

Eplain automatically reads the citations from the `.aux` file when your job starts.

If you don't want to see the messages about undefined citations, you can say `\xrefwarningfalse` before making any citations. Eplain automatically does this if the `.aux` file does not exist. You can restore the default by saying `\xrefwarningtrue`.

Here is a TEX input file that illustrates the various commands.

```
\input eplain                    % Reads the .aux file.
Two citations to Knuthian works:
  \cite[note]{surreal,concrete-math}.
\beginsection{References.}\par   % Title for the bibliography.
\bibliography{knuth}             % Use knuth.bib for the labels.
\bibliographystyle{plain}        % Number the references.
\end                             % End of the document.
```

If we suppose that this file was named '`citex.tex`' and that the bibliography data is in '`knuth.bib`' (as the `\bibliography` command says), the following commands do what's required. ('`$` ' represents the shell prompt.)

```
$ tex citex        (produces undefined citation messages)
$ bibtex citex     (read knuth.bib and citex.aux, write citex.bbl)
$ tex citex        (read citex.bbl, still have undefined citations)
$ tex citex        (one more time, to resolve the references)
```

The output looks something like (because we used the `plain` bibliography style):

Two citations to Knuthian works: [2,1 note].

References

[1] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading, Massachusetts, 1989.

[2] Donald E. Knuth. *Surreal Numbers*. Addison-Wesley, Reading, Massachusetts, 1974.

See the BibTEX documentation for information on how to write the bibliography databases, and the bibliography styles that are available. (If you want your references printed with names, as in [Knu74], instead of numbered, the bibliography style is `alpha`.)

4.3.1 Formatting citations

You may wish to change Eplain's formatting of citations; i.e., the result of your `\cite` commands. By default, the citation labels are printed one after another, separated by commas and enclosed in brackets, using the main text font. Some formats require other styles, such as superscripted labels. You can accomodate such formats by redefining the following macros.

`\printcitestart`

`\printcitefinish`
: Eplain expands these macros at the begining and end of the list of citations for each `\cite` command. By default, they produce a '`[`' and '`]`', respectively.

`\printbetweencitations`
: If a `\cite` command has multiple citations, as in `\cite{acp,texbook}`, Eplain expands this macro in between each pair of citations. By default, it produces a comma followed by a space.

`\printcitenote`
: This macro takes one argument, which is the optional note to the `\cite` command. If the `\cite` command had no note, this macro isn't used. Otherwise, it should print the note. By default, the note is preceded with a comma and a space.

Here is an example, showing you could produce citations as superscripted labels, with the optional notes in parentheses.

```
\def\printcitestart{\unskip $^\bgroup}
\def\printbetweencitations{,}
\def\printcitefinish{\egroup$}
\def\printcitenote#1{\hbox{\sevenrm\space (#1)}}
```

4.3.2 Formatting bibliographies

You may wish to change Eplain's formatting of the bibliography, especially with respect to the fonts that are used. Therefore, Eplain provides the following control sequences:

`\biblabelwidth`
: This control sequence represents a `\dimen` register, and its value is the width of the widest label in the bibliography. Although it is unlikely you will ever want to redefine it, you might want to use it if you redefine `\biblabelprint`, below.

`\biblabelprint`
: This macro takes one argument, the label to print. By default, the label is put in a box of width `\biblabelwidth`, and is followed by an enspace. When you want to change the spacing around the labels, this is the right macro to redefine.

`\biblabelcontents`
: This macro also takes one argument, the label to print. By default, the label is printed using the font `\bblrm` (below), and enclosed in brackets. When you want to change the appearance of the label, but not the spacing around it, this is the right macro to redefine.

`\bblrm` The default font used for printing the bibliography.

`\bblem` The font used for printing the titles and other "emphasized" material.

`\bblsc` In some styles, authors' names are printed in a caps-and-small-caps font. In those cases, this font is used.

`\bblnewblock`
: This is invoked between each of the parts of a bibliography entry. The default is to leave some extra space between the parts; you could redefine it to start each part on a new line (for example). A part is simply a main element of the entry; for example, the author is a part. (It was LaTEX that introduced the (misleading, as far as I am concerned) term 'block' for this.)

`\biblabelextraspace`
: Bibliography entries are typeset with a hanging indentation of `\biblabelwidth` plus this. The default is `.5em`, where the em width is taken from the `\bblrm` font. If you want to change this, you should do it inside `\bblhook`.

`\bblhook` This is expanded before reading the `.bbl` file. By default, it does nothing. You could, for example, define it to set the bibliography fonts, or produce the heading for the references. Two spacing parameters must be changed inside `\bblhook`: `\parskip`, which produces extra space between the items; and `\biblabelextraspace`, which is described above. (By the way, `\hookappend` won't work with `\bblhook`, despite the names. Just use `\def`.)

If you are really desperate, you can also hand-edit the `.bbl` file that BibTEX produces to do anything you wish.

4.4 Displays

By default, TEX centers displayed material. (Displayed material is just whatever you put between `$$`'s—it's not necessarily mathematics.) Many layouts would be better served if the displayed material was left-justified. Therefore, Eplain provides the command `\leftdisplays`, which indents displayed material by `\parindent` plus `\leftskip`, plus `\leftdisplayindent`.

You can go back to centering displays with `\centereddisplays`. (It is usually poor typography to have both centered and left-justified displays in a single publication, though.)

`\leftdisplays` also changes the plain TEX commands that deal with alignments inside math displays, `\displaylines`, `\eqalignno`, and `\leqalignno`, to produce left-justified text. You can still override this formatting by inserting `\hfill` glue, as explained in *The TEXbook*.

4.4.1 Formatting displays

If you want some other kind of formatting, you can write a definition of your own, analogous to `\leftdisplays`. You need only make sure that `\leftdisplaysetup` is called at the beginning of every display (presumably by invoking it in TEX's `\everydisplay` parameter), and to define `\generaldisplay`.

`\leftdisplays` expands the old value of `\everydisplay` before calling `\leftdisplaysetup`, so that any changes you have made to it won't be lost. That old token list as available as the value of the token register `\previouseverydisplay`.

4.5 Time of day

TEX provides the day, month, and year as numeric quantities (unless your TEX implementation is woefully deficient). Eplain provides some control sequences to make them a little more friendly to humans.

`\monthname` produces the name of the current month, abbreviated to three letters.

`\fullmonthname` produces the name of the current month, unabbreviated (in English).

`\timestring` produces the current time, as in '1:14 p.m.'

`\timestamp` produces the current date and time, as in '23 Apr 64 1:14 p.m.'. (Except the spacing is slightly different.)

`\today` produces the current date, as in '23 April 1964'.

4.6 Lists

Many documents require lists of items, either numbered or simply enumerated. Plain TEX defines one macro to help with creating lists, `\item`, but that is insufficient in many cases. Therefore, Eplain provides two pairs of commands:

`\numberedlist` ... `\endnumberedlist`

`\orderedlist` ... `\endorderedlist`
: These commands (they are synonyms) produce a list with the items numbered sequentially, starting from one. A nested `\numberedlist` labels the items with lowercase letters, starting with 'a'. Another nested `\numberedlist` labels the items with roman numerals. Yet more deeply nested numbered lists label items with '`*`'.

`\unorderedlist` ... `\endunorderedlist`
: This produces a list with the items labelled with small black boxes ("square bullets"). A nested `\unorderedlist` labels items with em-dashes. Doubly (and deeper) nested unordered lists label items with '*'s.

The two kinds of lists can be nested within each other, as well.

In both kinds of lists, you begin an item with `\li`. An item may continue for several paragraphs. Each item starts a paragraph.

You can give `\li` an optional argument, a cross-reference label. It's defined to be the "marker" for the current item. This is useful if the list items are numbered. You can produce the value of the label with `\xrefn`. See Section 4.9 [Cross-references], page 13.

You can also say `\listcompact` right after `\numberedlist` or `\unorderedlist`. The items in the list will then not have any extra space between them (see Section 4.6.1 [Formatting lists], page 10). You might want to do this if the items in this particular list are short.

Here is an example:

```
\numberedlist\listcompact
\li The first item.
\li The second item.

The second paragraph of the second item.
\endnumberedlist
```

4.6.1 Formatting lists

Several registers define the spacing associated with lists. It is likely that their default values won't suit your particular layout.

`\abovelistskipamount, \belowlistskipamount`
: The vertical glue inserted before and after every list, respectively.

`\interitemskipamount`
: The vertical glue inserted before each item except the first. `\listcompact` resets this to zero, as mentioned above.

`\listleftindent, \listrightindent`
: `\listrightindent` is the amount of space by which the list is indented on the right; i.e., it is added to `\rightskip`. `\listleftindent` is the amount of space, *relative to*

`\parindent`, by which the list is indented on the left. Why treat the two parameters differently? Because (a) it is more useful to make the list indentation depend on the paragraph indentation; (b) footnotes aren't formatted right if `\parindent` is reset to zero.

The three vertical glues are inserted by macros, and preceded by penalties: `\abovelistskip` does `\vpenalty\abovelistpenalty` and then `\vskip\abovelistskip`. `\belowlistskip` and `\interitemskip` are analogous.

In addition, the macro `\listmarkerspace` is called to separate the item label from the item text. This is set to `\enspace` by default.

If you want to change the labels on the items, you can redefine these macros: `\numberedmarker` or `\unorderedmarker`. The following registers might be useful if you do:

`\numberedlistdepth, \unorderedlistdepth`
: These keep track of the depth of nesting of the two kinds of lists.

`\itemnumber, \itemletter`
: These keep track of the number of items that have been seen in the current numbered list. They are both integer registers. The difference is that `\itemnumber` starts at one, and `\itemletter` starts at 97, i.e., lowercase 'a'.

You can also redefine the control sequences that are used internally, if you want to do something radically different: `\beginlist` is invoked to begin both kinds of lists; `\printitem` is invoked to print the label (and space following the label) for each item; and `\endlist` is invoked to end both kinds of lists.

4.7 Verbatim listing

It is sometimes useful to include a file verbatim in your document; for example, part of a computer program. The `\listing` command is given one argument, a filename, and produces the contents of that file in your document. `\listing` expands `\listingfont` to set the current font. The default value of `\listingfont` is `\tt`.

You can take arbitrary actions before reading the file by defining the macro `\setuplistinghook`. This is expanded just before the file is input.

If you want to have line numbers on the output, you can say `\let\setuplistinghook = \linenumberedlisting`. The line numbers are stored in the count register `\lineno` while the file is being read. You can redefine the macro `\printlistinglineno` to change how they are printed.

You can produce in-line verbatim text in your document with `\verbatim`. End the text with `|endverbatim`. If you need a '`|`' in the text, double it. If the first character of the verbatim text is a space, use `| `. (`| ` will work elsewhere in the argument, too, but isn't necessary.)

For example:

```
\verbatim| ||\#%&!|endverbatim
```

produces ` |\#%&!`.

Line breaks and spaces in the verbatim text are preserved.

You can change the verbatim escape character from the default '`|`' with `\verbatimescapechar` *char*; for example, this changes it to '`@`'.

```
\verbatimescapechar \@
```

The backslash is not necessary in some cases, but is in others, depending on the catcode of the character. The argument to `\verbatimescapechar` is used as `\catcode `*char*, so the exact rules follow that for `\catcode`.

Because `\verbatim` must change the category code of special characters, calling inside a macro definition of your own does not work properly. For example:

```
\def\mymacro{\verbatim &#%|endverbatim}% Doesn't work!
```

To accomplish this, you must change the category codes yourself before making the macro definition. Perhaps `\uncatcodespecials` will help you (see Section 6.1 [Category codes], page 35).

4.8 Contents

Producing a table of contents that is both useful and aesthetic is one of the most difficult design problems in any work. Naturally, Eplain does not pretend to solve the design problem. Collecting the raw data for a table of contents, however, is much the same across documents. Eplain uses an auxiliary file with extension '`.toc`' (and the same root name as your document) to save the information.

To write an entry for the table of contents, you say `\writetocentry{`*part*`}{`*text*`}`, where *part* is the type of part this entry is, e.g., '`chapter`', and *text* is the text of the title. `\writetocentry` puts an entry into the `.toc` file that looks like `\toc`*part*`entry{`*text*`}{`*page number*`}`. The *text* is written unexpanded.

A related command, `\writenumberedtocentry`, takes one additional argument, the first token of which is expanded at the point of the `\writenumberedtocentry`, but the rest of the argument is not expanded. The usual application is when the parts of the document are numbered. On the other hand, the one-level expansion allows you to use the argument for other things as well (author's names in a proceedings, say), and not have accents or other control sequences expanded. The downside is that if you *want* full expansion of the third argument, you don't get it—you must expand it yourself, before you call `\writenumberedtocentry`.

For example:

```
\writenumberedtocentry{chapter}{A $\sin$ wave}{\the\chapno}
\writetocentry{section}{A section title}
```

Supposing `\the\chapno` expanded to '`3`' and that the `\write`'s occurred on pages eight and nine, respectively, the above writes the following to the `.toc` file:

```
\tocchapterentry{A $\sin$ wave}{3}{8}
\tocsectionentry{A section title}{9}
```

You read the `.toc` file with the command `\readtocfile`. Naturally, whatever `\toc... entry` commands that were written to the file must be defined when `\readtocfile` is invoked. Eplain has minimal definitions for `\tocchapterentry`, `\tocsectionentry`, and `\tocsubsectionentry`, just to prevent undefined control sequence errors in common cases. They aren't suitable for anything but preliminary proofs.

After reading the `.toc` file, `\readtocfile` opens the file for writing, thereby deleting the information from the previous run. You should therefore arrange that `\readtocfile` be called *before* the first call to a `\writetoc...` macro. On the other hand, if you don't want to rewrite the

.toc file, perhaps because you are only running TEX on part of your manuscript, you can set \rewritetocfilefalse.

By default, the '.toc' file has the root \jobname. If your document has more than one contents—for example, if it is a collection of papers, some of which have their own contents—you can tell Eplain to use a different root name by defining the control sequence \tocfilebasename.

In addition to the usual table of contents, you may want to have a list of figures, list of tables, or other such contents-like list. You can do this with \definecontentsfile{*abbrev*}. All of the above commands are actually a special case that Eplain predefines with

```
\definecontentsfile{toc}
```

The *abbrev* is used both for the file extension and in the control sequence names.

4.9 Cross-references

It is often useful to refer the reader to other parts of your document; but putting literal page, section, equation, or whatever numbers in the text is certainly a bad thing.

Eplain therefore provides commands for symbolic cross-references. It uses an auxiliary file with extension .aux (and the same root name as your document) to keep track of the information. Therefore, it takes two passes to get the cross-references right—one to write them out, and one to read them in. Eplain automatically reads the .aux file at the first reference; after reading it, Eplain reopens it for writing.

You can control whether or not Eplain warns you about undefined labels. See Section 4.3 [Citations], page 6.

Labels in Eplain's cross-reference commands can use characters of category code eleven (letter), twelve (other), ten (space), three (math shift), four (alignment tab), seven (superscript), or eight (subscript). For example, '(a1 $&^_' is a valid label (assuming the category codes of plain TEX), but '%#\{' has no valid characters.

You can also do symbolic cross-references for bibliographic citations and list items. See Section 4.3 [Citations], page 6, and Section 4.6 [Lists], page 10.

4.9.1 Defining generic references

Eplain provides the command \definexref for general cross-references. It takes three arguments: the name of the label (see section above for valid label names), the value of the label (which can be anything), and the "class" of the reference—whether it's a section, or theorem, or what. For example:

```
\definexref{sec-intro}{3.1}{section}
```

Of course, the label value is usually generated by another macro using TEX count registers or some such.

\definexref doesn't actually define *label*; instead, it writes out the definition to the .aux file, where Eplain will read it on the next TEX run.

The *class* argument is used by the \ref and \refs commands. See the next section.

4.9.2 Using generic references

To retrieve the value of the label defined via `\definexref` (see the previous section), Eplain provides the following macros:

`\refn{`*label*`}`

`\xrefn{`*label*`}`
: `\refn` and `\xrefn` (they are synonyms) produce the bare definition of *label*. If *label* isn't defined, issue a warning, and produce *label* itself instead, in typewriter. (The warning isn't given if `\xrefwarningfalse`.)

`\ref{`*label*`}`
: Given the class *c* for *label* (see the description of `\definexref` in the previous section), expand the control sequence `\`*c* `word` (if it's defined) followed by a tie. Then call `\refn` on *label*. (Example below.)

`\refs{`*label*`}`
: Like `\ref`, but append the letter '`s`' to the `\...word`.

The purpose of the `\...word` macro is to produce the word 'Section' or 'Figure' or whatever that usually precedes the actual reference number.

Here is an example:

```
\def\sectionword{Section}
\definexref{sec-intro}{3.1}{section}
\definexref{sec-next}{3.2}{section}
See \refs{sec-intro} and \refn{sec-next} ...
```

This produces 'See Sections 3.1 and 3.2 ...'

4.10 Page references

Eplain provides two commands for handling references to page numbers, one for definition and one for use.

`\xrdef{`*label*`}`
: Define *label* to be the current page number. This produces no printed output, and ignores following spaces.

`\xref{`*label*`}`
: Produce the text 'p. *page-number*', which is the usual form for cross-references. The *page-number* is actually *label*'s definition; if *label* isn't defined, the text of the label itself is printed.

4.10.1 Equation references

Instead of referring to pages, it's most useful if equation labels refer to equation numbers. Therefore, Eplain reserves a `\count` register, `\eqnumber`, for the current equation number, and increments it at each numbered equation.

Here are the commands to define equation labels and then refer to them:

`\eqdef{`*label*`}`
: This defines *label* to be the current value of `\eqnumber`, and, if the current context is not inner, then produces a `\eqno` command. (The condition makes it possible to use `\eqdef` in an `\eqalignno` construction, for example.) The text of the equation number is produced using `\eqprint`. See Section 4.10.1.1 [Formatting equation references], page 15.

  If *label* is empty, you still get an equation number (although naturally you can't reliably refer to it). This is useful if you want to put numbers on all equations in your document, and you don't want to think up unique labels.

`\eqdefn{`*label*`}`
: This is like `\eqdef`, except it always omits the `\eqno` command. It can therefore be used in places where `\eqdef` can't; for example, in a non-displayed equation. The text of the equation number is not produced, so you can also use it in the (admittedly unusual) circumstance when you want to define an equation label but not print that label.

`\eqref{`*label*`}`
: This produces a formatted reference to *label*. If *label* is undefined (perhaps because it is a forward reference), it just produces the text of the label itself. Otherwise, it calls `\eqprint`.

`\eqrefn{`*label*`}`
: This produces the cross-reference text for *label*. That is, it is like `\eqref`, except it doesn't call `\eqprint`.

Equation labels can contain the same characters that are valid in general cross-references.

4.10.1.1 Formatting equation references

Both defining an equation label and referring to it should usually produce output. This output is produced with the `\eqprint` macro, which takes one argument, the equation number being defined or referred to. By default, this just produces '`(`*number*`)`', where *number* is the equation number. To produce the equation number in a different font, or with different surrounding symbols, or whatever, you can redefine `\eqprint`. For example, the following definition would print all equation numbers in italics. (The extra braces define a group, to keep the font change from affecting surrounding text.)

```
\def\eqprint#1{{\it (#1)}}
```

In addition to changing the formatting of equation numbers, you might to add more structure to the equation number; for example, you might want to include the chapter number, to get equation numbers like '(1.2)'. To achieve this, you redefine `\eqconstruct`. For example:

```
\def\eqconstruct#1{\the\chapternumber.#1}
```

(If you are keeping the chapter number in a count register named `\chapternumber`, naturally.)

The reason for having both `\eqconstruct` and `\eqprint` may not be immediately apparent. The difference is that `\eqconstruct` affects the text that cross-reference label is defined to be, while `\eqprint` affects only what is typeset on the page. The example just below might help.

Usually, you want equation labels to refer to equation numbers. But sometimes you might want a more complicated text. For example, you might have an equation '(1)', and then have a variation several pages later which you want to refer to as '(1*)'.

Therefore, Eplain allows you to give an optional argument (i.e., arbitrary text in square brackets) before the cross-reference label to \eqdef. Then, when you refer to the equation, that text is produced. Here's how to get the example just mentioned:

```
$$...\eqdef{a-eq}$$
...
$$...\eqdef[\eqrefn{a-eq}*]{a-eq-var}$$
In \eqref{a-eq-var}, we expand on \eqref{a-eq}, ...
```

We use `\eqrefn` in the cross-reference text, not `\eqref`, so that `\eqprint` is called only once.

4.10.1.2 Subequation references

Eplain also provides for one level of substructure for equations. That is, you might want to define a related group of equations with numbers like '2.1' and '2.2', and then be able to refer to the group as a whole: "... in the system of equations (2)...".

The commands to do this are `\eqsubdef` and `\eqsubdefn`. They take one *label* argument like their counterparts above, and generally behave in the same way. The difference is in how they construct the equation number: instead of using just `\eqnumber`, they also use another counter, `\subeqnumber`. This counter is advanced by one at every `\eqsubdef` or `\eqsubdefn`, and reset to zero at every `\eqdef` or `\eqdefn`.

You use `\eqref` to refer to subequations as well as main equations.

To put the two together to construct the text that the label will produce, they use a macro `\eqsubreftext`. This macros takes two arguments, the "main" equation number (which, because the equation label can be defined as arbitrary text, as described in the previous section, might be anything at all) and the "sub" equation number (which is always just a number). Eplain's default definition just puts a period between them:

```
\def\eqsubreftext#1#2{#1.#2}%
```

You can redefine `\eqsubreftext` to print however you like. For example, this definition makes the labels print as '2a', '2b', and so on.

```
\newcount\subref
\def\eqsubreftext#1#2{%
  \subref = #2           % The space stops a <number>.
  \advance\subref by 96  % 'a' is character code 97.
  #1\char\subref
}
```

Sadly, we must define a new count register, `\subref`, instead of using the scratch count register `\count255`, because '`#1`' might include other macro calls which use `\count255`.

4.11 Indexing

Eplain provides support for generating raw material for an index, and for typesetting a sorted index. A separate program must do the actual collection and sorting of terms, because TEX itself has no support for sorting.

Eplain's indexing commands were designed to work with the program MakeIndex, available from '`ftp.math.utah.edu`' in the directory '`pub/tex/makeindex`', and from CTAN hosts in '`tex-archive/indexing/makeindex`'; MakeIndex is also commonly included in prepackaged TeX distributions. It is beyond the scope of this manual to explain how to run MakeIndex, and all of its many options. See section "MAKEINDEX" in *MakeIndex*.

The basic strategy for indexing works like this:

1. For a document '`foo.tex`', Eplain's indexing commands (e.g., `\idx`; see the section 'Indexing terms' below) write the raw index material to '`foo.idx`'.
2. MakeIndex reads '`foo.idx`', collects and sorts the index, and writes the result to '`foo.ind`'.
3. Eplain reads and typesets '`foo.ind`' on a subsequent run of TeX. See the section 'Typesetting an index' below.

If your document needs more than one index, each must have its own file. Therefore, Eplain provides the command `\defineindex`, which takes an argument that is a single letter, which replaces '`i`' in the filenames and in the indexing command names described below. For example,

```
\defineindex{m}
```

defines the command `\mdx` to write to the file '`foo.mdx`'. Eplain simply does `\defineindex{i}` to define the default commands.

4.11.1 Indexing terms

Indexing commands in Eplain come in pairs: one command that only writes the index entry to the '`.idx`' file (see above section), and one that also typesets the term being indexed. The former always starts with '`s`' (for "silent"). In either case, the name always includes '*I*`dx`', where *I* is the index letter, also described above. Eplain defines the index '`i`' itself, so that's what we'll use in the names below.

The silent form of the commands take a subterm as a trailing optional argument. For example, `\sidx{truth}[definition of]` on page 75 makes an index entry that will eventually be typeset (by default) as

truth
  definition of, 75

Also, the silent commands ignore trailing spaces. The non-silent ones do not.

4.11.1.1 Indexing commands

Here are the commands.

- `\sidx{`*term*`}[`*subterm*`]` makes an index entry for *term*, optionally with subterm *subterm*. `\idx{`*term*`}` also produces *term* as output. Example:

  ```
  \sidx{truth}[beauty of]
  The beauty of truth is \idx{death}.
  ```

- \sidxname`{`*First M.*`}{`*von Last*`}`[*subterm*] makes an index entry for '*von Last*`, `*First M.*'. You can change the '`, `' by redefining `\idxnameseparator`. \idxname`{`*First M.*`}{`*von Last*`}` also produces *First M. von Last* as output. (These commands are useful special cases of `\idx` and `\sidx`.) Example:

```
\sidxname{Richard}{Stark}
\idxname{Donald}{Westlake} has written many kinds of novels, under
almost as many names.
```

- `\sidxmarked\`*cs*`{`*term*`}[`*subterm*`]` makes an index entry for *term*`[`*subterm*`]`, but *term* will be put in the index as `\`*cs*`{term}`, but still sorted as just *term*. `\idxmarked\`*cs*`{`*term*`}` also typesets `\`*cs*`{term}`. This provides for the usual ways of changing the typesetting of index entries. Example:

```
\def\article#1{``#1''}
\sidxmarked\article{Miss Elsa and Aunt Sophie}
Peter Drucker's \idxmarked\article{The Polanyis} is a remarkable essay
about a remarkable family.
```

- `\sidxsubmarked{`*term*`}\`*cs*`{subterm}` makes an index entry for *term, subterm* as usual, but also puts *subterm* in the index as `\`*cs*`{term}`. `\idxsubmarked{`*term*`}\`*cs*`{subterm}` also typesets *term* `\`*cs*`{subterm}`, in the unlikely event that your syntax is convoluted enough to make this useful. Example:

```
\def\title#1{{\sl #1}}
\sidxsubmarked{Anderson, Laurie}\title{Strange Angels}
The \idxsubmarked{Anderson}\title{Carmen} is a strange twist.
```

The commands above rely on MakeIndex's feature for separating sorting of an index entry's from its typesetting. You can use this directly by specifying an index entry as *sort*`@`*typeset*. For example:

```
\sidx{Ap-weight@$A_\pi$-weight}
```

will sort as `Ap-weight`, but print with the proper math. The `@` here is MakeIndex's default character for this purpose. See section "Style File-MakeIndex" in *MakeIndex*. To make an index entry with an `@` in it, you have to escape it with a backslash; Eplain provides no macros for doing this.

After any index command, Eplain runs `\hookaction{afterindexterm}`. Because the index commands always add a whatsit item to the current list, you may wish to preserve a penalty or space past the new item. For example, given a conditional `\if@aftersctnhead` set true when you're at a section heading, you could do:

```
\hookaction{afterindexterm}{\if@aftersctnhead \nobreak \fi}
```

4.11.1.2 Modifying index entries

All the index commands described in the previous section take an initial optional argument before the index term, which modify the index entry's meaning in various ways. You can specify only one of the following in any given command.

These work via MakeIndex's "encapsulation" feature. See Section 4.11.3 [Customizing indexing], page 21, if you're not using the default characters for the MakeIndex operators. The other optional argument (specifying a subterm) is independent of these.

Here are the possibilities:

`begin`
`end` These mark an index entry as the beginning or end of a range. The index entries must match exactly for MakeIndex to recognize them. Example:

```
\sidx[begin]{future}[Cohen, Leonard]
...
\sidx[end]{future}[Cohen, Leonard]
```

will typeset as something like

future,
  Cohen, Leonard, 65–94

`see` This marks an index entry as pointing to another; the real index term is an additional (non-optional) argument to the command. Thus you can anticipate a term readers may wish to look up, yet which you have decided not to index. Example:

```
\sidx[see]{analysis}[archetypal]{archetypal criticism}
```

becomes

analysis,
  archetypal, See archetypal criticism

`seealso` Similar to `see` (the previous item), but also allows for normal index entries of the referencing term. Example:

```
\sidx[seealso]{archetypal criticism}[elements of]{dichotomies}
```

becomes

archetypal criticism,
  elements of, 75, 97, 114, See also dichotomies

(Aside for the academically curious: The archetypally critical book I took these dichotomous examples from is Laurence Berman's *The Musical Image*, which I happened to co-design and typeset.)

`pagemarkup=`*cs*

This puts *cs* before the page number in the typeset index, thus allowing you to underline definitive entries, italicize examples, and the like. You do *not* precede the control sequence *cs* with a backslash. (That just leads to expansive difficulties.) Naturally it is up to you to define the control sequences you want to use. Example:

```
\def\defn#1{{\sl #1}}
\sidx[pagemarkeup=defn]{indexing}
```

becomes something like

```
indexing, \defn{75}
```

4.11.1.3 Proofing index terms

As you are reading through a manuscript, it is helpful to see what terms have been indexed, so you can add others, catch miscellaneous errors, etc. (Speaking from bitter experience, I can say it is extremely error-prone to leave all indexing to the end of the writing, since it involves adding many TEX commands to the source files.)

So Eplain puts index terms in the margin of each page, if you set `\indexproofingtrue`. It is `false` by default. The terms are typeset by the macro `\indexproofterm`, which takes a single argument, the term to be typeset. Eplain's definition of `\indexproofterm` just puts it into an `\hbox`, first doing `\indexprooffont`, which Eplain defines to select the font `cmtt8`. With this definition long terms run off the page, but since this is just for proofreading anyway, it seems acceptable.

On the other hand, we certainly don't want the index term to run into the text of the page, so Eplain uses the right-hand side of the page rather than the left-hand page (assuming a language read left to right here). So `\ifodd\pageno`, Eplain kerns by `\outsidemargin`, otherwise by `\insidemargin`. If those macros are undefined, `\indexsetmargins` defines them to be one inch plus `\hoffset`.

To get the proofing index entries on the proper page, Eplain defines a new insertion class `\@indexproof`. To unbox any index proofing material, Eplain redefines `\makeheadline` to call `\indexproofunbox` before the original `\makeheadline`. Thus, if you have your own output routine, that redefines or doesn't use `\makeheadline`, it's up to you to call `\indexproofunbox` at the appropriate time.

4.11.2 Typesetting an index

The command `\readindexfile{i}` reads and typesets the '`.ind`' file that MakeIndex outputs (from the '`.idx`' file which the indexing commands in the previous sections write). Eplain defines a number of commands that support the default MakeIndex output.

More precisely, `\readindexfile` reads `\indexfilebasename.`*index-letter*`nd`, where the *index-letter* is the argument. `\indexfilebasename` is `\jobname` by default, but if you have different indexes in different parts of a book, you may wish to change it, just as with bibliographies (see Section 4.3 [Citations], page 6).

MakeIndex was designed to work with LaTEX; therefore, by default the '`.ind`' file starts with `\begin{theindex}` and ends with `\end{theindex}`. If no `\begin` has been defined, Eplain defines one to ignore its argument and set up for typesetting the index (see below), and also defines a `\end` to ignore its argument. (In a group, naturally, since there is a primitive `\end`).

Eplain calls `\indexfonts`, sets `\parindent = 0pt`, and does `\doublecolumns` (see Section 4.15 [Multiple columns], page 24) at the `\begin{theindex}`. `\indexfonts` does nothing by default; it's just there for you to override. (Indexes are usually typeset in smaller type than the main text.)

It ends the setup with `\hookrun{beginindex}`, so you can override anything you like in that hook (see Section 6.6.3 [Hooks], page 38). For example:

```
\hookaction{beginindex}{\triplecolumns}
```

MakeIndex turns each main index entry into an `\item`, subentries into `\subitem`, and subsubentries into `\subsubitem`. By default, the first line of main entries are not indented, and subentries are indented 1em per level. Main entries are preceded by a `\vskip` of `\aboveitemskipamount`, `0pt plus2pt` by default. Page breaks are encouraged before main entries (`\penalty -100`), but prohibited afterwards—Eplain has no provision for "continued" index entries.

All levels do the following:

```
\hangindent = 1em
\raggedright
\hyphenpenalty = 10000
```

Each entry ends with `\hookrun{indexitem}`, so you can change any of this. For example, to increase the allowable rag:

```
\hookaction{indexitem}{\advance\rightskip by 2em}
```

Finally, MakeIndex outputs `\indexspace` between each group of entries in the '`.ind`' file. Eplain makes this equivalent to `\bigbreak`.

4.11.3 Customizing indexing

By default, MakeIndex outputs ‘`, `’ after each term in the index. To change this, you can add the following to your MakeIndex style (‘`.ist`’) file:

```
delim_0 "\\afterindexterm "
delim_1 "\\afterindexterm "
delim_2 "\\afterindexterm "
```

Eplain makes `\afterindexterm` equivalent to `\quad`.

You can also change the keywords Eplain recognizes (see Section 4.11.1.2 [Modifying index entries], page 18):

`\idxbeginrangeword`
: ‘begin’

`\idxendrangeword`
: ‘end

`\idxseeword`
: ‘see

`\idxseealsoword`
: ‘seealso’

You can also change the magic characters Eplain puts into the ‘`.idx`’ file, in case you’ve changed them in the `.ist` file:

`\idxsubentryseparator`
: ‘`!`’

`\idxencapoperator`
: ‘`|`’

`\idxbeginrangemark`
: ‘`(`’

`\idxendrangemark`
: ‘`)`’

There is no macro for the `actual` (‘`@`’ by default) character, because it’s impossible to make it expand properly.

Finally, you can change the (imaginary) page number that “see also” entries sort as by redefining `\idxmaxpagenum`. This is 99999 by default, which is one digit too many for old versions of MakeIndex.

4.12 Justification

Eplain defines three commands to conveniently justify multiple lines of text: `\flushright`, `\flushleft`, and `\center`.

They all work in the same way; let’s take `\center` as the example. To start centering lines, you say `\center` inside a group; to stop, you end the group. Between the two commands, each end-of-line in the input file also starts a new line in the output file.

The entire block of text is broken into paragraphs at blank lines, so all the TEX paragraph-shaping parameters apply in the usual way. This is convenient, but it implies something else that isn't so convenient: changes to any linespacing parameters, such as `\baselineskip`, will have *no effect* on the paragraph in which they are changed. TEX does not handle linespacing changes within a paragraph (because it doesn't know where the line breaks are until the end of the paragraph).

The space between paragraphs is by default one blank line's worth. You can adjust this space by assigning to `\blanklineskipamount`; this (vertical) glue is inserted after each blank line.

Here is an example:

```
{\center First line.

   Second line, with a blank line before.
}
```

This produces:

First line.

Second line, with a blank line before.

You may wish to use the justification macros inside of your own macros. Just be sure to put them in a group. For example, here is how a title macro might be defined:

```
\def\title{\begingroup\titlefont\center}
\def\endtitle{\endgroup}
```

4.13 Tables

Eplain provides a single command, `\makecolumns`, to make generating one particular kind of table easier. More ambitious macro packages might be helpful to you for more difficult applications. The files '`ruled.tex`' and '`TXSruled.tex`', available from '`lifshitz.ph.utexas.edu`' in '`texis/tables`', is the only one I know of.

Many tables are homogenous, i.e., all the entries are semantically the same. The arrangement into columns is to save space on the page, not to encode different meanings. In this kind of the table, it is useful to have the column breaks chosen automatically, so that you can add or delete entries without worrying about the column breaks.

`\makecolumns` takes two arguments: the number of entries in the table, and the number of columns to break them into. As you can see from the example below, the first argument is delimited by a slash, and the second by a colon and a space (or end-of-line). The entries for the table then follow, one per line (not including the line with the `\makecolumns` command itself).

`\parindent` defines the space to the left of the table. `\hsize` defines the width of the table. So you can adjust the position of the table on the page by assignments to these parameters, probably inside a group.

You can also control the penalty at a page break before the `\makecolumns` by setting the parameter `\abovecolumnspenalty`. Usually, the table is preceded by some explanatory text. You wouldn't want a page break to occur after the text and before the table, so Eplain sets it to `10000`. But if the table produced by `\makecolumns` is standing on its own, `\abovecolumnspenalty` should be decreased.

If you happen to give `\makecolumns` a smaller number of entries than you really have, some text beyond the (intended) end of the table will be incorporated into the table, probably producing an error message, or at least some strange looking entries. And if you give `\makecolumns` a larger number of entries than you really have, some of the entries will be typeset as straight text, probably also looking somewhat out of place.

Here is an example:

```
% Arrange 6 entries into 2 columns:
\makecolumns 6/2: % This line doesn't have an entry.
one
two
three
four
five
six
Text after the table.
```

This produces 'one', 'two', and 'three' in the first column, and 'four', 'five', and 'six' in the second.

4.14 Margins

TEX's primitives describe the type area in terms of an offset from the upper left corner, and the width and height of the type. Some people prefer to think in terms of the *margins* at the top, bottom, left, and right of the page, and most composition systems other than TEX conceive of the page laid out in this way. Therefore, Eplain provides commands to directly assign and increment the margins.

`\topmargin =` *dimen*
`\bottommargin =` *dimen*
`\leftmargin =` *dimen*
`\rightmargin =` *dimen*

These commands set the specified margin to the *dimen* given. The `=` and the spaces around it are optional. The control sequences here are not TEX registers, despite appearances; therefore, commands like `\showthe\topmargin` will not do what you expect.

`\advancetopmargin by` *dimen*
`\advancebottommargin by` *dimen*
`\advanceleftmargin by` *dimen*
`\advancerightmargin by` *dimen*

These commands change the specified margin by the *dimen* given.

Regardless of whether you use the assignment or the advance commands, Eplain always changes the type area in response, not the other margins. For example, when TEX starts, the left and right margins are both one inch. If you then say `\leftmargin = 2in`, the right margin will remain at one inch, and the width of the lines (i.e., `\hsize`) will decrease by one inch.

When you use any of these commands, Eplain computes the old value of the particular margin, by how much you want to change it, and then resets the values of TEX's primitive parameters to correspond. Unfortunately, Eplain cannot compute the right or bottom margin without help: you must tell it the full width and height of the final output page. It defines two new parameters for this:

`\paperheight`
: The height of the output page; default is 11in.

`\paperwidth`
: The width of the output page; default is 8.5in.

If your output page has different dimensions than this, you must reassign to these parameters, as in

```
\paperheight = 11in
\paperwidth = 17in
```

4.15 Multiple columns

Eplain provides for double, triple, and quadruple column output: say `\doublecolumns`, `\triplecolumns`, or `\quadcolumns`, and from that point on, the manuscript will be set in columns. To go back to one column, say `\singlecolumn`.

You may need to invoke `\singlecolumn` to balance the columns on the last page of output.

To do a "column eject", i.e., move to the top of the next column, do `\columnfill`. This does not actually force an eject, however: it merely inserts a kern of size `\@normalvsize` minus `\pagetotal` (`\@normalvsize` being the usual height of the page; to implement multicolumns, Eplain multiplies `\vsize` itself by the number of columns). In most circumstances, a column break will be forced after this kern (during the column splitting operation when the whole page is output), as desired.

The columns are separated by the value of the dimen parameter `\gutter`. Default value is two picas.

All the `\...columns` macros insert the value of the glue parameter `\abovedoublecolumnskip` before the multicolumn text, and the value of the glue parameter `\belowdoublecolumnskip` after it. The default value for both of these parameters is `\bigskipamount`, i.e., one linespace in plain TEX.

The macros take into account only the insertion classes defined by plain TEX; namely, footnotes and `\topinserts`. If you have additional insertion classes, you will need to change the implementation.

Also, Eplain makes insertions the full page width. There is no provision for column-width insertions.

4.16 Footnotes

The most common reference mark for footnotes is a raised number, incremented on each footnote. The `\numberedfootnote` macro provides this. It takes one argument, the footnote text.

If your document uses only numbered footnotes, you could make typing `\numberedfootnote` more convenient with a command such as:

```
\let\footnote = \numberedfootnote
```

After doing this, you can type your footnotes as `\footnote{`*footnote text*`}`, instead of as `\numberedfootnote{`*footnote text*`}`.

Eplain keeps the current footnote number in the count register `\footnotenumber`. So, to reset the footnote number to zero, as you might want to do at, for example, the beginning of a chapter, you could say `\footnotenumber=0`.

Plain TEX separates the footnote marker from the footnote text by an en space (it uses the `\textindent` macro). In Eplain, you can change this space by setting the dimension register `\footnotemarkseparation`. The default is still an en.

You can produce a space between footenotes by setting the glue register `\interfootnoteskip`. The default is zero.

`\parskip` is also set to zero by default before the beginning of each footnote (but not for the text of the footnote).

You can also control footnote formatting in a more general way: Eplain expands the token register `\everyfootnote` before a footnote is typeset, but after the default values for all the parameters have been established. For example, if you want your footnotes to be printed in seven-point type, indented by one inch, you could say:

```
\everyfootnote = {\sevenrm \leftskip = 1in}
```

By default, an `\hrule` is typeset above each group of footnotes on a page. You can control the dimensions of this rule by setting the dimension registers `\footnoterulewidth` and `\footnoteruleheight`. The space between the rule and the first footnote on the page is determined by the dimension register `\belowfootnoterulespace`. If you don't want any rule at all, set `\footenoteruleheight=0pt`, and, most likely, `\belowfootnoterulespace=0pt`. The defaults for these parameters typeset the rule in the same way as plain TEX: the rule is 0.4 points high, 2 true inches wide, with 2.6 points below it.

The space above the rule and below the text on the page is controlled by the glue register `\skip\footins`. The default is a plain TEX `\bigskip`.

4.17 Fractions

Exercise 11.6 of *The TEXbook* describes a macro `\frac` for setting fractions, but `\frac` never made it into plain TEX. So Eplain includes it.

`\frac` typesets the numerator and denominator in `\scriptfont0`, slightly raised and lowered. The numerator and denominator are separated by a slash. The denominator must be enclosed in braces if it's more than one token long, but the numerator need not be. (This is a consequence of `\frac` taking delimited arguments; see page 203 of *The TEXbook* for an explanation of delimited macro arguments.)

For example, `\frac 23/{64}` turns '23/64' into ${}^{23}\!/_{64}$.

4.18 Paths

When you typeset long pathnames, electronic mail addresses, or other such "computer" names, you would like TEX to break lines at punctuation characters within the name, rather than trying to find hyphenation points within the words. For example, it would be better to break the email address `letters@alpha.gnu.ai.mit.edu` at the '`@`' or a '`.`', rather than at the hyphenation points in '`letters`' and '`alpha`'.

If you use the `\path` macro to typeset the names, TEX will find these good breakpoints. The argument to `\path` is delimited by any other other than '`\`' which does not appear in the name itself. '`|`' is often a good choice, as in:

```
\path|letters@alpha.gnu.ai.mit.edu|
```

You can control the exact set of characters at which breakpoints will be allowed by calling `\discretionaries`. This takes the same sort of delimited argument; any character in the argument will henceforth be a valid breakpoint within `\path`. The default set is essentially all the punctuation characters:

```
\discretionaries |~!@$%^&*()_+`-=#{}[]:";'<>,.?\/|
```

If for some reason you absolutely must use `\` as the delimiter character for `\path`, you can set `\specialpathdelimiterstrue`. (Other delimiter characters can still be used.) TEX then processes the `\path` argument about four times more slowly.

4.19 Logos

Eplain redefines the `\TeX` macro of plain TEX to end with `\null`, so that the proper spacing is produced when `\TeX` is used at the end of a sentence. The other ...TEX macros listed here do this, also.

Eplain defines `\AMSTeX`, `\BibTeX` `\AMSLaTeX`, `\LAMSTeX`, `\LaTeX` `\MF`, and `\SLiTeX` to produce their respective logos. (Sorry, the logos are not shown here.) Some spelling variants of these are also supported.

4.20 Boxes

The solid rectangle that Eplain uses as a marker in unordered lists (see Section 4.6 [Lists], page 10) is available by itself: just say `\blackbox`.

You can create black boxes of arbitrary size with `\hrule` or `\vrule`.

You can also get unfilled rectangles with `\makeblankbox`. This takes two explicit arguments: the height and depth of the rules that define the top and bottom of the rectangle. (The two arguments are added to get the width of the left and right borders, so that the thickness of the border is the same on all four sides.) It also uses, as implicit arguments, the dimensions of `\box0` to define the dimensions of the rectangle it produces. (The contents of `\box0` are ignored.)

Here is an example. This small raised open box is suitable for putting next to numbers in, e.g., a table of contents.

```
\def\openbox{%
  \ht0 = 1.75pt \dp0 = 1.75pt \wd0 = 3.5pt
  \raise 2.75pt \makeblankbox{.2pt}{.2pt}
}
```

Finally, you can put a box around arbitrary text with `\boxit`. This takes one argument, which must itself be a (TEX) box, and puts a printed box around it, separated by `\boxitspace` white space (3 points by default) on all four sides. For example:

```
\boxit{\hbox{This text is boxed.}}
```

The reason that the argument must be a box is that when the text is more than one line long, TEX cannot figure out the line length for itself. Eplain does set `\parindent` to zero inside `\boxit`, since it is very unlikely you would want indentation there. (If you do, you can always reset it yourself.)

`\boxit` uses `\ehrule` and `\evrule` so that you can easily adjust the thicknesses of the box rules. See Section 4.2 [Rules], page 5.

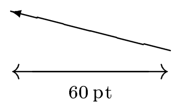
5 Arrow theoretic diagrams

This chapter describes definitions for producing commutative diagrams.

Steven Smith wrote this documentation (and the macros).

5.1 Slanted lines and vectors

The macros `\drawline` and `\drawvector` provide the capability found in LaTeX's picture mode to draw slanted lines and vectors of certain directions. Both of these macros take three arguments: two integer arguments to specify the direction of the line or vector, and one argument to specify its length. For example, '`\drawvector(-4,1){60pt}`' produces the vector



which lies in the 2d quadrant, has a slope of minus 1/4, and a width of 60 pt.

Note that if an `\hbox` is placed around `\drawline` or `\drawvector`, then the width of the `\hbox` will be the positive dimension specified in the third argument, except when a vertical line or vector is specified, e.g., `\drawline(0,1){1in}`, which has zero width. If the specified direction lies in the 1st or 2d quadrant (e.g., `(1,1)` or `(-2,3)`), then the `\hbox` will have positive height and zero depth. Conversely, if the specified direction lies in the 3d or 4th quadrant (e.g., `(-1,-1)` or `(2,-3)`), then the `\hbox` will have positive depth and zero height.

There are a finite number of directions that can be specified. For `\drawline`, the absolute value of each integer defining the direction must be less than or equal to six, i.e., `(7,-1)` is incorrect, but `(6,-1)` is acceptable. For `\drawvector`, the absolute value of each integer must be less than or equal to four. Furthermore, the two integers cannot have common divisors; therefore, if a line with slope 2 is desired, say `(2,1)` instead of `(4,2)`. Also, specify `(1,0)` instead of, say, `(3,0)` for horizontal lines and likewise for vertical lines.

Finally, these macros depend upon the LaTeX font `line10`. If your site doesn't have this font, ask your system administrator to get it. Future enhancements will include macros to draw dotted lines and dotted vectors of various directions.

5.2 Commutative diagrams

The primitive commands `\drawline` and `\drawvector` can be used to typeset arrow theoretic diagrams. This section describes (1) macros to facilitate typesetting arrows and morphisms, and (2) macros to facilitate the construction of commutative diagrams. All macros described in this section must be used in math mode.

5.2.1 Arrows and morphisms

The macros `\mapright` and `\mapleft` produce right and left pointing arrows, respectively. Use superscript (`^`) to place a morphism above the arrow, e.g., '`\mapright^\alpha`'; use subscript (`_`) to place a morphism below the arrow, e.g., '`\mapright_{\tilde l}`'. Superscripts and subscripts may be used simulataneously, e.g., '`\mapright^\pi_{\rm epimor.}`'.

Similarly, the macros `\mapup` and `\mapdown` produce up and down pointing arrows, respectively. Use `\rt` to place a morphism to the right of the arrow, e.g., '`\mapup\rt{\rm id}`'; use `\lft` to place a morphism to the left of the arrow, e.g., '`\mapup\lft\omega`'. `\lft` and `\rt` may be used simultaneously, e.g., '`\mapdown\lft\pi\rt{\rm monomor.}`'.

Slanted arrows are produced by the macro `\arrow`, which takes a direction argument (e.g., '`\arrow(3,-4)`'). Use `\rt` and `\lft` to place morphisms to the right and left, respectively, of the arrow. A slanted line (no arrowhead) is produced with the macro `\sline`, whose syntax is identical to that of `\arrow`.

The length of these macros is predefined by the default TEX dimensions `\harrowlength`, for horizontal arrows (or lines), `\varrowlength`, for vertical arrows (or lines), and `\sarrowlength`, for slanted arrows (or lines). To change any of these dimensions, say, e.g., '`\harrowlength=40pt`'. As with all other TEX dimensions, the change may be as global or as local as you like. Furthermore, the placement of morphisms on the arrows is controlled by the dimensions `\hmorphposn`, `\vmorphposn`, and `\morphdist`. The first two dimensions control the horizontal and vertical position of the morphism from its default position; the latter dimension controls the distance of the morphism from the arrow. If you have more than one morphism per arrow (i.e., a `^/_` or `\lft/\rt` construction), use the parameters `\hmorphposnup`, `\hmorphposndn`, `\vmorphposnup`, `\vmorphposndn`, `\hmorphposnrt`, `\hmorphposnlft`, `\vmorphposnrt`, and `\vmorphposnlft`. The default values of all these dimensions are provided in the section on parameters that follows below.

There is a family of macros to produce horizontal lines, arrows, and adjoint arrows. The following macros produce horizontal maps and have the same syntax as `\mapright`:

`\mapright`
: `$X\mapright Y$` $\equiv X \longrightarrow Y.$

`\mapleft`
: `$X\mapleft Y$` $\equiv X \longleftarrow Y.$

`\hline`
: `$X\hline Y$` $\equiv X \relbar\joinrel\relbar Y.$

`\bimapright`
: `$X\bimapright Y$` $\equiv X \rightrightarrows Y.$

`\bimapleft`
: `$X\bimapleft Y$` $\equiv X \leftleftarrows Y.$

`\adjmapright`
: `$X\adjmapright Y$` $\equiv X \leftrightarrows Y.$

`\adjmapleft`
: `$X\adjmapleft Y$` $\equiv X \rightleftarrows Y.$

`\bihline`
: `$X\bihline Y$` $\equiv X = Y.$

There is also a family of macros to produce vertical lines, arrows, and adjoint arrows. The following macros produce vertical maps and have the same syntax as `\mapdown`:

`\mapdown`
: (a down arrow)

`\mapup`
: (an up arrow)

`\vline`
: (vertical line)

`\bimapdown`
: (two down arrows)

`\bimapup` (two up arrows)

`\adjmapdown`
(two adjoint arrows; down then up)

`\adjmapup`
(two adjoint arrows; up then down)

`\bivline` (two vertical lines)

Finally, there is a family of macros to produce slanted lines, arrows, and adjoint arrows. The following macros produce slanted maps and have the same syntax as `\arrow`:

`\arrow` (a slanted arrow)

`\sline` (a slanted line)

`\biarrow` (two straight arrows)

`\adjarrow`
(two adjoint arrows)

`\bisline` (two straight lines)

The width between double arrows is controlled by the parameter `\channelwidth`. The parameters `\hchannel` and `\vchannel`, if nonzero, override `\channelwidth` by controlling the horizontal and vertical shifting from the first arrow to the second.

There are no adornments on these arrows to distinguish inclusions from epimorphisms from monomorphisms. Many texts, such as Lang's book *Algebra*, use as a tasteful alternative the symbol 'inc' (in roman) next to an arrow to denote inclusion.

Future enhancements will include a mechanism to draw curved arrows found in, e.g., the Snake Lemma, by employing a version of the `\path` macros of Appendix D of *The TEXbook*.

5.2.2 Construction of commutative diagrams

There are two approaches to the construction of commutative diagrams described here. The first approach, and the simplest, treats commutative diagrams like fancy matrices, as Knuth does in Exercise 18.46 of *The TEXbook*. This case is covered by the macro `\commdiag`, which is an altered version of the Plain TEX macro `\matrix`. An example suffices to demonstrate this macro. The following commutative diagram (illustrating the covering homotopy property; Bott and Tu, *Differential Forms in Algebraic Topology*)

$$\begin{tikzcd} Y \arrow[r, "f"] \arrow[d] & E \arrow[d] \\ Y \times I \arrow[ur, "f_t"] \arrow[r, "\bar f_t"] & X \end{tikzcd}$$

is produced with the code

```
$$\commdiag{Y&\mapright^f&E\cr \mapdown&\arrow(3,2)\lft{f_t}&\mapdown\cr
Y\times I&\mapright^{\bar f_t}&X}$$
```

Of course, the parameters may be changed to produce a different effect. The following commutative diagram (illustrating the universal mapping property; Warner, *Foundations of Differentiable Manifolds and Lie Groups*)

$$\begin{tikzcd}
V \otimes W \arrow[drr, "\tilde l"] & & \\
V \times W \arrow[u, "\phi"] \arrow[rr, "l"] & & U
\end{tikzcd}$$

is produced with the code

```
$$\varrowlength=20pt
\commdiag{V\otimes W\cr \mapup\lft\phi&\arrow(3,-1)\rt{\tilde l}\cr
V\times W&\mapright^l&U\cr}$$
```

A diagram containing isosceles triangles is achieved by placing the apex of the triangle in the center column, as shown in the example (illustrating all constant minimal realizations of a linear system; Brockett, *Finite Dimensional Linear Systems*)

$$\begin{tikzcd}
& R^m \arrow[dl, "\mathbf{B}"'] \arrow[dr, "\mathbf{G}"] & \\
R^n \arrow[rr, "\mathbf{P}"] \arrow[dd, "e^{\mathbf{A}t}"'] & & R^n \arrow[dd, "e^{\mathbf{F}t}"] \\
& & \\
R^n \arrow[rr, "\mathbf{P}"] \arrow[dr, "\mathbf{C}"'] & & R^n \arrow[dl, "\mathbf{H}"] \\
& R^q &
\end{tikzcd}$$

which is produced with the code

```
$$\sarrowlength=.42\harrowlength
\commdiag{&R^m\cr &\arrow(-1,-1)\lft{\bf B}\quad \arrow(1,-1)\rt{\bf G}\cr
R^n&\mapright^{\bf P}&R^n\cr
\mapdown\lft{e^{{\bf A}t}}&&\mapdown\rt{e^{{\bf F}t}}\cr
R^n&\mapright^{\bf P}&R^n\cr
&\arrow(1,-1)\lft{\bf C}\quad \arrow(-1,-1)\rt{\bf H}\cr
&R^q\cr}$$
```

Other commutative diagram examples appear in the file `commdiags.tex`, which is distributed with this package.

In these examples the arrow lengths and line slopes were carefully chosen to blend with each other. In the first example, the default settings for the arrow lengths are used, but a direction for the arrow must be chosen. The ratio of the default horizontal and vertical arrow lengths is approximately the golden mean $\gamma = 1.618\ldots$; the arrow direction closest to this mean is `(3,2)`. In the second example, a slope of $-1/3$ is desired and the default horizontal arrow length is 60 pt; therefore, choose a vertical arrow length of 20 pt. You may affect the interline glue settings of `\commdiag` by redefining the macro `\commdiagbaselines`. (cf. Exercise 18.46 of *The TeXbook* and the section on parameters below.)

The width, height, and depth of all morphisms are hidden so that the morphisms' size do not affect arrow positions. This can cause a large morphism at the top or bottom of a diagram to impinge upon the text surrounding the diagram. To overcome this problem, use

TeX's `\noalign` primitive to insert a `\vskip` immediately above or below the offending line, e.g., '`$$\commdiag{\noalign{\vskip6pt}X&\mapright^\int&Y\cr ...}`'.

The macro `\commdiag` is too simple to be used for more complicated diagrams, which may have intersecting or overlapping arrows. A second approach, borrowed from Francis Borceux's *Diagram* macros for LaTeX, treats the commutative diagram like a grid of identically shaped boxes. To compose the commutative diagram, first draw an equally spaced grid, e.g.,

$$\begin{matrix} . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \end{matrix}$$

on a piece of scratch paper. Then draw each element (vertices and arrows) of the commutative diagram on this grid, centered at each grid point. Finally, use the macro `\gridcommdiag` to implement your design as a TeX alignment. For example, the cubic diagram

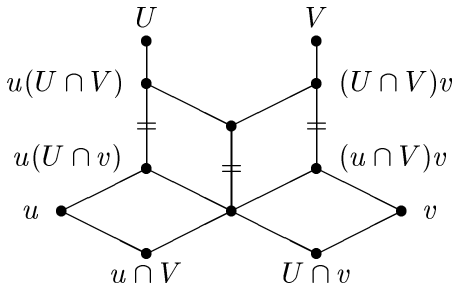
$$\begin{tikzcd}
& B \arrow[rr, "b"] \arrow[dd, "f"'] & & D \arrow[dd, "h"] \\
A \arrow[ur, "a"] \arrow[rr, "c"] \arrow[dd, "e"'] & & C \arrow[ur, "d"'] \arrow[dd, "g"] & \\
& F \arrow[rr, "j"'] & & H \\
E \arrow[ur, "i"] \arrow[rr, "k"'] & & G \arrow[ur, "l"'] &
\end{tikzcd}$$

that appears in Francis Borceux's documentation can be implemented on a 7 by 7 grid, and is achieved with the code

```
$$\harrowlength=48pt \varrowlength=48pt \sarrowlength=20pt
\def\cross#1#2{\setbox0=\hbox{$#1$}%
  \hbox to\wd0{\hss\hbox{$#2$}\hss}\llap{\unhbox0}}
\gridcommdiag{&&B&&\mapright^b&&D\cr
&\arrow(1,1)\lft a&&&&\arrow(1,1)\lft d\cr
A&&\cross{\hmorphposn=12pt\mapright^c}{\vmorphposn=-12pt\mapdown\lft f}
&&C&&\mapdown\rt h\cr\cr
\mapdown\lft e&&F&&\cross{\hmorphposn=-12pt\mapright_j}
{\vmorphposn=12pt\mapdown\rt g}&&H\cr
&\arrow(1,1)\lft i&&&&\arrow(1,1)\rt l\cr
E&&\mapright_k&&G\cr}$$
```

The dimensions `\hgrid` and `\vgrid` control the horizontal and vertical spacing of the grid used by `\gridcommdiag`. The default setting for both of these dimensions is 15 pt. Note that in the example of the cube the arrow lengths must be adjusted so that the arrows overlap into neighboring boxes by the desired amount. Hence, the `\gridcommdiag` method, albeit more powerful, is less automatic than the simpler `\commdiag` method. Furthermore, the ad hoc macro `\cross` is introduced to allow the effect of overlapping arrows. Finally, note that the positions of four of the morphisms are adjusted by setting `\hmorphposn` and `\vmorphposn`.

One is not restricted to a square grid. For example, the proof of Zassenhaus's Butterfly Lemma can be illustrated by the diagram (appearing in Lang's book *Algebra*)



This diagram may be implemented on a 9 by 12 grid with an aspect ratio of 1/2, and is set with the code

```
$$\hgrid=16pt \vgrid=8pt \sarrowlength=32pt
\def\cross#1#2{\setbox0=\hbox{$#1$}%
  \hbox to\wd0{\hss\hbox{$#2$}\hss}\llap{\unhbox0}}
\def\l#1{\llap{$#1$\hskip.5em}}
\def\r#1{\rlap{\hskip.5em$#1$}}
\gridcommdiag{&&U&&&&V\cr &&\bullet&&&&\bullet\cr
&&\sarrowlength=16pt\sline(0,1)&&&&\sarrowlength=16pt\sline(0,1)\cr
&&\l{u(U\cap V)}\bullet&&&&\bullet\r{(U\cap V)v}\cr
&&&\sline(2,-1)&&\sline(2,1)\cr
&&\cross{=}{\sline(0,1)}&&\bullet&&\cross{=}{\sline(0,1)}\cr\cr
&&\l{^{\textstyle u(U\cap v)}}\bullet&&\cross{=}{\sline(0,1)}&&
 \bullet\r{^{\textstyle(u\cap V)v}}\cr
&\sline(2,1)&&\sline(2,-1)&&\sline(2,1)&&\sline(2,-1)\cr
\l{u}\bullet&&&&\bullet&&&&\bullet\r{v}\cr
&\sline(2,-1)&&\sline(2,1)&&\sline(2,-1)&&\sline(2,1)\cr
&&\bullet&&&&\bullet\cr &&u\cap V&&&&U\cap v\cr}$$
```

Again, the construction of this diagram requires careful choices for the arrow lengths and is facilitated by the introduction of the ad hoc macros `\cross`, `\r`, and `\l`. Note also that superscripts were used to adjust the position of the vertices $u(U \cap v)$ and $(u \cap V)v$. Many diagrams may be typeset with the predefined macros that appear here; however, ingenuity is often required to handle special cases.

5.2.3 Commutative diagram parameters

The following is a list describing the parameters used in the commutative diagram macros. These dimensions may be changed globally or locally.

`\harrowlength`
: (Default: 60 pt) The length of right or left arrows.

`\varrowlength`
: (Default: 0.618`\harrowlength`) The length of up or down arrows.

`\sarrowlength`
: (Default: 60 pt) The horizontal length of slanted arrows.

`\hmorphposn`
: (Default: 0 pt) The horizontal position of the morphism with respect to its default position. There are also the dimensions `\hmorphposnup`, `\hmorphposndn`, `\hmorphposnrt`, and `\hmorphposnlft` for `^/_` or `\lft/\rt` constructions.

`\vmorphposn`
: (Default: 0 pt) The vertical position of the morphism with respect to its default position. There are also the dimensions `\vmorphposnup`, `\vmorphposndn`, `\vmorphposnrt`, and `\vmorphposnlft` for `^/_` or `\lft/\rt` constructions.

`\morphdist`
: (Default: 4 pt) The distance of morphisms from slanted lines or arrows.

`\channelwidth`
: (Default: 3 pt) The distance between double lines or arrows.

`\hchannel, \vchannel`
: (Defaults: 0 pt) Overrides `\channelwidth`. The horizontal and vertical shifts between double lines or arrows.

`\commdiagbaselines`
: (Default: `\baselineskip=15pt \lineskip=3pt \lineskiplimit=3pt`) The parameters used by `\commdiag` for setting interline glue.

`\hgrid`
: (Default: 15 pt) The horizontal spacing of the grid used by `\gridcommdiag`.

`\vgrid`
: (Default: 15 pt) The vertical spacing of the grid used by `\gridcommdiag`.

6 Programming definitions

The definitions in this section are only likely to be useful when you are writing nontrivial macros, not when writing a document.

6.1 Category codes

Plain TEX defines `\active` (as the number 13) for use in changing category codes. Although the author of *The TEXbook* has "intentionally kept the category codes numeric", two other categories are commonly used: letters (category code 11) and others (12). Therefore, Eplain defines `\letter` and `\other`.

Sometimes it is cleaner to make a character active without actually writing a `\catcode` command. The `\makeactive` command takes a character as an argument to make active (and ignores following spaces). For example, here are two commands which both make `\` active:

```
\makeactive`\\   \makeactive92
```

Usually, when you give a definition to an active character, you have to do so inside a group where you temporarily make the character active, and then give it a global definition (cf. the definition of `\obeyspaces` in *The TEXbook*). This is inconvenient if you are writing a long macro, or if the character already has a global definition you do not wish to transcend. Eplain provides `\letreturn`, which defines the usual end-of-line character to be the argument. For example:

```
\def\mymacro{... \letreturn\myreturn ... }
\mymacro hello
there
```

The end-of-line between '`hello`' and '`there`' causes `\myreturn` to be expanded.

The TEXbook describes `\uncatcodespecials`, which makes all characters which are normally "special" into "other" characters, but the definition never made it into plain TEX. Eplain therefore defines it.

Finally, `\percentchar` expands into a literal '%' character. This is useful when you `\write` TEX output to a file, and want to avoid spurious spaces. For example, Eplain writes a `\percentchar` after the definition of cross-references. The macros `\lbracechar` and `\rbracechar` expand similarly.

6.2 Allocation macros

Plain TEX provides macros that allocate registers of each primitive type in TEX, to prevent different sets of macros from using the same register for two different things. The macros are all named starting with '`new`', e.g., `\newcount` allocates a new "count" (integer) register. Such allocations are usually needed only at the top level of some macro definition file; therefore, plain TEX makes the allocation registers `\outer`, to help find errors. (The error this helps to find is a missing right brace in some macro definition.)

Sometimes, however, it is useful to allocate a register as part of some macro. An outer control sequence cannot be used as part of a macro definition (or in a few other contexts: the parameter text of a definition, an argument to a definition, the preamble of an alignment, or in conditional text that is being skipped). Therefore, Eplain defines "inner" versions of all the allocation macros, named with the prefix '`inner`': `\innernewbox`, `\innernewcount`, `\innernewdimen`, `\innernewfam`,

\innernewhelp, \innernewif, \innernewinsert, \innernewlanguage, \innernewread, \innernewskip, \innernewtoks, \innernewwrite.

You can also define non-outer versions of other macros in the same way that Eplain defines the above. The basic macro is called \innerdef:

```
\innerdef \innername {outername}
```

The first argument (*innername*) to \innerdef is the control sequence that you want to define. Any previous definition of *innername* is replaced. The second argument (*outername*) is the *characters* in the name of the outer control sequence. (You can't use the actual control sequence name, since it's outer!)

If the outer control sequence is named *cs*, and you want to define inner*cs* as the inner one, you can use \innerinnerdef, which is just an abbreviation for a call to \innerdef. For example, these two calls are equivalent:

```
\innerdef\innerproclaim{proclaim}
\innerinnerdef{proclaim}
```

6.3 Iteration

You can iterate through a comma-separated list of items with \for. Here is an example:

```
\for\name:=karl,kathy\do{%
   \message{\name}%
}%
```

This writes 'karl' and 'kathy' to the terminal. Spaces before or after the commas in the list, or after the :=, are *not* ignored.

\for expands the iterated values fully (with \edef), so this is equivalent to the above:

```
\def\namelist{karl,kathy}%
\for\name:=\namelist\do ...
```

6.4 Macro arguments

It is occasionally useful to redefine a macro that takes arguments to do nothing. Eplain defines \gobble, \gobbletwo, and \gobblethree to swallow one, two, and three arguments, respectively.

For example, if you want to produce a "short" table of contents—one that includes only chapters, say—the easiest thing to do is read the entire .toc file (see Section 4.8 [Contents], page 12), and just ignore the commands that produce section or subsection entries. To be specific:

```
\let\tocchapterentry = \shorttocchapter
\let\tocsectionentry = \gobbletwo
\let\tocsubsectionentry = \gobbletwo
\readtocfile
```

(Of course, this assumes you only have chapters, sections, and subsections in your document.)

In addition, Eplain defines \eattoken to swallow the single following token, using \let. Thus, \gobble followed by '{...}' ignores the entire brace-enclosed text. \eattoken followed by the same ignores only the opening left brace.

Eplain defines a macro \identity which takes one argument and expands to that argument. This may be useful if you want to provide a function for the user to redefine, but don't need to do anything by default. (For example, the default definition of `\eqconstruct` (see Section 4.10.1.1 [Formatting equation references], page 15) is `\identity`.)

You may also want to read an optional argument. The established convention is that optional arguments are put in square brackets, so that is the syntax Eplain recognizes. Eplain ignores space tokens before an optional argument, via `\futurenonspacelet`.

You test for an optional argument by using `\@getoptionalarg`. It takes one argument, a control sequence to expand after reading the argument, if present. If an optional argument is present, the control sequence `\@optionalarg` expands to it; otherwise, `\@optionalarg` is `\empty`. You must therefore have the category code of `@` set to 11 (letter). Here is an example:

```
\catcode`@=\letter
\def\cmd{\@getoptionalarg\finishcmd}
\def\finishcmd{%
  \ifx\@optionalarg\empty
    % No optional argument present.
  \else
    % One was present.
  \fi
}
```

If an optional argument contains another optional argument, the inner one will need to be enclosed in braces, so TEX does not mistake the end of the first for the end of the second.

6.5 Converting to characters

Eplain defines `\xrlabel` to produce control sequence names for cross-reference labels, et al. This macro expands to its argument with an '`_`' appended. (It does this because the usual use of `\xrlabel` is to generate a control sequence name, and we naturally want to avoid conflicts between control sequence names.)

Because `\xrlabel` is fully expandable, to make a control sequence name out of the result you need only do

```
\csname \xrlabel{label}\endcsname
```

The `\csname` primitive makes a control sequence name out of any sequence of character tokens, regardless of category code. Labels can therefore include any characters except for '`\`', '`{`', '`}`', and '`#`', all of which are used in macro definitions themselves.

`\sanitize` takes a control sequence as an argument and converts the expansion of the control sequence into a list of character tokens. This is the behavior you want when writing information like chapter titles to an output file. For example, here is part of the definition of `\writenumberedtocentry`; `#2` is the title that the user has given.

```
...
\def\temp{#2}%
...
  \write\tocfile{%
    ...
    \sanitize\temp
    ...
  }%
```

6.6 Expansion

This section describes some miscellanous macros for expansion, etc.

6.6.1 `\csn` and `\ece`

`\csn{`*name*`}` simply abbreviates `\csname` *name* `\encsname`, thus saving some typing. The extra level of expansion does take some time, though, so I don't recommend it for an inner loop.

`\ece{`*token*`}{`*name*`}` abbreviates

```
\expandafter token \csname name \endcsname
```

For example,

```
\def\fontabbrevdef#1#2{\ece\def{@#1font}{#2}}
\fontabbrevdef{normal}{ptmr}
```

defines a control sequence `\@normalfont` to expand to `ptmr`.

6.6.2 `\edefappend`

`\edefappend` is a way of adding on to an existing definition. It takes two arguments: the first is the control sequence name, the second the new tokens to append to the definition. The second argument is fully expanded (in the `\edef` that redefines the control sequence).

For example:

```
\def\foo{abc}
\def\bar{xyz}
\edefappend\foo{\bar karl}
```

results in `\foo` being defined as '`abcxyzkarl`'.

6.6.3 Hooks

A *hook* is simply a name for a group of actions which is executed in certain places—presumably when it is most useful to allow customization or modification. TeX already provides many builtin hooks; for example, the `\every` … token lists are all examples of hooks.

Eplain provides several macros for adding actions to hooks. They all take two arguments: the name of the hook and the new actions.

`hookaction` *name actions*
`hookappend` *name actions*
`hookprepend` *name actions*

Each of these adds *actions* to the hook *name*. (Any previously-defined actions are retained.) *name* is not a control sequence, but rather the characters of the name.

`hookactiononce` *name* `\`*cs*

`\hookactiononce` adds *cs* to *name*, like the macros above, but first it adds

`\global\let` *“cs* `\relax`

to the definition of *cs*. (This implies *cs* must be a true expandable macro, not a control sequence `\let` to a primitive or some other such thing.) Thus, *cs* is expanded the next time the hook *name* is run, but it will disappear after that.

The `\global` is useful because `\hookactiononce` is most useful when the grouping structure of the TEX code could be anything. Neither this nor the other hook macros do global assignments to the hook variable itself, so TEX's usual grouping rules apply.

The companion macro to defining hook actions is `\hookrun`, for running them. This takes a single argument, the name of the hook. If no actions for the hook are defined, no error ensues.

Here is a skeleton of general `\begin` and `\end` macros that run hooks, and a couple of calls to define actions. The use of `\hookprepend` for the begin action and `\hookappend` for the end action ensures that the actions are executed in proper sequence with other actions (as long as the other actions use `\hookprepend` and `\hookappend` also).

```
\def\begin#1{ ... \hookrun{begin} ... }
\def\end#1{ ... \hookrun{end} ... }
\hookprepend{begin}\start_underline
\hookappend{end}\finish_underline
```

6.6.4 Properties

A *property* is a name/value pair associated with another symbol, traditionally called an *atom*. Both atom and property names are control sequence names.

Eplain provides two macros for dealing with property lists: `\setproperty` and `\getproperty`.

`\setproperty` *atom propname value*

`\setproperty` defines the property *property* on the atom *atom* to be *value*. *atom* and *propname* can be anything acceptable to `\csname`. *value* can be anything.

`\getproperty` *atom propname*

`\getproperty` expands to the value stored for *propname* on *atom*. If *propname* is undefined, it expands to nothing (i.e., `\empty`).

The idea of properties originated in Lisp (I believe). There, the implementation truly does associate properties with atoms. In TEX, where we have no builtin support for properties, the association is only conceptual.

The following example typesets ‘`xyz`’.

```
\setproperty{a}{pr}{xyz}
\getproperty{a}{pr}
```

6.6.5 `\expandonce`

`\expandonce` is defined as `\expandafter\noexpand`. Thus, `\expandonce` *token* expands *token* once, instead of to TEX primitives. This is most useful in an `\edef`.

For example, the following defines `\temp` to be `\foo`, not ‘`abc`’.

```
\def\foo{abc}
\def\bar{\foo}
\edef\temp{\expandonce\bar}
```

6.6.6 `\ifundefined`

`\ifundefined{`*cs*`}` *t* `\else` *f* `\fi` expands the *t* text if the control sequence `\`*cs* is undefined or has been `\let` to `\relax`, and the *f* text otherwise.

Since `\ifundefined` is not a primitive conditional, it cannot be used in places where TeX might skip tokens "at high speed", e.g., within another conditional—TeX can't match up the `\if`'s and `\fi`'s.

This macro was taken directly from *The TeXbook*, page 308.

6.6.7 `\futurenonspacelet`

The `\futurelet` primitive allows you to look at the next token from the input. Sometimes, though, you want to look ahead ignoring any spaces. This is what `\futurenonspacelet` does. It is otherwise the same as `\futurelet`: you give it two control sequences as arguments, and it assigns the next nonspace token to the first, and then expands the second. For example:

```
\futurenonspacelet\temp\finishup
\def\finishup{\ifx\temp ...}
```

6.7 Obeying spaces

`\obeywhitespace` makes both end-of-lines and space characters in the input be respected in the output. Unlike plain TeX's `\obeyspaces`, even spaces at the beginnings of lines turn into blank space.

By default, the size of the space that is produced by a space character is the natural space of the current font, i.e., what `\ ` produces.

Ordinarily, a blank line in the input produces as much blank vertical space as a line of text would occupy. You can adjust this by assigning to the parameter `\blanklineskipamount`: if you set this negative, the space produced by a blank line will be smaller; if positive, larger.

Tabs are not affected by this routine. In particular, if tabs occur at the beginning of a line, they will disappear. (If you are trying to make TeX do the "right thing" with tabs, don't. Use a utility program like *expand* instead.)

6.8 Writing out numbers

`\numbername` produces the written-out form of its argument, i.e., 'zero' through 'ten' for the numbers 0–10, and numerals for all others.

6.9 Mode-specific penalties

TeX's built-in `\penalty` command simply appends to the current list, no matter what kind of list it is. You might intend a particular penalty to always be a "vertical" penalty, however, i.e., appended to a vertical list. Therefore, Eplain provides `\vpenalty` and `\hpenalty` which first leave the other mode and then do `\penalty`.

More precisely, `\vpenalty` inserts `\par` if the current mode is horizontal, and `\hpenalty` inserts `\leavevmode` if the current mode is vertical. (Thus, `\vpenalty` cannot be used in math mode.)

6.10 Auxiliary files

It is common to write some information out to a file to be used on a subsequent run. But when it is time to read the file again, you only want to do so if the file actually exists. `\testfileexistence` is given an argument which is appended to `\jobname`, and sets the conditional `\iffileexists` appropriately.

For example:

```
\testfileexistence{toc}%
\iffileexists
   \input \jobname.toc
\fi
```

Appendix A GNU GENERAL PUBLIC LICENSE

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW.

EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Appendix B Regain your programming freedom

Until a few years ago, programmers in the United States could write any program they wished. This freedom has now been taken away by two developments: software patents, which grant the patent holder an absolute monopoly on some programming technique, and user interface copyright, which forbid compatible implementations of an existing user interface.

In Europe, especially through the GATT treaty, things are rapidly approaching the same pass.

B.1 Software patents

The U.S. Patent and Trademark Office has granted numerous software patents on software techniques. Patents are an absolute monopoly—independent reinvention is precluded. This monopoly lasts for seventeen years, i.e., forever (with respect to computer science).

One patent relevant to TeX is patent 4,956,809, issued to the Mark Williams company on September 11, 1990, applied for in 1982, which covers (among other things)

> representing in a standardized order consisting of a standard binary structure file stored on auxiliary memory or transported on a communications means, said standardized order being different from a different order used on at least one of the different computers;
>
> Converting in each of the different computers binary data read from an auxiliary data storage or communications means from the standardized order to the natural order of the respective host computer after said binary data are read from said auxiliary data storage or communications means and before said binary data are used by the respective host computer; and
>
> Converting in each of the different computers binary data written into auxiliary data storage or communications means from the natural order of the respective host computer to the standardized order prior to said writing.

... in other words, storing data on disk in a machine-independent order, as the DVI, TFM, GF, and PK file formats specify. Even though TeX is “prior art” in this respect, the patent was granted (the patent examiners not being computer scientists, even less computer typographers). Since there is a strong presumption in the courts of a patent’s validity once it has been granted, there is a good chance that users or implementors of TeX could be successfully sued on the issue.

As another example, the X window system, which was intended to be able to be used freely by everyone, is now being threatened by two patents: 4,197,590 on the use of exclusive-or to redraw cursors, held by Cadtrak, a litigation company (this has been upheld twice in court); and 4,555,775, held by AT&T, on the use of backing store to redraw windows quickly.

Here is one excerpt from a recent mailing by the League for Programming Freedom (see Section B.3 [What to do?], page 50) which I feel sums up the situation rather well. It comes from an article in *Think* magazine, issue #5, 1990. The comments after the quote were written by Richard Stallman.

> “You get value from patents in two ways,” says Roger Smith, IBM Assistant General Counsel, intellectual property law. “Through fees, and through licensing negotiations that give IBM access to other patents.
>
> “The IBM patent portfolio gains us the freedom to do what we need to do through cross-licensing—it gives us access to the inventions of others that are the key to rapid innovation. Access is far more valuable to IBM than the fees it receives from its 9,000

> active patents. There's no direct calculation of this value, but it's many times larger than the fee income, perhaps an order of magnitude larger."

This information should dispel the belief that the patent system will "protect" a small software developer from competition from IBM. IBM can always find patents in its collection which the small developer is infringing, and thus obtain a cross-license.

However, the patent system does cause trouble for the smaller companies which, like IBM, need access to patented techniques in order to do useful work in software. Unlike IBM, the smaller companies do not have 9,000 patents and cannot usually get a cross-license. No matter how hard they try, they cannot have enough patents to do this.

Only the elimination of patents from the software field can enable most software developers to continue with their work.

The value IBM gets from cross-licensing is a measure of the amount of harm that the patent system would do to IBM if IBM could not avoid it. IBM's estimate is that the trouble could easily be ten times the good one can expect from one's own patents—even for a company with 9,000 of them.

B.2 User interface copyright

(This section is copied from the GCC manual, by Richard Stallman.)

> *This section is a political message from the League for Programming Freedom to the users of the GNU font utilities. It is included here as an expression of support for the League on my part.*

Apple, Lotus and Xerox are trying to create a new form of legal monopoly: a copyright on a class of user interfaces. These monopolies would cause serious problems for users and developers of computer software and systems.

Until a few years ago, the law seemed clear: no one could restrict others from using a user interface; programmers were free to implement any interface they chose. Imitating interfaces, sometimes with changes, was standard practice in the computer field. The interfaces we know evolved gradually in this way; for example, the Macintosh user interface drew ideas from the Xerox interface, which in turn drew on work done at Stanford and SRI. 1-2-3 imitated VisiCalc, and dBase imitated a database program from JPL.

Most computer companies, and nearly all computer users, were happy with this state of affairs. The companies that are suing say it does not offer "enough incentive" to develop their products, but they must have considered it "enough" when they made their decision to do so. It seems they are not satisfied with the opportunity to continue to compete in the marketplace—not even with a head start.

If Xerox, Lotus, and Apple are permitted to make law through the courts, the precedent will hobble the software industry:

- Gratuitous incompatibilities will burden users. Imagine if each car manufacturer had to arrange the pedals in a different order.
- Software will become and remain more expensive. Users will be "locked in" to proprietary interfaces, for which there is no real competition.
- Large companies have an unfair advantage wherever lawsuits become commonplace. Since they can easily afford to sue, they can intimidate small companies with threats even when they don't really have a case.

- User interface improvements will come slower, since incremental evolution through creative imitation will no longer be permitted.
- Even Apple, etc., will find it harder to make improvements if they can no longer adapt the good ideas that others introduce, for fear of weakening their own legal positions. Some users suggest that this stagnation may already have started.
- If you use GNU software, you might find it of some concern that user interface copyright will make it hard for the Free Software Foundation to develop programs compatible with the interfaces that you already know.

B.3 What to do?

(This section is copied from the GCC manual, by Richard Stallman.)

To protect our freedom from lawsuits like these, a group of programmers and users have formed a new grass-roots political organization, the League for Programming Freedom.

The purpose of the League is to oppose new monopolistic practices such as user-interface copyright and software patents; it calls for a return to the legal policies of the recent past, in which these practices were not allowed. The League is not concerned with free software as an issue, and not affiliated with the Free Software Foundation.

The League's membership rolls include John McCarthy, inventor of Lisp, Marvin Minsky, founder of the Artificial Intelligence lab, Guy L. Steele, Jr., author of well-known books on Lisp and C, as well as Richard Stallman, the developer of GNU CC. Please join and add your name to the list. Membership dues in the League are $42 per year for programmers, managers and professionals; $10.50 for students; $21 for others.

The League needs both activist members and members who only pay their dues.

To join, or for more information, phone (617) 492-0023 or write to:

League for Programming Freedom
1 Kendall Square #143
P.O. Box 9171
Cambridge, MA 02139

You can also send electronic mail to `league@prep.ai.mit.edu`.

Here are some suggestions from the League for things you can do to protect your freedom to write programs:

- Don't buy from Xerox, Lotus or Apple. Buy from their competitors or from the defendants they are suing.
- Don't develop software to work with the systems made by these companies.
- Port your existing software to competing systems, so that you encourage users to switch.
- Write letters to company presidents to let them know their conduct is unacceptable.
- Tell your friends and colleagues about this issue and how it threatens to ruin the computer industry.
- Above all, don't work for the look-and-feel plaintiffs, and don't accept contracts from them.
- Write to Congress to explain the importance of this issue.

House Subcommittee on Intellectual Property
2137 Rayburn Bldg
Washington, DC 20515

Senate Subcommittee on Patents, Trademarks and Copyrights
United States Senate
Washington, DC 20510

(These committees have received lots of mail already; let's give them even more.)

Express your opinion! You can make a difference.

Macro index

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Concept index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

Table of Contents