Integrating Noninteractive Document Processors into an Interactive Environment[*]

Pehong Chen[†]
Michael A. Harrison

Computer Science Division
University of California
Berkeley, CA 94720

April 27, 1987

Abstract

Conventional document preparation involving noninteractive processors is usually carried out in an unintegrated fashion. In this paper we describe an integrated environment based on a host of TEX-related processors, using the interactive text editor EMACS as its backbone. The major focus of our design is to derive a class of simple and clean abstractions which result in a generic document structure that applies across various TEX dialects. From the user's point view, the notion of "dialects" is hidden. There are only few object types to deal with. All external programs are controlled at the editor top level and the user interface is coherent and straightforward. The resulting environment has been widely used and has proved very effective in improving user productivity.

1 Introduction

This paper describes a document preparation environment based on GNU EMACS [18] and the TEX [9] family of processors. EMACS is an extensible interactive text editor [17]. TEX and its auxiliary programs are noninteractive document compiling facilities. Both of these systems are widely distributed and are in the public domain. By integrating all TEX-related software with EMACS, we have been able to create an environment which effectively reduces the overhead involved in the edit-format-debug cycle.

Conventional document preparation involving noninteractive processors is usually carried out in an unintegrated fashion. Normally, the user creates the document using a text editor,

[*]Sponsored in part by the National Science Foundation under Grant MCS-8311787, and by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4871, monitored by Space Naval Warfare Systems Command, under Contract No. N00039-84-C-0089.

[†]Additional support has been provided by an IBM Graduate Fellowship.

executes the formatter and other pre- and post-processing programs at the top level of operating system, collects bugs, returns to the text editor to correct them, and the cycle continues until no more syntax errors are found. Then the author has to rely on hardcopy output to fix up the document appearance (semantic errors).

The only close interaction between the user and the system exists in text editing. However, the text editor knows nothing about the source language's syntax, not to mention its semantics. Error checking is dependent completely upon these noninteractive processors. Whenever an error is corrected, the whole document needs to be reformatted. In some circumstances even the entire output needs to be printed in order to catch any problems with the target appearance. A typical scenario is running the ordinary `troff` family of processors [8] with the visual editor `vi` [7] under UNIX. The TEX family offers more flexibility because TEX produces output files in the DVI format[1] [6]. There are standard utilities which can "split" a DVI file into component files, hence it is possible to print individual pages without reformatting the entire document as `troff` would. Furthermore, we have developed output previewers on our workstations, thereby eliminating any hardcopy dependence during the debugging phase.

A customizable text editor like EMACS provides the necessary ingredients for integrating all this together. Using the Lisp subsystem supported by EMACS one can define modes which "understand" properties specific to particular languages. The most common customization is a set of editing facilities such as indentation, abbreviation expansion, delimiter matching, etc. These are not syntax-directed *per se*, but do help the user detect or avoid some errors in an early stage of the cycle. But editing is just one of many aspects of document preparation. There are separate tasks such as spelling checking, bibliography making, cross referencing, indexing, etc. which are either preprocessing or postprocessing relative to the formatting job. Each of these tasks requires at least one stand-alone program to handle an input file which may or may not be a byproduct of formatting.

In our EMACS-based TEX environment, the editor is responsible for as much of this as possible. If something must be handled by an off-line program, it spawns the process so that the job is still under its control. In other words, all the user has to deal with in preparing a document is the interface provided by the special editor mode. This environment facilitates not just editing but the preparation of bibliography and index files as well. With simple abstractions, it hides the notion of different formatter types so that operations with respect to various TEX dialects become generic. It also mimics a separate compilation scheme, yielding an improved debugging cycle. The resulting system is a host of about 8,000 lines of Lisp programs organized as two EMACS modes: *TEX-mode* [4] which handles document preparation in general and *BIBTEX-mode* [3] which helps maintaining bibliography database files in particular. The programs have been distributed to a number of academic and industrial sites since 1985.

In this paper we identify some key issues in the design of this environment. First we review what is available in the unintegrated situation and why integration is desirable. We then show the basic abstractions which turn out to be the most important factor in making the system a coherent working environment. In the following section we discuss major features of our

[1] DVI stands for DeVice Independent format.

environment and explain their implications to batch-oriented document preparation. Finally some novel algorithms are described in the appendices.

2 The Unintegrated Environment

There are some primitive tools for producing documents available from either the UNIX TEX distribution[2] or the UNIX environment itself. But these tools are unintegrated and the collection is incomplete. In several instances, we have developed new software to handle the desired tasks. This section identifies some key issues and important properties of the processors involved.

2.1 Formatting

Before introducing the different TEX-based formatters, we need to clarify the notion of *procedurality*. This has to do with the granularity of control the user is allowed to possess over the formatting functions. Equivalently, one can think of this as the amount of information a system must know *a priori* in terms of the document's style and structure. At one end of the spectrum we have what may be called the pure *procedural* scheme which requires the user to specify exactly *how* the formatting ought to take place. Appearing at the other end is the pure *declarative* (or *descriptive*) scheme in which based on some predefined attributes the user specifies just *what* a document should be and the formatting details associated with various document styles are hidden.

The "purest" form of TEX is called **initex** in which no macros have been defined. But **initex** is too primitive to be very useful by itself. One must define macros based on the **initex** primitives to create higher level abstractions. The various TEX dialects are essentially **initex** preloaded with their corresponding macro packages. There are now four primary TEX dialects:

1. Plain TEX [9]: This is **initex** preloaded with a macro package called **plain.tex**. In the appropriate context, the name TEX often refers to plain TEX. It is by and large a procedural language. No document styles are predefined, hence the user has full control over the document appearance. It has a very nice subsystem for specifying mathematical formulas, but preparing tables is especially difficult.

2. LATEX [10]: This is a more declarative system. One must specify a predefined style for each document. It is in this respect that LATEX is similar to SCRIBE [1]. Many plain TEX commands, if not overridden, are automatically imported to LATEX. The table specification subsystem is much improved over that in plain TEX and is syntactically similar to `tbl` [11].

[2]The distribution is organized by the TEX User's Group, P.O. Box 9506, Providence, RI 02940. Users of different operating systems receive the distribution tape from different sources, however.

3. SLiTeX [10, Appendix A]: This is a variant of LaTeX specially tailored to the production of technical slides. It inherits some of the functionality of LaTeX but is loaded with a host of larger sized fonts for projection legibility.

4. AMS-TeX [16]: In the "procedurality" spectrum, this is in between plain TeX and LaTeX. A richer mathematical subsystem is available since it was designed to be used for typesetting by the American Mathematical Society.

Normally the user formats a document by invoking one of the processors listed above. In the EMACS world, it is unrealistic to create one mode for each TeX dialect because all TeX dialects use the same convention that a source file name terminates with `.tex` extension. EMACS, on the other hand, depends on language-specific file name extensions to load the appropriate mode.[3]. What is needed here is an operator overloading mechanism such that type becomes implicit and relevant operations become generic. By doing so only a single TeX mode is needed and the user is alleviated from remembering what type of document is being processed.

2.2 Spelling Checking

Spelling checking in UNIX can be done by the program `spell` and its auxiliary processors `spellout` and `spellin`. The `spell` program compares each unique word in the input file against a dictionary and returns a file containing the misspelled words. The two auxiliary processors `spellout` and `spellin` respectively checks out and checks in words in the input file against a specified hashed spelling list. They help to maintain a customized dictionary in a hashed form so that "false drops" such as people's names, technical jargon, etc. can be recognized by the spelling checker.

Many syntactic constructs do not have to go through the spelling checker at all. For TeX-based documents, it would be redundant to check all the commands (anything started with a backslash '\'), comments (from '%' to end of line), special symbols, etc. There are two programs (`detex` and `delatex`)[4] which filter out unnecessary words from a plain TeX or LaTeX document as far as spelling checking is concerned. One could conceivably write a third filter for AMS-TeX, but `detex` appears to work satisfactorily.

It is important to integrate the spelling checker and these filters with an editor so that misspelled words may be corrected interactively. It is also desirable to have an integrated on-line dictionary lookup facility so that the user doesn't have to abort the on-going error correction process to find out the right spelling. Furthermore, the system should allow the user to construct customized dictionaries for different documents. The spelling checking subsystem in *TeX-mode* achieves all this based on these utility programs and the basic abstractions given in Section 3.

[3]This is called *autoloading* in EMACS.

[4]Several people have written utilities of this kind. The ones we are using are due to Howard Trickey of AT&T Bell Laboratories. It is available in the UNIX TeX distribution.

2.3 Bibliography Making

The making of a bibliography comprises the following steps:

1. Entering bibliography entries in a database.
2. Making citations to these entries in a document source file. The citations should be in a symbolic form.
3. Extracting the specified entries from database and create the actual bibliography or reference file with respect to the current document.
4. Replacing all symbolic citations in the source by the actual keys that appear in the bibliography/reference file.

LaTeX has a companion system called BibTeX [13] which handles Step 3. BibTeX defines a number of entry types (e.g. `BOOK`, `ARTICLE`, `INPROCEEDINGS`, etc.) and entry fields (e.g. `TITLE`, `AUTHOR`, `PUBLISHER`, etc.) There is a fixed mapping between the types and fields. That is, each entry type is associated with a predefined suite of fields, some of which are mandatory (i.e. unfilled fields are considered errors) while some are optional (i.e. unfilled fields are discarded). Furthermore, it supports several built-in bibliography styles which define the appearance of the resulting bibliography/reference file as well as that of the actual reference entries (numeric or alphabetic);[5] additional styles may be defined by the user.

To resolve citations from symbolic references to actual ones, a four-pass LaTeX/BibTeX is involved. First the program `latex` is used to collect all symbolic references from the source. Then the bibliography processor `bibtex` must be invoked to extract the entries from database files and create the actual bibliography source file. Next `latex` is executed to obtain the symbolic/actual cross reference information. Finally it is used again to do the actual replacement. The actual keys appear in the output file while the symbolic references in the source remain intact.

This bibliography subsystem was designed specifically for LaTeX, but it would be desirable to import the same facility to plain TeX and $\mathcal{AMS}$-TeX documents. Also, the preparation of database entries ought to be automated and a lookup or query interface should be available for the user to make citations in editing sessions. In our environment, this becomes all possible with a preprocessing mechanism built entirely on top of Emacs. None of the processors involved need to be modified.

2.4 Indexing

The following steps are involved in index preparation.

1. Placing index commands (`\index{`*key*`}`) in the document source which may comprise multiple files. An index command takes a single argument: the key to be indexed.

[5] Built-in styles include one that gives an order based on the last name of the first author, one that's based on the citations' order of appearance in the document source, and two other variants.

2. Creating a raw index file whose entries each consists of two arguments: the index key and the page on which the index command appears.

3. Processing the raw index file. This means all index keys are sorted alphabetically. Page numbers under the same key are merged and successive numbers are collected into intervals (e.g. `1, 2, 3, 4, 5` is replaced by `1-5`). Subitems within an entry, if any, are properly handled.

4. Formatting the processed index. The result is the actual index.

LaTeX provides hooks to do tasks associated with Steps 2 and 4. It recognizes the index commands and generates a raw index file as a byproduct of formatting. Also, it has an index environment defined in various styles that allows the alphabetized version to be formatted into the final result. Exporting these hooks to TeX and $\mathcal{A}\mathcal{M}\mathcal{S}$-TeX is relatively straightforward. There are no provisions, however, for handling tasks associated with Steps 1 and 3. As a result, we have designed and implemented a fairly general purpose index processor called `makeindex` that handles not just the index of TeX documents but their glossaries as well. It can also be used as an index postprocessor for any raw index files generated by systems other than TeX. We also implemented a subsystem in *TeX-mode* for placing index commands in the document source in a systematic and efficient way (Step 1). Section 4.4 describes some key features of these systems. A more detailed discussion on the design tradeoffs is given in [5].

2.5 Printing and Previewing

TeX achieves device independence by representing its output in a simple generic format called DVI [6]. A device driver is needed to convert a DVI file to a specific printer language. Specifying a printer option in the UNIX spooling scheme (`lpr`) will invoke the right printer driver. There is a program called `dviselect`[6] which extracts designated pages or page ranges from a given DVI file and produces a new one. We also wrote a tool called `dvisplit` which allows one to print designated chapters or sections from a given DVI file. Together, these tools reduce the printing overhead if debugging is dependent on hardcopy.

To eliminate hardcopy dependence, window system-based previewing tools have been developed for our workstations which include a DVI previewer called `dvitool` [12] for the the SUN window system [2] and a similar program called `dvi2x` [14] which runs under the X window system [15]. Both of these tools read a DVI file on a page by page basis. Specific optimizations are taken into account so that inter- and intra-page scrollings are efficient. In addition to a fairly complete set of scrolling functions, they allow text being displayed to be magnified or shrunk in a wide range of steps. Font information may be queried by simple mouse clicking. There is a repositioning mechanism in both tools that points the document to its original position prior to a reread operation which may be invoked by the user because

[6]This program is written by Chris Torek of Maryland University. It is available in the UNIX TeX distribution.

the document has been reformatted. This simple feature turns out to contribute significantly to the reduction of document debugging overhead.

In our EMACS-based environment, printing and previewing are both controlled by the *TeX-mode* top level, which also keeps track of other operations such as formatting, spelling checking, bibliography making, indexing, etc.

3 Basic Abstractions

The major focus of our design is to derive a class of simple and clean abstractions. There are several aspects to this problem. First, what is the generic document structure that applies across various TeX dialects? Second, what are the objects that must be made explicit and what are their interrelationships? Finally, how can the notion of different dialects be hidden? Also of importance is the user interface which in this context refers to the the set of editing commands that maps the collection of operators to the class of objects. Because so many processors are involved, the critical user interface issue is coherence in terms of command naming and key bindings. We have created two EMACS modes: *TeX-mode* and *BIBTeX-mode*. The former is used to edit TeX documents and to control all the jobs described in the previous section. The latter is used to edit BIBTeX database files which works hand in hand with *TeX-mode* in doing citation lookups and error corrections.

3.1 Document Structure and Separate Processing

There are two levels of structure involved: *source* and *target*. At the source level, *TeX-mode* makes the distinction between a *document* and a *file* by acknowledging that multiple files may be included in a TeX-based document. TeX and LaTeX use the `\input` and `\include` commands to include external files in the current document. This allows us to view a document as a tree of files with edges being the connecting commands. This document tree has a root called the *master file* (level 0) which may include external files (level 1), each of which may in turn include more files (level 2), and so forth. Operations involving the entire document must be started from the master file. The processing sequence is the preorder traversal of the tree. In *TeX-mode*, each individual file has a link to the master file to assure that any global commands initiated in its buffer will always start from the master. By default, a file points to itself as is the case for single file documents. The link to the master file also makes it possible to separately compile any component file or a part of it.

The next important abstraction at the source level is a *file*, or when loaded in EMACS, a *buffer*. Objects of even smaller granularity include *regions* and *words*. A *region* is a piece of text, including any white space, bounded by a marker and the current cursor position (i.e. *point* in EMACS). A *word* in *TeX-mode* is a piece of text with no white space in it. Superimposing any further logical structure is infeasible because such structure would vary in different dialects. LaTeX probably has the most complex logical structure while plain TeX has a very limited one. For our purposes, the file level tree structure plus internal regions and words are sufficient.

At the target level, there are fewer abstractions. One is the entire DVI file which represents all the pages processed. The only other abstraction might be a subsequence of the pages processed. While an arbitrary substring of a DVI file is not a legal DVI file, it is a trivial matter to create one by providing the correct enclosing environment. Normally DVI files themselves are not visited in EMACS because they are binary data. Therefore, in a buffer bound to the TEX source `foo.tex`, the implicit operand for operations such as *preview* and *print* will be `foo.dvi` instead of `foo.tex`. With this abstraction, it is possible to preview or print a DVI file partially as well as in its entirety from an EMACS buffer visiting a corresponding document source.

3.2 Document Types and Generic Operators

TEX-mode maintains the notion of *document type* which may be either TEX, LATEX, SLITEX, or AMS-TEX in our current version. The type information is needed when the user tries to execute operations involving programs which are type-specific, such as invoking a formatter (i.e. `tex`, `latex`, `slitex`, or `amstex`) and using a document filter (i.e. `detex` or `delatex`). However, such information is implicit to the user except for the first time — once specified it will be saved as a comment line in the document to be read by later invocations. The user can also specify a default document type in his EMACS startup file so that even the one time inquiry becomes unnecessary. This is useful for users who never use more than one TEX dialect. In other words, from the user's point of view, operations in *TEX-mode* are generic. For instance, an operation is known as *format* at all times instead of as `tex`, `latex`, `slitex`, or `amstex` under different situations. *TEX-mode* uses operator overloading implicitly by consulting the type information.

3.3 User Interface and Job Control

By and large, the two modes obey the naming convention that a function name consists of three parts: prefix (`tex-` or `bibtex-`), generic operator, and abstract object. The corresponding key binding will be the **C-c** prefix,[7] followed by **C-** and the first letter of the middle part, then the first letter of the last part. One example is the *TEX-mode* function `tex-format-document` with its corresponding key binding being **C-c C-fd**. *BIBTEX-mode* deals with a simpler set of objects such as bibliography entries and their component fields. The function to duplicate the previous entry, for instance, is called `bibtex-dup-previous-entry` which is bound to **C-c C-dp**. There are minor variations of this convention due to the constraint of limited keys, but the usage is generally unform.

This user interface is very simple and gives good mnemonic clues. For example, in *TEX-mode* all spelling functions begin with the **C-c C-s** prefix, bibliography making functions with **C-c C-b**, indexing facilities with **C-c C-i**, formatting with **C-c C-f**, printing with **C-c C-p**, etc. A context-sensitive menu is displayed according to the nature of the current operation. The interfaces to these menus are uniform across the different contexts.

[7]**C-c** means Control **c**. That is, typing the **c** key while holding down the Control key. The notation here is that **C-** stands for the control key.

Every job needed in the entire course of document preparation is initiated from *TEX-mode* or *BIBTEX-mode*. In *TEX-mode*, most generic operators take the current document, file (buffer), or region as an operand, but some apply to words. In *BIBTEX-mode*, objects include the current bibliography file, an entry, and fields in an entry. From the user's point of view, not only the typing overhead is greatly reduced, but the need to swap context both mentally and physically between the editor and operating system top level is eliminated.

4 The Integrated Environment

The integration is achieved by building a system of Lisp programs on top of EMACS. The environment is functionally divided into several subsystems for such tasks as delimiter matching, spelling checking, bibliography making, indexing, job control, etc. Only the most essential functions are loaded initially; subsystems are loaded on demand.[8] Some major system features are as follow.

4.1 Delimiter Matching

TEX-mode supports several delimiter matching schemes. By matching we mean the motion of cursor from the closing delimiter momentarily to the opening delimiter position. Such motion is normally triggered when the closing delimiter is typed. The bouncing cursor usually prevents unbalanced delimiters and thus reduces the overhead in debugging. EMACS has built-in mechanisms for matching delimiter pairs with different opening and closing symbols (e.g. `(...)`, `{...}`, etc.)

Matching becomes more complex when the opening and closing delimiters are identical. For instance, *math mode* and *display math mode* in TEX are respectively represented by a pair of single and double dollar signs (`$...$` and `$$...$$`). Math mode may occur inside display math mode, such as in a table environment, but not vice versa. Matching these types of delimiters is very difficult. A correct mechanism not only has to know which self-inserting `$` or `$$` is an opening delimiter and which is a closing one but also must retain enough information so that the second `$` in `$$` does not match its preceding `$`. In *TEX-mode* we have a mechanism which does this type of matching correctly. A description of the underlying algorithm is included in Appendix A at the end of this paper.

It is a common practice in TEX to include a piece of text in a pair of braces with a leading command. This comes in two forms: `\foo{...}` and `{\foo ...}`. The former means the text in the braces is the argument of `\foo`. The latter means the command `\foo` is setting a global register (e.g. current font type, size, etc.), and the pair of braces denotes its scope. Both cases agree in that syntactically there is a prefix (e.g. `\foo{` and `{\foo`) and a suffix (e.g. '`}`' in both cases). *TEX-mode* provides an assorted set of commands to include neighboring words or the current region in a pair of these "extended delimiters".

One of the most commonly used commands in LATEX is a pair of `\begin` and `\end` that embraces a large piece of text under a certain *environment*. Environments can be nested in

[8]Again, this is done by the *autoload* mechanism in Emacs.

the obvious way, just as in any block-structured language. With several levels of environments in place, proper indentations become essential to readability. *TEX-mode* has a facility that opens and closes LATEX environments automatically with proper indentations inserted.

Finally all delimiter matching schemes mentioned above can be customized. A new pair of delimiters may be defined statically by putting the information in EMACS' profile (`.emacs`) to have it available for every *TEX-mode* session. Alternatively, the user can enter the information interactively to *TEX-mode* so that a new delimiter pair is bound for only one particular EMACS session.

4.2 Spelling Checking

Based on the abstractions mentioned earlier, the spelling of an entire document can be checked and corrected interactively, as can individual buffers (files), regions, and words. These commands all share the same prefix (**C-c C-s**), followed by a one letter suffix of **d**, **b**, **r**, and **w**, respectively. The interface for correcting misspelled words is identical in all four cases: the current erroneous word is displayed, followed by a list of options. Aside from the obvious operations such as scrolling down to the next or up to the previous misspelled word in the list, visiting the next or the previous instance of current error, and query replacing all instances of current error with a new string, there is an option which allows the user to do a dictionary lookup based on arbitrary prefix, infix, or suffix searching. The returned matches will be placed in a new buffer. The user can visit the buffer and select the correctly spelled word which will then be zapped into the replacement prompt and the correction session begins. Correction is based on the standard EMACS query replace mode. That is, for each instance of the pattern found, the user can decide whether or not to replace it by the replacement string.

Except word checking, which is essentially an on-line dictionary lookup facility, other types of objects are checked more or less in the same manner. In the case of a region, the piece of text in question is first copied into a temporary buffer and then screened by the document filter `detex` or `delatex`, with result "piped to" `spell` for checking. This is a special case of the buffer command which does the same thing for the entire buffer (file). In the document case, each component file is checked according to the preorder traversal of the document inclusion tree.

Another import feature of the spelling interface is the support for automatic construction and invocation of customized dictionaries. There are several levels of detail involved. First the user can define a global list of words which are likely to appear in most of his documents but are not included in the system dictionary. Whenever spelling checking is invoked in *TEX-mode*, this *per user* spelling list will be added to the system dictionary and the resulting new dictionary will be the basis of spelling checking.

Perhaps more useful is the *per document* spelling list. This is a list of words created as a result of an interactive spelling checking session. At the end of such a session, an option can be invoked to collect all the uncorrected words (assumed to be correct) plus those defined in the global spelling list and hash them into an extended version of the system dictionary. When the same document's spelling is to be checked later, these words will not appear in the correction list any more. New words can be added to this list incrementally as other

```
@INBOOK{,
=============================== REQUIRED FIELDS ============================
    -------------- Exclusive OR fields: specify exactly one -------------
        AUTHOR = {},
        EDITOR = {},
    -------------- Inclusive OR fields: specify one or both -------------
        CHAPTER = {},
        PAGES = {},
    ------------- Rest of required fields: specify every one ------------
        TITLE = {},
        PUBLISHER = {},
        YEAR = {},
=============================== OPTIONAL FIELDS ============================
        VOLUME = {},
        SERIES = {},
        ADDRESS = {},
        EDITION = {},
        MONTH = ,
        NOTE = {}
}
```

Figure 1: A skeleton bibliography entry of type INBOOK.

component files of the same document are spelling checked. In fact, for each customized dictionary an unhashed list of words is maintained by *TEX-mode* so that words can also be removed from or added to the spelling list by hand. Based on time stamps, the hashed list is applied directly in spelling checking if it is newer than the unhashed list; otherwise the unhashed list is first used to derive the hashed list which is then applied.

The increase in these customized dictionaries' hash table size is negligible because the number of words likely to be added are very small compared to those already incorporated in the system dictionary. Therefore, it would be fair to assume the time required to consult each word against a customized dictionary is equal to that against the system dictionary. The primary gain in maintaining customized dictionaries is a significant saving in user interaction time because fewer "false drops" will appear in the correction list after the first run. Appendix B gives some detailed information on the algorithms used in maintaining customized dictionaries.

4.3 Bibliography Making

This is an area where the two major editing modes *TEX-mode* and *BIBTEX-mode* work hand in hand to provide a friendly on-line bibliography/citation facility. The user uses *BIBTEX-*

mode to prepare BibTeX database files, *TeX-mode* to make citations and to generate actual bibliography files, and the combined system to detect and correct any errors in the database or the citations. The same system not only works with LaTeX for which BibTeX was originally created, but with plain TeX and AMS-TeX documents as well.

4.3.1 Bibliography Database Manipulation

A BibTeX database file is one that has a file name extension of `.bib` and contains one or more BibTeX entries. *BibTeX-mode* uses a form-based user interface for the preparation of these entries. It supports all standard BibTeX bibliography entry types as built-in functions so that to insert a new entry the user only has to specify a type which in this case corresponds to an Emacs function. A skeleton instance of the specified type will be generated automatically with the various predefined fields left empty for the user to fill in. A host of supporting functions such as scrolling, field copying, entry duplicating, ..., etc. is provided to facilitate this content-filling process.

Figure 1 is an instance of the entry type `INBOOK`. The user invokes an entry like this by typing

```
M-x @inbook RET
```

where `M-x` is the command to call on Emacs' Meta-X prompt (i.e. holding down the meta or escape key and type `x`) and `RET` means carriage return. Function name completion is supported at this prompt; hence only the shortest distinguishable prefix needs to be typed before `RET`. In the skeleton entry, banner lines are displayed to give hints regarding the nature of specific fields. These banners, along with all unfilled optional fields, will be removed when a cleanup command is issued in the mode. The cleanup operation will also catch any mandatory but unfilled fields so that correcting errors of this kind does not have to wait till `bibtex` is executed.

There is an extended abbreviation mechanism in the mode. BibTeX has a simple abbreviation scheme: any unquoted text of a field is regarded as an abbreviation and a macro expansion will be performed at compile time. We call this a *field abbreviation*. In *BibTeX-mode*, chunks of fields may be collected to form a *group abbreviation*. A typical use of this is when a number of articles in the same journal or proceedings are to be included in a `.bib` file. In that situation, most of the fields will be identical except perhaps the `TITLE`, `AUTHOR`, and `PAGES` fields. With a group abbreviation defined, the user can invoke a new entry, enter just these three entries and replace others by the group expansion.

One would normally like to put all abbreviation definitions in a central place and not repeat them in each individual `.bib` files. BibTeX itself does not not support file inclusion. What *BibTeX-mode* does is to allow the user to define a list of default abbreviation files, making it possible to browse and interpolate abbreviations defined externally. The interface is very similar to that of the spelling checking subsystem. Same options are bound to the same keys as much as possible.

The user can debug, preview, or create a hardcopy draft of any BibTeX data file. This is done by executing the BibTeX processor on a temporary file which contains citations to every entry of the target .bib file. If there are any errors, a correction mechanism will position the cursor to the spot, prompting for fixes. At the end of a successful session, a draft is created in the formatted form, which can then be previewed or printed.

4.3.2 Symbolic Citations

Each bibliography entry in the database requires a symbolic name such as `knuth:tex`. In the document source, this entry would be referenced by the command `\cite{knuth:tex}`. The user can certainly enter this command manually, but that requires him either to remember what actually appears in the database entry or to manually visit the data file, locate the entry, and look up the name. With the *TEX-mode* lookup facility, neither is necessary. All that is needed is to specify a data file name and a keyword related to the entry such as `Knuth`, `TeX`, the publisher's name, or any regular expression. A list of matching entries will be returned. The user can confirm an entry, in which case the `\cite` command plus the entry name is inserted at current cursor position. Alternatively, a request can be made to show the content of an entry, or to scroll and inspect the next/previous match, to create a new entry, or even to change the search key. A search path may be specified so that when a wildcard file name is given, every data file in the path is looked up.

4.3.3 Bibliography Processing

In the LATEX/BIBTEX combination, the input to the bibliography processor `bibtex` is a file generated by `latex` which contains all citations made in the document. Note that collecting citation entries has nothing to do with formatting *per se*. In *TEX-mode*, these entries are collected prior to formatting. The bibliography processor `bibtex` is then spawned to process these entries. Finally symbolic-to-actual substitutions are done by the editor at the source level. It also interpolates the actual bibliography file generated by `bibtex` at the appropriate place (usually before the end of document). Hidden from the user is a file of cross references to be used to recover symbolic references from actual numbers when new citations are added to the document and a new bibliography/reference file is needed.

There are two important issues with respect to this source-level bibliography making scheme. The first is related to the incremental growth of citations in a document. Recall that all citations are done initially in a symbolic form. A *TEX-mode* bibliography making session will replace all of them by their actual counterparts in the document source. As new citations are entered, the document will have mixed symbolic and actual references. Our approach is to keep the most recent symbolic/actual cross reference information in a file. Thus whenever a new bibliography is called for, the cross reference information is first consulted to recover symbolic citations from actual numbers. The cycle continues by collecting citation keys, spawning `bibtex`, performing symbolic-to-actual substitutions, and so forth.

The second problem has to do with the efficiency of the symbolic-to-actual substitution mechanism. In a text editor, replacing a target string by a source string requires a series of operations composed of (1) locating the target pattern, (2) erasing the target string, and (3) inserting the source string. Locating the target pattern is normally based on string matching.

Since the same entry can be cited at multiple places, the search has to cover each and every file included in the document. This is an extremely expensive operation.

The pattern matching overhead is completely avoided in *TEX-mode*'s symbolic-to-actual substitution mechanism. The trick is that in the first citation collecting sweep, each entry's position[9] is recorded. Moving the cursor to a designated position is very fast. The substitution mechanism processes entries in each component file in reverse order so that current replacement will not destroy the recorded position of the next entry. That is, the last instance of a symbolic citation is visited and replaced by an actual reference first, then the previous entry, and so on.

4.3.4 Evaluation

The integrated bibliography handling facility is superior than what is available in the unintegrated situation. Some noticeable advantages are the following:

- *BIBTEX-mode* alleviates the user from any concerns regarding the format of bibliography entries. The form-based user interface makes the preparation of bibliography database files an easy task.

- Based on the two corporating modes, errors in both the database and document citations can be corrected interactively. The system will position the cursor to the spot in question and the user will have a menu of options to correct the mistake.

- Due to the lookup facility, the user does not have to memorize or type in the exact entry names in order to make citations. The system prompts you one by one the matching entries found in the specified bibliography database. The selected entry will be interpolated into the source automatically.

- Recall that in the LATEX paradigm, a total of three separate passes of `latex` plus an intermediate pass of `bibtex` is required to process the bibliography in a document (see Section 2.3). The bibliography handling subsystem in *TEX-mode* turns this into a strictly preprocessing job relative to formatting. Due to the automatic invocation of `bibtex`, the error correcting facility, and the automatic substitution mechanism, only one or two runs of `latex`'s are needed to produce the final output — depending on the presence of non-citation symbolic references.

- Once the bibliography/reference file has been created, there is an inspection facility which allows the user to examine the content of a citation entry interactively, yielding an effective link between a citation's context and its content. This is available for citation entries in both symbolic and actual forms.

- The same mechanism not only works for LATEX documents for which BIBTEX was originally designed, but for any TEX dialect as well.

[9]In EMACS, each character is assigned an offset relative to the beginning of buffer. This offset is what we mean by the position here.

4.4 Index Preparation

We have designed and implemented two major subsystems for doing indexing in any TEX-based document. One subsystem provides a systematic approach to placing index commands in the document source, while the other deals with transforming the index from raw entries (generated by the formatter) to the final result (sorted form). We describe some of the key features of the two subsystems below. Detailed information can be found in [5].

4.4.1 Index Placing Subsystem

TEX-mode's index placing facility is based on a very simple framework. All the author needs is to specify a *pattern* and a *key*. The editor then finds the pattern, issues a menu of options and inserts the index command, along with the key as its argument, upon the user's request. As a special case, when the pattern and key are identical, neighboring words of the current cursor position or the text in the current region can be inserted as the index argument in one *TEX-mode* command.

There are two query-insert modes to operate with: one based on single key-pattern pair and the other on multiple key-pattern pairs. In the former mode, the user specifies a pattern and a key, and for every instance of the pattern found, he decides whether to insert the index command with the specified key, or a variant of it. In the latter mode, each key-pattern pair in a global list is processed in a way identical to that of the former mode.

Placing index commands is a task that has been performed traditionally in an ad hoc fashion. It is often tedious and time-consuming. Our facility offers a systematic and efficient approach to this effort. It has been used in producing indexes for a book and a number of manuals and has proved very useful and effective.

4.4.2 Index Processor

Our index processor `makeindex` transforms raw index entries generated by the formatter into the final index file. The tasks performed in this transformation include permutation (entries are sorted alphabetically), page number merging, multi-level indexing (three levels of subindexes are recognized), style handling (customizable input and output formats), and various special effects such as cross referencing (*see* and *see also*), setting page numbers in different fonts, etc.

Due to the style handling facility, `makeindex` is largely independent of the typesetting system and independent of the format being used. There is an interface in *TEX-mode* to `makeindex` much the same as that to other external programs such as `spell` and `bibtex`.

4.5 Formatting

In *TEX-mode*, the user can format the entire document, a component file in a document, or an arbitrary region in a buffer. All of these share the same command prefix (**C-c C-f**) and are distinguished by the mnemonic suffix of **d**, **b**, and **r** which denote, respectively, the entire document, the current buffer, and the current region. Given the master file linkage in each component file, formatting the entire document is relatively straightforward. As mentioned earlier in Section 3.1, the document master file will automatically be visited from where the format-document operation will be initiated.

Separately formatting component files is more involved. The notion of master file plays an important role in separate formatting. A document *preamble* and similarly a *postamble* can be associated with the master file to contain the document's global context. To separately compile a component file or its subregion, a mechanism is available in *TEX-mode* that includes in a temporary file the document's preamble and postamble with the selected text inserted in between. The system will then run the formatter on this temporary file. This technique is primarily for debugging purposes as there is no provision for linking separately generated DVI files into one big DVI file.[10] However, for users wanting only a quick look at a relatively small portion of a document in the debugging phase, this automatic facility turns out to be of value.

A similar mechanism to selectively format parts of the document is available in LATEX. The command `\includeonly` appearing in the beginning of the document specifies to LATEX that only its arguments are to be formatted. Each of these should be an argument of the command `\include` which interpolates a component file at where it appears. Like our facility in *TEX-mode*, unless these component files are specifically set up to start at a new page, separately formatted files cannot be merged in any obvious way.

Our separate formatting mechanism has a number of advantages over that of LATEX. First, it works for any TEX dialect as long as the file inclusion command is known (so that the file inclusion tree can be built). Second, our granularity is finer; any piece of text in a document can be separately formatted for a quick look at its result. From the point of view of document debugging, this is more practical than always having to reformat an entire component file. Third, there is no need in our mechanism to change any command arguments. In LATEX, the user has to make a change to the argument list of `\includeonly` each time a different component file is to be separately formatted.

4.6 Previewing and Printing

In *TEX-mode*, separate processing exists not just in document formatting, but in its previewing and printing. Similar to the format command, the commend **C-c C-d** followed by either **d**, **b**, or **r** formats the object given and displays its result using the DVI previewer available in a particular window system.

The DVI file associated with the current document can be previewed or printed in its entirety. *TEX-mode* can also invoke `dviselect` or `dvisplit` so that a partial document can be obtained. When this partial previewing/printing is requested, the system will prompt the user for a string of arbitrary page ranges. These pages will then be extracted to form a new DVI file on which the previewer or the print scheme operates.

[10]It would be feasible and in fact easy to combine these separately formatted output files to yield the final result if each component starts at a new page. This is normally not the case for certain document styles.

4.7 Miscellaneous

There are a number of small features which enhance efficiency substantially. In both *TEX-mode* and *BIBTEX-mode*, there is a mechanism which positions the cursor to the the exact line and column where an error occurs. Also supported is the ability to comment or uncomment out a region of text. This is a useful feature for something which is relevant in some situations but irrelevant in others as far as formatting is concerned. Both `dvitool` and `dvi2x` have the ability to reposition the newly read document to the page where it previously was at. This saves a lot of user time as that particular spot may be the only place of interest at a particular document debugging stage. When a formatting operation is started, the old pending session, if any, will automatically be terminated before the new session begins. All of these features are individually straightforward in concept, but together they contribute to a considerable gain in user productivity.

5 Conclusions

In the work described in this paper, we did not attempt to rewrite an editor or a document formatter. Rather, we focused on integrating TEX and related programs under a coherent user interface, using EMACS as the backbone. A customizable editor like EMACS, which offers not just editing functionality but computing capabilities in general and full Lisp support in particular, provides the vital glue to integrating a host of noninteractive processors together. Mode-specific editing is a standard practice in EMACS. Our approach goes beyond mode-specific editing, providing a number of useful document processing facilities in an integrated fashion.

With very simple abstractions, we have been able to create an effective document preparation environment that is robust enough to handle different TEX dialects with diverse characteristics. There are only two fundamental concepts: *document type* and *master file*. But the system based on these concepts produces surprisingly flexible and powerful results. Yet these concepts are transparent to the user most of the time. Under our basic construct, utilities originally developed for one version of TEX have been easily adopted to other dialects. Most importantly, the user can enjoy substantial productivity improvements using this integrated system.

What we see as an important implication of *TEX-mode/BIBTEX-mode* is the importance of a programmable editor layer in document preparation environments. Supporting a programming language like Lisp at the editor level has significant benefits in terms of document pre- and post-processing. Tasks like spelling checking, bibliography making, indexing, job control, etc. need not be built into the system kernel. Instead, they can be separated and implemented on top of the editor's programming substrate. The same functionality can be realized this way, but at the same time better interactive behavior is achieved and greater flexibility is created. This has been proved by the system we built. We believe the techniques will apply to more elaborate document preparation environments as well.

6 Acknowledgements

We are grateful to the following people who have made valuable comments and suggestions during various stages of the development: Fred Douglis, Jim Larus, Rusty Wright, Art Werschulz, and Ben Zorn.

References

[1] *Scribe Document Production System User Manual.* Unilogic Ltd., Pittsburgh, Pennsylvania, April 1984.

[2] *SunView Programmer's Guide, Release A of 17.* Sun Microsystems, Mountain View, California, February 1986.

[3] Pehong Chen. *GNU Emacs BIBTEX-mode.* Technical Report 87/317, Computer Science Division, University of California, Berkeley, California, 1986.

[4] Pehong Chen. *GNU Emacs TEX-mode.* Technical Report 87/316, Computer Science Division, University of California, Berkeley, California, 1986.

[5] Pehong Chen and Michael A. Harrison. *Automating Index Preparation.* Technical Report 87/347, Computer Science Division, University of California, Berkeley, California, March 1987. Submitted for publication.

[6] David Fuchs. Device independent file format. *TUGBoat*, 3(2):14–19, October 1982.

[7] William Joy. *An Introduction to Display Editing with Vi.* 1980. Appears in UNIX 4.2BSD User's Manual.

[8] Brian W. Kernighan, Michael E. Lesk, and Joseph F. Ossanna. Document preparation. *The Bell System Technical Journal*, 57(6):2115–2135, July–August 1978.

[9] Donald E. Knuth. *The TEX Book.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1984. Reprinted as Vol. A of *Computers & Typesetting*, 1986.

[10] Leslie Lamport. *LaTEX: A Document Preparation System. User's Guide and Reference Manual.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[11] Michael E. Lesk. *Tbl—A Program to Format Tables.* Computer Science Techincal Report 49, AT&T Bell Laboratories, Murray Hill, New Jersey, September 1976. Also available in UNIX User's Manual.

[12] Jeffrey W. McCarrell. *An Overview of DVITool.* VORTEX internal report, Computer Science Division, University of California, Berkeley, California, December 1986.

[13] Oren Patashnik. *BIBTEXing.* Computer Science Department, Stanford University, Stanford, California, March 1985.

[14] Steve Procter. *Documentation on DVI2X*. VORTEX internal report, Computer Science Division, University of California, Berkeley, California, March 1987.

[15] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 6(3), 1987.

[16] Michael D. Spivak. *The Joy of TEX*. American Mathematical Society, 1985.

[17] Richard M. Stallman. EMACS: the extensible, customizable self-documenting display editor. In *Proc. of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 147–156, Portland, Oregan, June 8–10 1981. A somewhat extended version appears in *Interactive Programming Environments*, Barstow et al. (eds.), McGraw-Hill Book Company, 1984, pp. 300–325.

[18] Richard M. Stallman. *GNU Emacs Manual, Fourth Edition, Version 17.* Free Software Foundation, Cambridge, Massachusetts, February 1986.

A Matching Identical Delimiters

Automatic delimiter matching refers to the momentary bouncing of cursor from the self-inserting closing symbol to the opening symbol position. Matching delimiters with distinct opening and closing symbols is relatively straightforward in most screen-oriented text editors. In EMACS, for example, one simply modifies syntax entries to achieve this effect.

Matching delimiters with identical opening and closing symbols (call them `id_delim`'s) is virtually unsupported in most text editors. In TEX-*mode*, there is an underlying facility for matching arbitrary `id_delim`'s — the most obvious ones being `"..."`, `|...|`, and of course, TEX's math mode (`$...$`) and display math mode (`$$...$$`).

For each class of `id_delim`'s, TEX-*mode* maintains a buffer-specific doubly-linked list of markers pointing to their respective positions. Whenever a new `id_delim` is typed, its marker is inserted into the list according to the positional order. During the search for the correct insertion point, every preceding `id_delim` in the list is verified. The marker is removed from the list if it no longer points to an `id_delim` (i.e. the corresponding symbol has been deleted from the buffer). Also maintained in this insertion search is an even-odd count which designates the parity of the newly entered `id_delim`. If it is an even-numbered delimiter, bouncing will take place; otherwise, only its symbol is inserted in the buffer.

A fundamental assumption is that the text between the opening and closing `id_delim`'s always stays in one paragraph. The list of pointers is reconstructed whenever a new paragraph is started. In TEX, there is usually a blank line between paragraphs; thus locating the beginning of a paragraph is quite straightforward in a text editor like EMACS. Based upon this assumption a significant amount of searching overhead is avoided in maintaining the list.

In general, this scheme works for any `id_delim`'s. The situation becomes more complicated in TEX's two mathematical modes where pairs of single and double dollar signs can be mixed. In addition to performing what has been described previously, a two-character look-behind is necessary for every `$` typed to distinguish the two modes. Basically, `$...$`

within `$$...$$` is legal in TEX, but not the opposite. A heuristic is involved to determine when to coerce `$` to `$$`.

B Maintaining Customized Dictionaries

An unhashed spelling list is a file (the `.usl` file) containing words assumed to be correct. A hashed spelling list is a file (the `.hsl` file) which contains a hashed version of the words listed in the `.usl` file. The `.hsl` file is machine dependent under UNIX; a hash table created on machine A will not work on machine B. On the contrary, the `.usl` file is machine independent because it is simply a list of words in ASCII representation. There are two levels of spelling lists:

1. *per user* spelling lists. The following code can be included in one's Emacs profile:

```
(setq tex-usl-default <file-name>)
(setq tex-hsl-default <file-name>)
```

2. *per document* spelling lists. For a document rooted at `foo.tex`, `foo.usl` and `foo.hsl` will be created at the end of a spelling correction session upon request.

The algorithm that determines which file to use as the spelling list is as follows. Again, for a document rooted at `foo.tex`,

1. If `foo.hsl` exists and is newer than `foo.usl`, it is used as the spelling list.

2. If `foo.hsl` doesn't exist and `tex-hsl-default` is non-nil, the file bound to this default is used.

3. Otherwise, if `foo.usl` exists, a new `foo.hsl` is created by adding words in the file bound to `tex-usl-default`, if any, and those in `foo.usl` to the system hashed list (`/usr/dict/hlista`).

4. Otherwise, no customized spelling list is used.

At the end of a spelling correction session, the user can invoke an incremental spelling list saving facility. First, an option to reexamine uncorrected words is available; the user can remove whatever not supposed to go into the list. Then, the incremental saving takes place. The algorithm is as follows.

1. If `foo.hsl` exists and is newer than `foo.usl`, uncorrected words from this buffer are added to the hashed list.

2. If `foo.usl` exists, uncorrected words are merged into the file; otherwise a new file `foo.usl` is created to hold these words.

3. If `foo.hsl` doesn't exist, a new `foo.hsl` is created by adding words in the new `foo.usl` and those in the file bound to `tex-usl-default`, if any, to the system hashed list (`/usr/dict/hlista`).