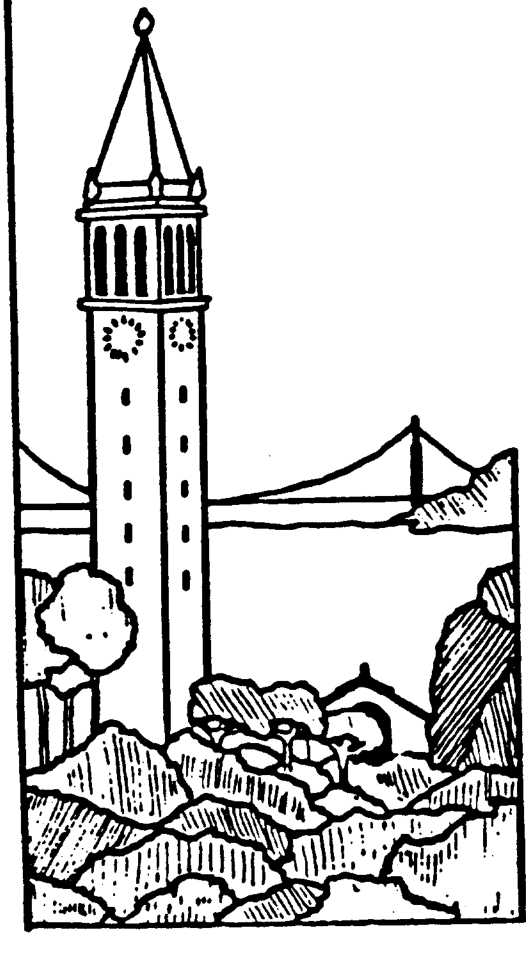
Automating Index Preparation

Pehong Chen

Michael A. Harrison



Report No UCB/CSD 87/347

March 23, 1987

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Automating Index Preparation*

Pehong Chen†
Michael A. Harrison

Computer Science Division
University of California
Berkeley, CA 94720

March 23, 1987

Abstract

Index preparation is a tedious and time-consuming task. In this paper we indicate how the indexing process can be automated in a way which is largely independent of a specific typesetting system and independent of the format being used. Fundamental issues related to this process are identified and analyzed. Specifically, we describe a framework for placing index commands in the document and a general purpose index processor which transforms a raw index into an alphabetized version. The resulting system has proved very useful and effective in producing indexes for a book and several technical manuals. An evaluation of our system against indexing facilities available across a variety of document preparation environments is given.

1 Introduction

Although there has been a great deal of activity in electronic publishing [6], there are still aspects of document composition that have not been fully automated. One of the most time-consuming concerns is the preparation of the index. In ordinary books, an index is an important feature which allows a reader to access essential information easily. A poor index with many omissions or poorly chosen concepts actually detracts from other aspects of the book. For highly complex technical material which might include computer programs, it is highly desirable to have different kinds of indices which might reference even the identifiers of a programming language. A good example of an elaborate indexing scheme can be found in Knuth's TeX program [15] and his WEB system [13] in general. For computer programs like these, completeness is essential and the accuracy of traditional hand methods will not suffice.

*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.

Standard authors' guides such as [2] recommend that index terms be marked on page proofs or on a separate set of galley proofs. The traditional method is to use 3×5 cards, appropriately called index cards. A page number is added to a card when a reference is encountered. Sorting is done by hand and the process is tedious and error-prone. The breaking of a rubber band or the dropping of a box of cards can be traumatic events. Computers offer an opportunity to significantly reduce the amount of labor invested in this process while noticeably improving the quality of the resulting index.

In this present paper, we indicate how the indexing process can be automated in a way which is largely independent of a specific typesetting system and independent of the format being used. Specifically, we describe a framework for placing index commands in the document and a general purpose index processor which transforms a raw index into an alphabetized version. These concepts have been implemented as part of an extensive authoring environment [11]. This includes a suite of Lisp programs for the index placing facility and a C program for the index processor. The resulting system has been successfully used in producing indexes for a book [10] and a number of technical manuals.

We focus on issues under both language-based and direct manipulation [12,19] paradigms. In a language-based system, the user specifies the document with interspersed commands which is then passed to a formatter and the output is obtained. In a direct manipulation environment[1] the user manipulates the document output appearance directly by invoking built-in commands available through menus and buttons. There is no such concept as the document specification language in direct manipulation systems.

The problem and our solutions are presented in a top-down fashion: first we discuss the desirable features of an index processor, followed by some design and implementation considerations for such a processor — taking into account its general-purpose nature, then the framework with which the author enters index commands or tags with much reduced overhead. The examples we show are all in LaTeX [16], a high-level document preparation language based on TeX [14]. The model, however, is not restricted to any particular formatting language, nor to the language-based paradigm only. Next we examine some issues which are unique to indexing under electronic documentation environments that don't seem to find appropriate counterparts in traditional printed material. Finally we evaluate our indexing facilities against those available in other formatting systems such as SCRIBE [5], TROFF [18], and some WYSIWYG environments.

2 Index Preparation

Index preparation is a process involving the following steps:

I. Placing index commands in the document source, which presumably comprises multiple files. An index command takes a single argument: the key to be indexed.

[1]Some people like to call this type of system WYSIWYG (what-you-see-is-what-you-get). We use the terms direct manipulation and WYSIWYG interchangeably in the paper.

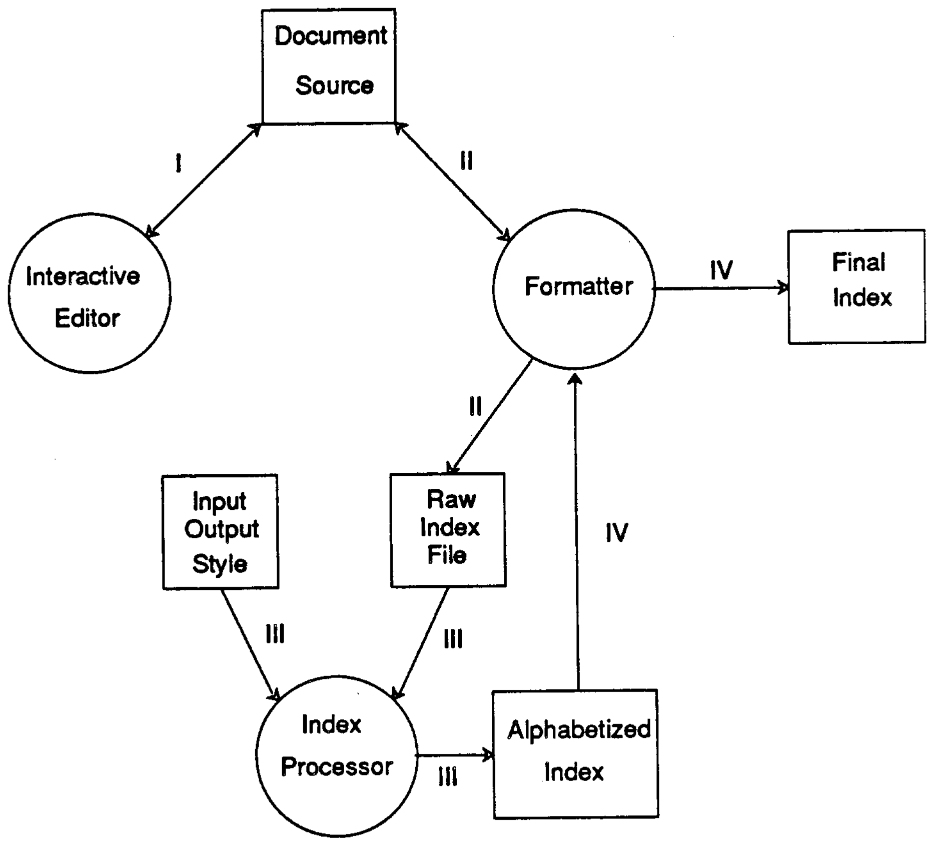


Figure 1: The sequential flow of index processing. Circles in the picture represent processors, squares are documents or auxiliary files. In Step I, the author uses an editor to place index commands in the document. In Step II, a raw index is generated as a by-product of formatting. In Step III, this raw index together with some optional style information are taken as input to the index processor and an alphabetized version is created. Finally in Step IV, the index is formatted to yield the ultimate result.

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

III. 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.

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

The idea is illustrated in Figure 1, where roman capitals I–IV marking the edges correspond to the four steps mentioned here. It is clear that this is a highly sequential procedure, for the input to one step depends upon the result from the previous one.

Figure 2 demonstrates an example as a stepwise development of the process. In LaTeX all commands begin with a backslash (\).[2] Figure 2.a shows some occurrences of index commands (`\index`) in the document source, with corresponding pages listed on the left. The page number is not part of the source file since at file-preparation time, we do not know on which page a given textual material will eventually appear. Figure 2.a includes these numbers just to indicate that ultimately these entries would appear on those pages. Figure 2.b shows a raw index file is generated by LaTeX. After running through the index processor, it becomes an alphabetized index with commands specifying a particular output appearance (Figure 2.c). The result after formatting is shown in Figure 2.d.

Based on the example given in Figure 2, these four steps are explained below, where Steps I and III are further expanded in Sections 4 and 3, respectively. Issues involved in Steps II and IV are less complex and are only covered in this section.

2.1 Placing Index Commands

Step I deals with placing index commands in the document. In a language-based environment, the commands can simply be inserted in the document source with a text editor. They will be utilized by the formatter in generating raw index entries (Step II) but will contribute nothing to the output appearance as far as the corresponding pages are concerned.

In a direct manipulation system, index commands cannot be entered directly in the document under manipulation. A possible solution is to put them in "shadow pages" instead of the document output representation. A shadow document is the original document plus special tags which, among other things, mark logical objects like comments, bibliographical citations, cross references, indexes, etc. These tags are essential to document composition but do not correspond to physical appearance in their original forms. Upon request the corresponding markers of these tags can be displayed along with the original document for editing purposes. For the user's visual cue, each type of tags can be represented by a different marker symbol. Normally for each tag entered in the document, an embedded annotation can be specified. An additional window can be created to show the associated annotation if necessary. This

[2] We should say in TeX in general, for LaTeX is a dialect of TeX.

Page iv:	`\index{alpha}`	`\indexentry{alpha}{iv}`
Page 1:	`\index{alpha}`	`\indexentry{alpha}{1}`
Page 2:	`\index{alpha}`	`\indexentry{alpha}{2}`
Page 3:	`\index{alpha}`	`\indexentry{alpha}{3}`
Page 11:	`\index{alphabeta|see{beta}}`	`\indexentry{alphabeta|see{beta}}{11}`
Page 14:	`\index{alpha@{\it alpha\/}}`	`\indexentry{alpha@{\it alpha\/}}{14}`
	`\index{beta|bold}`	`\indexentry{beta|bold}{14}`
Page 22:	`\index{alpha!beta!gamma}`	`\indexentry{alpha!beta!gamma}{22}`
Page 38:	`\index{alpha!delta}`	`\indexentry{alpha!delta}{38}`

```
\begin{theindex}

\item alpha, iv, 1-3
  \subitem beta
    \subsubitem gamma, 22
  \subitem delta, 38
\item {\it alpha\/}, 14
\item alphabeta, \see{beta}{11}

\indexspace

\item beta, \bold{14}

\end{theindex}
```

```
alpha, iv, 1–3
    beta
      gamma, 22
    delta, 38
alpha, 14
alphabeta, see beta

beta, 14
```

Figure 2: An example in LaTeX showing the stepwise development of index processing. (a) *Top Left*: Occurrences of index commands in the document source. Note that page numbers are unknown at the time of input when a source-based formatter like LaTeX is used. We include page numbers here simply to illustrate where each instance will occur. (b) *Top Right*: raw index file generated by LaTeX. (c) *Bottom Left*: alphabetized index file. (d) *Bottom Right*: formatted final index.

shadow document approach is widely adopted by WYSIWYG systems such as Xerox STAR [1], FRAME MAKER [3], and MICROSOFT WORD 3.0 [4].

The primary issue in step I for both paradigms is whether or not a systematic mechanism can be derived to facilitate the entering of index commands or tags. Section 4 gives details of a general model we have designed which accomplishs this task.

2.2 Generating the Raw Index

Step II is concerned with attaching the current page number to each index command placed in the document. The command used in the generated raw index file may be renamed (in our example, it is changed from `\index` to `\indexentry`). The entries are in the exact order in which they appear in the document source. Thus as long as the current page number is accessible, be it language-based or direct manipulation, generating raw index entries is relatively straightforward.

There are minor differences between the two paradigms in this step, however. The generation of raw index entries in a language-based system, like formatting itself, is by and large a "batch job". In a WYSIWYG editor, it is easier to maintain the list of raw index entries incrementally because the document being manipulated is always formatted so the page number is always current.

2.3 Index Processing

A number of issues are of interest in the processing of raw index entries. We describe some high-level issues below with references to the example given in Figure 2; details about how these tasks can be realized are postponed until Section 3.

1. *Permutation.* Index entries are sorted alphabetically (Figure 2.c). The index processor must differentiate among different types of keys such as strings, numbers, and special symbols. Upper and lower case letters should be distinguished. Furthermore, it may be necessary to handle roman, arabic, and alphabetic page numbers.

2. *Merging.* Different page numbers corresponding to the same index key are merged into one list. Also, three or more successive page numbers are abbreviated as a range (as in the case of `alpha, iv, 1-3`, Figure 2.c).

3. *Subindexing.* Multi-level indexing is supported. This refers to that entries sharing a common prefix are grouped together under the same prefix key. The special symbol '`!`' serves as the level operator in the example (Figure 2.a and 2.b). Primary indexes are converted to first level items (the `\item` entries in Figure 2.c) while subindexes are converted to lower level items (e.g. `\subitem` or `\subsubitem` entries in Figure 2.c).

4. *Actual Field.* The distinction between a *sort key* and its *actual field* is made explicit. Sort keys are used in comparison while their actual counterparts are what end up being placed in the printed index. In the example the '`@`' sign is used as the actual field

operator which means its preceding string is the sort key and its succeeding string is the actual key (e.g. the `\index{alpha@{\it alpha\/}}` in Figure 2.a). The same sort key with and without an actual field are treated as two separate entries (cf. `alpha` and *alpha* in the example). If a key contains no actual operator, it is used as both the sort field and the actual field.

The separation of a sort key from its actual field makes entry sorting much easier. If there were only one field, the comparison routine would have to ignore syntactic sugar related to output appearance and compare only the "real" keywords. For instance, in `{\it alpha\/}`, the program has to ignore the font setting command `\it`, the italic correction command `\/`, and the scope delimiters `{}` and concentrate only on the string `alpha`. In general, it is impossible to know all the patterns that the index processor ought to ignore. But with the separation of the fields, the sort key is used as a verbatim string in comparison; any special effect can be achieved via the actual field. The comparison algorithm has nothing to ignore and is thus simplified.

5. *Page Encapsulation.* Page numbers can be encapsulated using the '`|`' operator. In the example, page 14 on which `\index{beta}` occurs is set in boldface, as represented by the command `\bold`. The ability to set page numbers in different fonts allows the index to convey more information about whatever is being indexed. For instance, the place where a definition occurs can be set in one font, its primary example in a second, and others in a third.

6. *Cross Referencing.* Some index entries make references to others. In our example the `alphabeta` entry is a reference to `beta`, as indicated by the *see* phrase. However, the page number disappears after formatting (Step IV), hence it is immaterial where index commands dealing with cross references like *see* occur in the document. This is a special case of page encapsulation (`see{beta}` appears after the '`|`' operator). Variations like *see also* which gives page numbers as well as references to other entries will work in the same way.

7. *Input/Output Style.* In order to meet the design goal of being formatter- and format-independent, the index processor must be able to handle a variety of formats. There are two reasons for considering this in the input side: (1) Raw index files generated by systems other than LaTeX may not comply to the default format. (2) The basic framework established for processing indexes can also be used to process other objects of similar nature (e.g. glossaries). But these other objects will certainly have a different keyword (e.g. `\glossaryentry` as opposed to `\indexentry`) in the very least. Similarly in the output side the index style may vary for different systems. Even within the same formatting system, the index may have to look differently under different publishing requirements. In other words, there must be a way to inform the processor what input format to expect and in what output style should the final index be generated.

2.4 Index Formatting

Two key issues in this last step are (1) support for multiple styles and (2) formatting independence. First, the formatting style macros used in Step III output must be defined. In our example, the global environment (`\begin{theindex}...\end{theindex}`) tells LaTeX to use a two-column page layout. Each `\item` is left justified against the column margin and each `\subitem` is indented by 20 points, `\subsubitem` by 30, etc. There is a vertical space bound to `\indexspace` inserted before the beginning of a new letter (e.g. before `beta`).

The formatting independence problem refers to whether or not the final index can be formatted independently with respect to the entire document. Indexing is supposed to be the last step of document preparation, which means it is attempted only when the entire document is finalized. It is desirable to be able to generate the index without reformatting the entire document. In order to do separate formatting, the global context must be known. In our design, this is made possible by the customizable style facility. One can redefine `preamble` and `postamble` to invoke a style consistent with the original document.

The other information needed to perform effective separate formatting is the starting page number for the index. A related style issue is involved here. Some styles require that the index start on an even or odd page number. In either case, there must be provisions for including the correct starting page number in the pre-formatted version of index.

3 Index Processing

The index processor performs the tasks indicated in Section 2.3 — permutation, page number merging, subindexing, style handling, and other special effects — in multiple passes. First the input format and output style are scanned and analyzed. Entries in the input file are then processed. Next, all legal entries are sorted. Finally the output index is generated in the last pass. We examine each of these passes closely in this section.

3.1 Input Format

Table 1 is a summary of the input format which consists of a list of <*specifier*, *attribute*> tuples. These attributes are the essential tokens and delimiters needed in scanning the input index file. Default string constants are enclosed in double quotes (`"..."`) while character constants are in single quotes (`'x'`). The user can override the default value by specifying a specifier and a new attribute in the style file. The attribute of `keyword` is self-explanatory; those corresponding to `arg_open` and `arg_close` denote the argument opening and closing delimiters, respectively. The meanings of special operators such as `level`, `actual`, and `encap` have been covered earlier in Section 2.3.

The two range delimiters `range_open` and `range_close` are to be used in accordance with the `encap` operator. When `range_open` immediately follows `encap` (i.e. `\index{...|(...}`) it tells the index processor that an explicit range is starting. Conversely `range_close` signals the closing of a range. In our design, three or more successive page numbers are abbreviated as a range implicitly. The two range delimiters enforce an explicit page range formation. This

specifier	*attribute*	*default*	*meaning*
`keyword`	*string*	`"\\indexentry"`	index command
`arg_open`	*char*	`'{'`	argument opening delimiter
`arg_close`	*char*	`'}'`	argument closing delimiter
`range_open`	*char*	`'('`	page range opening delimiter
`range_close`	*char*	`')'`	page range closing delimiter
`level`	*char*	`'!'`	index level delimiter
`actual`	*char*	`'@'`	actual key designator
`encap`	*char*	`'|'`	page number encapsulator
`quote`	*char*	`'"'`	quote symbol
`escape`	*char*	`'\\'`	symbol which escapes `quote`
`page_compositor`	*string*	`"-"`	composite page delimiter

Table 1: Input style parameters.

makes it possible to index an entire section or a large piece of text related to a certain concept without having to insert an index command in every single page.

The `quote` operator allows one to escape any symbols. Thus `\index{foo"@goo}` means a sort key of `foo@goo` rather than a sort key of `foo"` and an actual key of goo. As an exception, `quote`, when preceded by `escape` (i.e. `\index{...\"...}`), does not escape its succeeding letter. This special case is included because `\"` is the umlaut command in TEX. Requiring `quote` itself to be quoted in this case (i.e. `\""`) is feasible but somewhat awkward. But the additional rule here is that `quote` and `escape` must be distinct.

A page number can be a composite of one or more fields separated by a certain delimiter bound to `page_compositor` (e.g. II-12 for page 12 of Chapter II). This attribute allows the lexical analyzer to separate these fields, making the sorting of page numbers easier.

Finally the meaning of white space is something related to lexical analysis but not listed in Table 1. There are two possibilities. The most obvious approach is to treat every index key as a *verbatim* item and return whatever is given, including each white space. These keys will then be compared with white space in place. A somewhat more realistic approach called *white space compression* ignores all leading and trailing blanks as well as the white space immediately preceding or following any special symbol discussed above. Interword white space may be compressed into a single blank by the scanner. Our index processor implements both schemes, taking white space compression as the default case and white space verbatim as optional.

specifier	*attribute*	*default*	*meaning*
preamble	*string*	"\\begin{theindex}\n"	index preamble
postamble	*string*	"\n\n\\end{theindex}\n"	index postamble
setpage_prefix	*string*	"\n \\setcounter{page}{"	page setting command prefix
setpage_suffix	*string*	"}\n"	page setting command suffix
group_skip	*string*	"\n\n \\indexspace\n"	intergroup vertical space
lethead_prefix	*string*	""	new letter heading prefix
lethead_suffix	*string*	""	new letter heading suffix
lethead_flag	*number*	0	flag designating new letter
item_0	*string*	"\n \\item "	level 0 item separator
item_1	*string*	"\n \\subitem "	level 1 item separator
item_2	*string*	"\n \\subsubitem "	level 2 item separator
item_01	*string*	"\n \\subitem "	levels 0/1 separator
item_x1	*string*	"\n \\subitem "	levels x/1 separator
item_12	*string*	"\n \\subsubitem "	levels 1/2 separator
item_x2	*string*	"\n \\subsubitem "	levels x/2 separator
delim_0	*string*	", "	level 0 key/page delimiter
delim_1	*string*	", "	level 1 key/page delimiter
delim_2	*string*	", "	level 2 key/page delimiter
delim_n	*string*	", "	inter page number delimiter
delim_r	*string*	"--"	page range designator
encap_prefix	*string*	"\\"	page encapsulator prefix
encap_infix	*string*	"{"	page encapsulator infix
encap_suffix	*string*	"}".	page encapsulator suffix
page_precedence	*string*	"rnaRA"	page type precedence
line_max	*number*	72	maximum line length
indent_space	*string*	"\t\t"	indentation for wrapped lines
indent_length	*number*	16	length of indentation

Table 2: Output style parameters.

According to our design, the minimum data structure needed to hold the raw index is an array of entries, each of which is an aggregate of $2n$ fields for the index key, where n is the maximum number of index levels. Half of these fields will be holding sort keys for levels 0 through $n-1$; the other half will contain their corresponding actual keys. Also, some additional fields must be included to hold the literal page number, its corresponding numeric counterpart, and the page encapsulator string. After lexical analysis, an input index entry is decomposed into these fields, getting ready to be sorted.

3.2 Output Style

Table 2 summarizes the output style parameters. Again, it is a list of <*specifier*, *attribute*> pairs. In the default column, '\n' and '\t' denote a new line and a tab, respectively. We further divide these parameters into the following groups:

1. *Context.* Together, `preamble` and `postamble` define the context in which the index is to be formatted.

2. *Starting Page.* The starting page number can either be supplied by the user or retrieved automatically from the document transcript. In either case, this number can be enclosed with `setpage_prefix` and `setpage_suffix` to yield a page number initializing command.

3. *New Group/Letter.* The string bound to `group_skip` denotes the extra vertical space needed when a group is started. For a group beginning with a different letter, the parameters `lethead_prefix` and `lethead_suffix` (both with a default nil string) denote the group heading. The flag `lethead_flag` has a default value of 0, which means other than `group_skip` nothing else will be inserted before the group. On the other hand, if this flag is positive, the strings bound to `lethead_prefix` and `lethead_suffix` will be inserted with an instance of the new letter in uppercase in between. Similarly a lowercase letter will be inserted if the flag is negative.

4. *Entry Separators.* This group includes everything with the `item_` prefix. First, `item_`i denotes the command and indentation to be inserted when a key is started from a level greater than or equal to i. Second, `item_`ij has a similar meaning, but with $i = j - 1$. Finally the two `item_x`j's are included to handle the situation where the parent level has no page numbers. Some styles require cases like these be different from those with page numbers.

   Table 2 depicts a system which supports three levels of subindexing. In general, suppose n is the number of index levels supported, there will be n `item_`i's $(0 \leq i \leq n-1)$, $(n-1)$ `item_`ij's $(1 \leq j \leq n-1, i = j-1)$, and $(n-1)$ `item_x`j's $(1 \leq j \leq n-1)$.

5. *Page Delimiters.* Each level has a key/page delimiter which defines what is to be inserted between a key and its first page number. Inter-page delimiter is specified by `delim_n`. Range designator is given by `delim_r`.

6. *Page Encapsulator.* The attributes corresponding to `encap_prefix`, `encap_infix`, and `encap_suffix` form what is to be placed into the output when an encapsulator is specified for a certain entry. Suppose `foo` is the specified encapsulator and `N` is the page number, the output sequence is

```
encap_prefix  foo  encap_infix  N  encap_suffix
```

7. *Page Precedence.* Five different types of numerals are supported by most systems for page numbering. These are lowercase roman (`r`), numeric or arabic (`n`), lowercase alphabetic (`a`), uppercase roman (`R`), and uppercase alphabetic (`A`). The string bound to `page_precedence` (default `"rnaRA"`) specifies their order.

8. *Line Wrapping.* In the output index file, the merged list of page numbers can be wrapped in multiple lines, if it is longer than `line_max`. The newly wrapped line is indented by `indent_space` whose length is `indent_length`. This artificial line wrapping does not make any difference in formatting, but does provide increased readability for the pre-formatted final index. This feature may seem somewhat trivial at first glance, but if no formatters are involved whatsoever, the readability of the verbatim output index becomes important immediately.

3.3 Sorting Entries

Entries in the raw index file are sorted by comparing index keys as primary and page numbers as secondary. Index keys are sorted first; within the same index key, page numbers are sorted numerically. Sort keys and numeric page numbers are used in comparison while the contents of the actual fields and that of the literal page field are entered into the resulting index (cf. the end of Section 3.1).

3.3.1 Sorting Index keys

A complete index key is an aggregate of one or more sort keys plus the same or fewer number of actual keys. The comparison is based on sort keys, but if two aggregates have identical sort fields and page numbers, the actual keys can be used to distinguish their order. Given two aggregates, their primary sort keys are compared first, then their secondary keys are compared — if the two primaries are identical, and then the third, and so forth. We discuss a number of issues related to sort key comparisons below, where the term "index key" refers to a single instance of sort key rather than the aggregate index key.

Index keys can be categorized into the following groups: *strings*, *numbers*, and *symbols*. A string is a pattern whose leading character is a letter in the alphabet. A number is a pattern consists of all digits. A symbol is a pattern lead by a character not in the union of the English alphabet and arabic digits or something lead by a digit but mixed with non-digits. Members of the same group should appear in sequence. Hence there are two issues concerning ordering: one deals with entries within a group; the other is the global precedence among the three groups in question.

There are no set rules as to which group should have precedence over the other. If we randomly pick books from a shelf and examine their indexes, some would have symbols appear before numbers, which precede strings, while some would have strings appearing first, followed by symbols and numbers. Without loss of generality, our implementation adopts the former precedence rule, but rearranging it is straightforward.

To alphabetize strings, two possible ordering schemes may be considered: *word ordering* and *letter ordering*. Their only difference is whether a blank is treated as an effective letter in the comparison. In word ordering, a blank takes precedence over any letter in the alphabet, whereas in letter ordering, a space is ignored in the comparison and the next non-blank letter is used in its place. This is best illustrated by the following example:

word order	*letter order*
sea lion	seal
seal	sea lion

Comparing strings with mixed upper- and lower-case letters is another interesting issue. The following rule is used in our implementation: letters are first sorted with uppercase and lowercase considered identical; then, within identical words the uppercase letter precedes its lowercase counterpart. Of course, other variations are possible.

In our implementation, numbers are sorted in numeric order. For instance, 9 precedes 10, as in

9 (nine), 123
10 (ten), *see* Derek, Bo

Finally within the group of symbols, characters are compared according to their ASCII ordering in our implementation.

3.3.2 Sorting Page Numbers

There are three basic types of numerals for page numbers: *roman*, *alphabetic*, and *arabic*. A roman numeral is a string composed from the following set of letters:[3]

```
i, v, x, l, c, d, m
```

whose decimal counterpart is

1, 5, 10, 50, 100, 500, 1,000.

A roman page number can also be in all upper case of these letters, but not in mixed upper and lower cases.

An alphabetic page number is any letter in the English alphabet in either upper- or lowercase. Normally lowercase roman numerals have precedence over arabic page numbers, which precede alphabetic page numbers. However, other precedence schemes should be permitted. In our system, the user can specify a different precedence rule by rearranging the order of a five-letter string bound to `page_precedence`. As shown in Table 2, the default is `"rnaRA"`, which denote, in that order, lowercase roman, numeric or arabic, lowercase alphabetic, uppercase roman, and uppercase alphabetic, respectively.

Some document styles organize page numbers according to chapter or section breaks. This means the page count is reset every time a new chapter or section begins. Each page number

[3]Actually, the complete set of roman numerals has four more letters $\bar{v}$, $\bar{x}$, $\bar{c}$, and $\bar{m}$ which correspond to 5,000, 10,000, 100,000, and 1,000,000, respectively. The use of these "barred" letters is unlikely to occur in document processing. Hence they are omitted in the discussion for simplicity.

in this case has two fields: a chapter or section number and the page count, separated by a certain delimiter. For instance, the third page of chapter 12 is 12–3 and the second page of Appendix D is D–2. In some other context, the occurrence of an index key may refer to a section number rather than a page number. This implies the number to be sorted may have more than two fields. For example, Chapter 3, Section 2, Subsection 1 is represented as 3.2.1. In other words, the sorting mechanism must be able to handle composite page numbers. The field separator (i.e. '-' or '.') and the maximum number of levels allowed must be known to the index processor, perhaps through style specification.

In our implementation, page numbers are all assumed to be in the composite form. The lexical analyzer decomposes and transforms each of them into an array of individual numbers in corresponding numeric values. Depending on its type, each of these individual numbers is offset by a constant so that sorting will be based on straight numeric comparisons. This offset is determined by the precedence and the range of that particular numeral type.

3.4 Creating Output Index Entries

Once all input entries are sorted, the output index file can be created. First the attribute bound to `preamble` is placed into the output file. Then goes the string

`setpage_prefix` *N* `setpage_suffix`,

provided *N* is the starting page number and such a setting is requested. Next each entry in the sorted list is processed in order. Finally the attribute bound to `postamble` is appended at the end of the output file.

To place appropriate specifiers for the current entry, it is necessary to look behind at the previous entry. The current entry is discarded if it is a duplicate of the previous one. If the current key turns out to be the beginning of a new group, the extra vertical space bound to `group_skip` will be inserted. In addition, if the new group contains strings beginning with a letter different from that of the previous group, the attributes bound to `lethead_`'s will be used. Suppose the new letter in uppercase is *X*, if `lethead_flag` is positive, then

`lethead_prefix` *X* `lethead_suffix`

will be inserted. Similarly a lowercase letter will be inserted if the flag is negative.

For a brand new entry (i.e. the current and the previous entries share no common prefix), the attribute of the corresponding level specifier (`item_`*i*'s in our example) is entered into the output buffer, along with the actual key, the key/page delimiter (`delim_`*i*'s), and then the literal page number, if any. If an entry has a non-nil encapsulator string, the three `encap_`'s are used to enclose the page number.

If the current index key shares a common prefix with its predecessor, the attribute corresponding to the current subindexing level is attached in front of that particular level's actual key. If the current entry has an aggregate index key identical to the previous one, the current page number is concatenated to the end of a page list with `delim_n` attached in front of it. This page list is wrapped around if it becomes too long (see Section 3.2, item 8). When a new entry is found, the page list is flushed into the output and a new list is created.

Forming an implicit page range is relatively straightforward. First off, the page number of a new entry is always recorded and assumed to be the range opening number. The current page number is in range if it is the immediate successor of that of the previous entry and is more than two pages apart from the opening number. An in-range page is dropped unless it is the closing number, in which case the opening and closing numbers are inserted into the page list with the separator `delim_r` in between.

A somewhat subtle issue arises with regard to explicit page range formation. Consider the following list of entries

```
\indexentry{alpha|(}{1}
\indexentry{alpha|bold}{3}
\indexentry{alpha|)}{7}
```

which depicts an explicit range between pages 1 and 7. The problem is with the second entry which is in this explicit range but with a page encapsulator inconsistent with that of the range opener. Facing this inconsistency, one can simply ignore this in-range entry and issue an error message in the output transcript. This probably makes sense because normally entries like

```
\indexentry{beta|}{3}
\indexentry{beta|bold}{3}
```

are considered an anomaly. But what if a generic concept is covered between pages 1 and 7 while a primary definition occurs at page 3 and is to be set in boldface? Our index processor is more tolerant. Instead of ignoring the entry, it extracts the entry in question out of the range so that the result becomes

```
\item alpha, \bold{3}, 1-7.
```

Meanwhile, a warning message will go into the transcript cautioning the author about this anomaly.

3.5 Miscellaneous

Greater flexibility can be achieved by providing the user with options to choose between *white space verbatim* and *white space compression*, between *word ordering* and *letter ordering*, and among various group precedence schemes. This can all be implemented as extra parameters in the style specification or as command line switches.

An important feature of our index processor is an option to set the starting page number for separate formatting purposes. In addition to any numeric page numbers, it also accepts three special non-numeric values: *any*, *odd*, and *even*. In any of the three special cases, the starting page number is retrieved from the transcript generated as a result of formatting the document source. If the user specifies *any*, the starting page number of the index is the last page of the source plus 1. In the other two cases (*odd* or *even*), it is adjusted accordingly.

For separate formatting to be successful, new attributes must be assigned to `preamble` and `postamble` to incorporate a global context consistent with the main document (cf. Section 2.4). In LaTeX, each document must begin with a command that tells the system what style the document is in. What we need in this case is simply append the same style command used in the main document source in front of the default index preamble given in Table 2. Then, with an appropriate starting page number setting and this new style specification, the file generated by the index processor is all set for separate formatting.

Error handling and warning messages are of great importance. Except sorting, each of the other passes in index processing (style scanning, input index scanning, and output index creation) may produce errors. To generate informative error/warning messages the premise in a batch-oriented approach is keeping track of line numbers in the various files involved. There are basically two input files (the raw index and the style specification) and one output file (the sorted index). Due to the multi-pass nature, some errors won't be caught until the last pass. Warning messages about certain line numbers in the output file make no sense to the user because as errors the processor may not be able to generate anything meaningful for them. As a remedy, input line numbers in the raw index can be saved along with keywords and page numbers by the scanner so that errors caught after sorting may be related to their original occurrences in the input file.

In a WYSIWYG environment, it is possible to do this more interactively. That is, the erroneous entry can be displayed in a dialogue box with a request for correction; or the editor can position itself to the error spot in question and the user can make changes on-line.

4 Placing Index Commands

In this section we introduce a simple framework for placing index commands in a document. It assumes an interactive editor is available with the following functionality:

- *String Search.* This refers to the positioning of cursor to the specified pattern. Regular expression search is a plus, but not essential.
- *Query-Insert.* This refers to displaying a menu of options and upon user's selection, the insertion of a specified key, together with other constant strings (e.g. the index command and its argument delimiters).

We have an implementation of this framework built on top of GNU Emacs [20] as part of an interactive environment for composing TEX-based documents [11]. We believe the underlying model applies not just to conventional text editors but to WYSIWYG systems as well.

4.1 Basic Framework

The basic framework is very simple. 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. In our example, suppose both *pattern* and *key* are `alpha`, then the inserted string after an instance of `alpha` in the document will be `\index{alpha}`. This will be a visible insertion in a source-based situation and will be an invisible insertion for a WYSIWYG system (or a visible one for its shadow pages).

Before the actual insertion is made, it is desirable to make a confirmation request that allows a menu of options to be presented of which *confirm* and *ignore* are the most obvious ones. Thus for each instance of the pattern found, the user can decide if it is to be indexed.

Representing patterns as regular expressions gives significantly more power to this query-insert operation. The same key can represent a complicated string of a basic pattern, its capitalized form, its acronym, and other abbreviations. For instance, the following patterns may all be indexed by the key UCB,

```
University of California, Berkeley
Berkeley
berkeley
UCB
```

As a special case of this <*key*, *pattern*> setup, one can take words in the neighborhood of current cursor position as the implicit value for both the key and the pattern. Some editors allow the use of special characters to delimit word boundaries. This can be used in searching to cut down on the number of "false drops". For example, one can position the cursor after the desired pattern and with one editor command (typically in two or three key strokes), an index entry will be inserted with the preceding word (or words) as the implicit key. The advantage of this facility is that there is no need to do the typing as far as the key-pattern pair is concerned. The same idea also applies to a region of text, which is a piece of continuous text in the document. In Emacs this is everything between a marker and the current cursor position. More generally, the implicit operand can be the *current selection*, in which case the bounding positions of the selected text are not necessarily the insertion point.

We also have a special command to index every author name which appears in the bibliography or references section of a document. This involves skipping citation entries without an author field and for each author name found, issuing a query-insert prompt similar to the normal case. Instead of entering a name directly as the index command argument, it has to display it in the form of last name followed by first and middle names for confirmation, as in

```
Confirm: Knuth, Donald E.
```

This is because we want last names to be the primary sort keys and the name separation heuristic we derived does not always work for people with multi-word last names. The confirmation prompt allows the user to correct it before automatic insertion takes place.

4.2 Key-Pattern List

A collection of these <*key*, *pattern*> pairs can be compiled in a list. A global function can then be invoked to process each pair for the entire document or parts of it. This list can be created off-line by the user, or automatically in an incremental fashion as the user confirms new index insertions in an interactive session. The pattern matching mechanism must be able to recognize and skip instances already indexed so that unnecessary repetitions are avoided. In our system, this key-pattern list is per document. If a document includes multiple files, the global function will process each of them according to the preorder traversal of the file inclusion tree.

4.3 Indexing Menu

For each instance of the pattern found, a menu of options such as the following may be presented.

- *Confirm.*
- *Ignore.*
- *Key-Pattern List.* Add the current <*key*, *pattern*> pair to the list associated with the current document.
- *Index Level.* Prompt the user for an index prefix. The `level` operator (‘`!`’), if not given at the end of the specified string, should be inserted automatically between the prefix and the current key.
- *Actual Field.* Prompt the user for the actual field corresponding to the current (sort) key. The `actual` operator (‘`@`’) should be automatically inserted in between.
- *Page Encapsulator.* Prompt the user for the page number encapsulator. The `encap` operator (‘`|`’), if not given, should be attached in front of the specified string. Encapsulators corresponding to popular fonts such as bold, italic, and slanted, or to cross references like *see* and *see also* can be implemented as a submenu to yield even greater convenience.

4.4 Extended Framework

A typical scenario for placing index commands under the extended framework is as follows. 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 (i.e. a combination of `level`, `actual`, and `encap`). In the latter mode, each key-pattern pair in the global list is processed in a way identical to that of the former mode. In essence the former is just a special case of the latter. Also, the menu option to enter the current key-pattern pair into the global list makes it possible to converge the two modes.

5 Direct Manipulation and Beyond

Although we have not implemented our ideas under the direct manipulation paradigm, a number of WYSIWYG systems do support an indexing facility close to our specification. We discuss three of them briefly to indicate the applicability of the principles depicted in the previous sections. Also we take one step further and examine a more elaborate on-line indexing facility made possible by the electronic media.

5.1 WYSIWYG Indexing

In Xerox PARC's CEDAR environment [21] the TIOGA editor supports an auxiliary application called *IndexTool* [8] that would automatically prepare multiple multi-level indexes (general index, author index, etc.) with cross references (*see* and *see also*) and with substitution phrases (index the phrase "data structures" under "data structure" to handle these automatically). The *IndexTool* would take a selection and create an index entry attached to the document over the selection range. Index entries could be edited in a separate tool to permit creating the cross references and substitution text. TIOGA also has a regular expression search capability via the *EditTool*. The *EditTool* permits a wide range of search and replace operations.

In FRAME MAKER [3], index tags can be placed and edited using a combination of a *Markers* tool and a *Search* tool. The *Markers* tool allows one to specify invisible tags such as subjects, comments, and of course, index entries. For each marker an associated annotation can be specified. In the indexing case, this will be the key to appear in the final index. Page encapsulations discussed previously can also be specified in the *Markers* window. This includes explicit page range, fonts, and an option to disable page numbers so that cross references like *see* can be realized. The *Search* tool can be used to locate the desired pattern in plain or regular expressions. An invisible character `\m` can be specified in the *Search* tool to identify each occurrence of index markers in the document. Whenever a marker is found, the corresponding annotation will be displayed in the *Markers* window. The annotated text can incorporate special symbols to yield multi-level indexing and actual field substitution similar to the ones described in Section 3. A processor called *fmBook* can then be executed off-line to collect index markers, sort them, and finally generate a formatted index whose style is customizable via system supplied property sheets.

In the Macintosh version of MICROSOFT WORD 3.0 [4], an index command is designated by the "index code" `.i.` The text between such a code and an "end-of-entry code" such as a semicolon is regarded as the index key. A colon (`:`) in the entry text acts as the index level operator. The output appearance can be refined by using variants of the index code. For instance, `.iB.` and `.iI.` set the page number in boldface and italic fonts, respectively, `.i(.` and `.i).` create an explicit page range, etc. Index entries need to be "compiled" into the actual index which appears at the end of the document. But before that takes place, the system must be notified that these index codes are *hidden text*. An option in the preference sheet can be set to display hidden text embedded in the document. But hidden text must be "hidden" when index entries are being compiled; otherwise page number computation will be incorrect. There are no special tools for entering index codes. The *find* and *change* commands in the *Search Menu* do not support regular expressions. There are no query-insert mode in the search mechanism. Although abbreviations can be registered as glossary entries, a great many keystrokes are still required in placing a single index entry.

5.2 Dynamic Indexing

The hints we have been giving with regard to implementing an indexing facility under direct manipulation and the three real systems discussed previously are all operating in a multi-pass fashion much the same as one that works in a language-based system. However, based on its interactive nature, a WYSIWYG document editor can probably come up with an indexing subsystem that follows the same central ideas, but with an user interface more closely related to the direct manipulation paradigm.

The "directness" may be observed in the following scenario. The author specifies input/output styles by selecting options available in system supplied property sheets. For each key selected in the document, its corresponding entry in the index is immediately displayed, along with entries already entered. Sorting is done as the entry is generated for the output. Consequently, the internal structure for these entries can no longer be an array which is suitable for fast sorting algorithms such as quick sort. Instead, a balanced tree may be the preferred structure. Insertion and deletion reorganize the tree automatically and a certain traversal of the tree yields the correct ordering. A symbolic link between a page number in the index entry and its corresponding actual page is required so that the index doesn't have to be regenerated when the document is reformatted. In other words, when a page number changes due to reformatting, all instances of that page in the index change automatically.

One possible extension to this model is the ability to point at an entry in the index and have the corresponding page located automatically with the keyword highlighted. Thus indexing becomes a dynamic behavior from the reader's point of view. This typifies the power of "dynamics" which an electronic document environment is able to offer that does not exist in traditional printed static material. In addition to dynamic indexing, one can do such hypertext operations as navigation, filtering, summarizing, etc. [22] effectively based on markup tags and embedded annotations.

6 Evaluation

6.1 Index Placing Subsystem

An important aspect of our system is the framework for placing index commands in the document. This is a task that has been performed traditionally in an ad hoc fashion. It is not clear how things are done in this area under the UNIX TROFF environment. Reference [9] does not indicate any way to assist the author in this process.

Entering index codes in MICROSOFT WORD 3.0 is awkward because its search mechanism lacks full regular expression support and query mode is not available whatsoever. Just to mark one piece of text as an index entry, an ordinary session requires 8 mouse clicks and 4 keystrokes. Even with accelerated keyboard abbreviations, it still takes 2 mouse clicks and 8 keystrokes to mark a single index entry. This is under the situation where the index pattern has been located and is identical to the index key. Obviously locating the pattern may involve extra mouse clicks and keystrokes. Moreover, if the index key is different from the pattern, more keystrokes would be necessary to enter the text for the index key. As a reminder, this happens to the marking of each and every instance of index entries. No global scheme is possible in MICROSOFT WORD 3.0.

The situation in FRAME MAKER is somewhat better because of its more powerful keyboard macro registration capability. In FRAME MAKER, a specific combination of tools can be used to enter index tags at desired places. Operations can be recorded as keyboard macros so that repetitions can be done by a single keystroke (the invocation of the keyboard macro). The problem is that a new keyboard macro has to be defined for each key-pattern pair. Furthermore, it lacks the systematic global scheme available in our extended framework (see Sections 4.4) to make the whole process efficient.

In contrast, using our proposed mechanism it takes only 1 to 3 keystrokes to mark a piece of text as an index entry. We have a global scheme for marking index entries in the entire document which may span over multiple files. In the single key-pattern case, the same key can be inserted at a variety of places described by a single regular expression. Patterns already indexed are skipped automatically. A number of options are available upon each occurrence of the pattern. Thus marking each instance takes just one keystroke to confirm and it works uniformly and continuously throughout the entire document. This can be expanded to a scheme of multiple key-pattern pairs which works iteratively on a list of different index entries. Clearly under our system, a significant amount of time is saved not just in typing *per se*, but in the user's mental reaction time due to the side effect of unautomated discrete repetitions as in MICROSOFT WORD 3.0 and FRAME MAKER.

In the production of an index for the book [10], our Emacs Lisp implementation of the index placing subsystem has proved very useful and effective. The provision for multi-pair <*key*, *pattern*> query-insert has been the most time-saving operation according to our experience. With minor modifications, the same facility would work on placing index commands for other formatting languages like TROFF and SCRIBE. Adapting it to WYSIWYG environments is more involved, but given proper editor programming power, the basic principles discussed in Section 4 should apply without much difficulty.

6.2 Index Processor

The other major portion of our work is a complete index processor implementation with all the features mentioned in Section 3. The resulting processor is a single C program called *MakeIndex*. Actually, *MakeIndex* had several predecessors all written as UNIX shell scripts with embedded `sed` [17] and `awk` [7] code. We quickly became unsatisfied with them for various reasons. One of the concerns had to do with efficiency. Interpreted languages such as `sed` and `awk` are satisfactory for rapid prototyping, but they exact certain penalities at run time. In our experience, *MakeIndex* was able to process a book index of 3,300 entries in less than 50 seconds of user time plus an extra 5% of system time on a client node of SUN 3/50 (MC68020 at 12.5 MHz). This is at least an order of magnitude faster than using its most recent `sed`/`awk` predecessor which has only half the features of *MakeIndex*.

The efficiency issue may evaporate if one argues that normally an index processor is not used until the document is at its final stage. As far as indexing is concerned, most of the time will be spent on placing index commands in the document source. There will be only few invocations to process raw index entries and therefore a slower processor probably does not matter. However, we believe a speedup of over an order of magnitude is something not to be overlooked.

Perhaps more importantly, we switched to a C implementation for its data structuring and dynamic storage allocation capabilities. The general purpose nature has been our design goal from the very beginning. Given all the features described in Section 3, it would be very difficult, if not impossible, to implement them all using `awk`. For instance, in `awk` a dynamic linked list of index entries is impossible and the style handling mechanism with a comprehensive error processing facility such as that demonstrated in Sections 3.1 and 3.2 is very difficult to realize.

Our approach is the direct opposite of that taken by `make.index`, a host of indexing tools [9] built for TROFF. As part of the UNIX TROFF document preparation environment, these tools are canonical examples of the pipelining model. They are a collection of small `awk` programs working together in a very long pipeline. Their claim is that by breaking the system down into small pieces, it is easier for the user to adapt the package to various situations not possible to envision in advance. Their approach, therefore, seems to follow the "do it yourself" metaphor whereby you get a toolkit and assemble the final product yourself. The basic package covers only those features used by its authors. A third party user has to modify their `awk` code if something different is desired.

The design of *MakeIndex* is quite different. Our intention is to build a complete system by carefully analyzing the tasks involved in index processing. Parameters are derived to form a table-driven style handling facility. Thus formatter and format independence is achieved by simple style specification. All the underlying data structures and processing mechanisms are transparent. Yet it is robust enough to handle different precedence schemes, ordering rules, etc. To use the system, the user needs not deal with the processing details. If the default is inappropriate, the only adaptation required is a straightforward table specification for the style facility.

For instance, by assigning the output index header `index.head` defined in `make.index` to our `preamble` and other related commands to `item_`*i*'s, it is very easy to produce an alphabetized index in TROFF format from a raw index generated by TROFF. The same technique applies to other formatting systems. SCRIBE [5], for example, has an indexing subsystem which supports a subset of the functionality described in Section 3. By adapting its raw index file format as input style and its output appearance commands as output style, its indexing capability can be readily expanded using *MakeIndex*. In both cases, there is no need to modify the original code used to generate the raw index (Step II), nor is it necessary to modify *MakeIndex* itself.

In essence, the ease of adaptation is not so important as the design specification itself. If for some reason *MakeIndex* cannot be directly applied to systems like certain direct manipulation environments, the issues raised in Section 3 should still be of value to other implementations.

7 Conclusions

Index preparation is a time-consuming task. Relying on conventional index cards and hand sorting does not make sense in the age of electronic publishing. We started to implement an index processing subsystem for LaTeX, but it has since evolved in several directions.

In one we have made the index processor extremely flexible with respect to input and output formats. This means the processor we implemented is independent of a specific typesetting system and independent of the format being used. For any document preparation system adopting the multi-pass approach, its index or glossary can be processed without modifying the code. All that is needed is a different style specification.

In a second direction, we have built an editing interface which, among many other things, provides the author with substantial help in placing index commands in the document source. The underlying framework is very simple to implement and has proven to be very useful.

In a third direction, we have turned our focus to indexing under the direct manipulation paradigm. We have found that the same specification derived for a language-based environment will suffice in a WYSIWYG document editor. The indexing facilities in TIOGA, FRAME MAKER, and MICROSOFT WORD 3.0 serve as our indirect proofs. Last but not least, an electronic document environment offers more "dynamics" than the print medium and certainly deserves further investigation.

What is not covered in this report is how to arrive at the list of index keys in the first place. Typically an index key represents something of conceptual importance. Automatic derivation of concepts from a document requires "artificial intelligence". Because an index is a comparatively simple object, it is not difficult to envision how this automatic derivation of index keys could be done. The problem is that the amount of knowledge which must be built into the system and the work required exceeds the benefits to be achieved.

8 Acknowledgements

We would like to thank Leslie Lamport for his collaboration in the index processor design. The "sea lion/seal" and "9 vs. 10 (Bo Derek)" examples in Section 3.2 are also due to him. We are grateful to Richard Beach for his description of TIOGA's index processing facility and to Ravi Sethi for the information on TROFF's index processing tools.

References

[1] *8010 STAR Information System Reference Library, Release 4.2.* Xerox Office Systems, El Segundo, California, 1984.

[2] *Addison-Wesley Guide for Authors.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1982.

[3] *Frame Maker Reference Manual, Version 1.0.* Frame Technology Corporation, San Jose, California, February 1987.

[4] *Reference to Microsoft Word, Word Processing Program for the Apple Macintosh, Version 3.0.* Microsoft Corporation, Seattle, Washington, January 1987.

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

[6] *The Seybold Report on Publishing Systems.* Seybold Publications Inc., Media, Pennsylvania. Published 22 times a year.

[7] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *AWK: A Pattern Scanning and Processing Language (User's Manual).* Computer Science Technical Report No. 118, AT&T Bell Laboratories, Murray Hill, New Jersey, June 1985.

[8] Richard J. Beach. Personal communication.

[9] Jon J. Bentley and Brian W. Kernighan. *Tools for Printing Indexes.* Computer Science Technical Report No. 128, AT&T Bell Laboratories, Murray Hill, New Jersey, October 1986.

[10] David H. Brandin and Michael A. Harrison. *The Technology War.* John Wiley and Sons, Inc., New York, New York, 1987.

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

[12] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. In D. A. Norman and S. W. Draper, editors, *User-Centered System Design*, pages 87–124 (Chapter 5), Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey, 1986.

[13] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

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

[15] Donald E. Knuth. *TEX: The Program.* Volume B of *Computers & Typesetting*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

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

[17] Lee E. McMahon. *SED – A Non-interactive Text Editor.* Computer Science Technical Report, AT&T Bell Laboratories, Murray Hill, August 1978. Also available in UNIX User's Manual.

[18] Joseph F. Ossanna. *Nroff/Troff User's Manual.* Computer Science Technical Report No. 54, AT&T Bell Laboratories, Murray Hill, New Jersey, October 1976. Also available in UNIX User's Manual.

[19] Ben Shneiderman. Direct manipulation: a step beyond programming languages. *IEEE Computer*, 16(8):57–69, August 1983.

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

[21] Warren Teitelman. A tour through Cedar. *IEEE Software*, 1(2):44–73, April 1984.

[22] Nicole Yankelovich, Morman Meyrowitz, and Andries van Dam. Reading and writing the electronic book. *IEEE Computer*, 18(10):15–30, October 1985.