Index Preparation and Processing*

Pehong Chen†

Michael A. Harrison

Computer Science Division

University of California

Berkeley, CA 94720

Abstract

Index preparation is a tedious and time-consuming task. This paper indicates how the indexing process can be automated in a way that 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 develop a framework for placing index commands in the document. In addition, the design of a general purpose index processor that transforms a raw index into an alphabetized version is described. The resulting system has proved very useful and effective in producing indexes for several books, technical reports, and manuals. A comparison of our system with indexing facilities available from a variety of other document preparation environments is given.

Keywords: index placement, index processing, source-language model, direct manipulation.

1 Introduction

Although there has been a great deal of activity in electronic publishing [1], there are still aspects of document composition that have not been fully automated. One of the most time-consuming concerns is the preparation of an index. In ordinary books, an index allows a reader

*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 and Naval Warfare Systems Command, under Contract No. N00039-84-C-0089.

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

to access essential information easily. A poor index with many omissions or poorly chosen concepts detracts from other aspects of the book. For highly complex technical material that may include computer programs, different kinds of indices may reference even the identifiers of a programming language. A good example of an elaborate indexing scheme can be found in Knuth's *TEX: The Program* [2] and his WEB system [3] in general. For computer programs like these, completeness is essential and the accuracy of traditional hand methods will not suffice for software engineering applications.

Standard authors' guides, such as Reference [4], 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. Computers offer an opportunity to significantly reduce the amount of labor invested in this process while noticeably improving the quality of the resulting index.

This paper indicates how the indexing process can be automated in a way that is largely independent of a specific typesetting system and independent of the format being used. Specifically, we develop a framework for placing index commands in the document. In addition, the design of a general purpose index processor that transforms a raw index into an alphabetized version is described. These concepts have been implemented as part of an extensive authoring environment [5]. This environment 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 some books [6,7] and a number of technical reports and manuals.

Indexing issues under both *source-language* and *direct-manipulation* [8,9] paradigms are considered. In a source-language based system, the user specifies the document with interspersed commands, which is then passed to a formatter, and the output is obtained. The source-language usually provides some abstraction and control mechanisms such as procedures, macros, conditionals, variables, etc. in much the same way as a high-level programming language does. In a direct-manipulation environment, the user manipulates the document output appearance directly by invoking built-in operators available through menus and buttons. These systems are highly interactive; the result of invoking an operation is observed instantaneously, thereby creating an illusion that the user is "directly" manipulating the underlying object.

The document attributes in direct-manipulation systems are usually specified by a declarative language encapsulated as form-based property sheets. These property sheets correspond to textual markup tags that can be imported to or exported from the direct-manipulation system for document interchange purposes. While a source representation is maintained explicitly in the source-language model, the notion of a document semantics specification language is somewhat implicit in direct-manipulation systems.

Some people have called direct-manipulation systems WYSIWYG (*what-you-see-is-what-you-get*). The two concepts are not equivalent, however. WYSIWYG refers to the correspondence between what is presented on a video display and what can be generated on some other device. In the context of electronic publishing, WYSIWYG means that there is a very close relationship in terms of document appearance between a screen representation and the final hardcopy. Direct manipulation is a more general concept that models user interfaces. A direct-manipulation document preparation system may not have a WYSIWYG relationship between its display representation and the final hardcopy. Conversely, a batch-oriented, source-language based formatter may be coupled with a previewer that is WYSIWYG.

This paper presents some general indexing problems and our solutions in a top-down fashion. First we discuss the desirable features of an index processor and then some design and implementation considerations for such a processor. Our goal is to arrive at general purpose solutions. Next, a framework is introduced under which an author enters index commands or tags with much reduced overhead. The examples shown are in LaTeX [10], a high-level document preparation language based on TeX [11]. The model, however, is not restricted to any particular formatting language, nor to the source-language paradigm. We also examine some unique indexing issues in electronic document development environments that do not seem to find appropriate counterparts in traditional printed material. Finally our indexing facilities are evaluated against those available in other formatting systems such as *Scribe* [12], *troff* [13], and some direct-manipulation environments.

2 Basic Tasks

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.

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. Here, all index keys are sorted alphabetically. Page numbers under the same key are merged and successive numbers may be 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 here. This procedure is a highly sequential, for the input to one step depends upon the result from the previous one.

Figure 2 exemplifies a stepwise development of the process. In LaTeX and TeX, all commands begin with a backslash (\). 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, it is unclear 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 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.

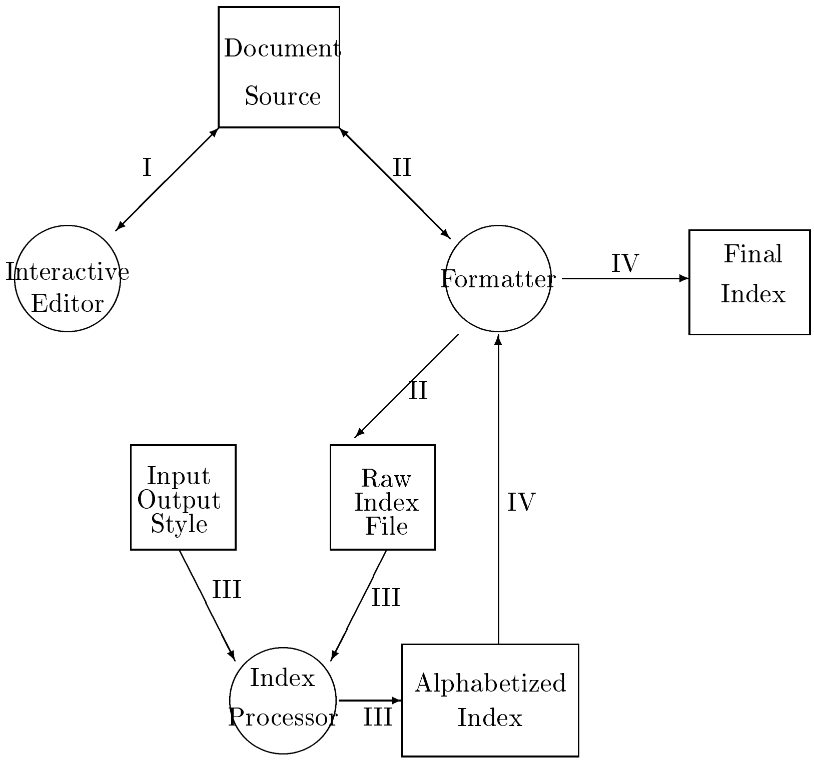

Figure 1: Circles in the picture represent processors, squares are documents or auxiliary files. In Step I, the author uses an editor to place index commands in the document. In Step II, a raw index is generated as a by-product of formatting. In Step III, this raw index together with some optional style 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.

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: *The stepwise development of index processing*. This example is specified in LaTeX. (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. Page numbers are included 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.

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

2.1 Placing Index Commands

Step I deals with placing index commands in the document. In a source-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 that, 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 shadow document approach is widely adopted by direct-manipulation document development or desktop publishing systems such as Xerox *Star* [14], *FrameMaker* [15], *MicroSoft Word* [16], and *Ventura Publisher* [17].

The primary issue in step I for both paradigms is whether or not a systematic mechanism can be derived for the entering of index commands or tags. Details of a general model that accomplishes this task are given below.

2.2 Generating the Raw Index

Step II concerns 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 source-language or direct-manipulation based, 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 source-language based system, like formatting itself, is by and large a "batch job". In a direct-manipulation 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

Processing raw index entries raises several issues. Some high-level issues are described below with references to the example given in Figure 2; how these tasks can be realized is detailed in the next section.

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.

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). If citations on successive pages are logically distinct, good indexing practice suggests that they should not be represented by a range. Our system allows user control of this practice.

Subindexing. Multi-level indexing is supported. Here, 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).

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 `{}`. In general, it is impossible to know all the patterns that the index processor should 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.

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.

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. The page number, however, 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, work similarly.

Input/Output Style. In order to be formatter- and format-independent, the index processor must be able to handle a variety of formats. There are two reasons for considering this independence issue in the input side: Raw index files generated by systems other than LaTeX may not comply to the default format, and 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 of the input format and the output style.

2.4 Index Formatting

Two key issues in this last step are support for multiple styles and 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 of the entire document. Indexing is typically the last step of document preparation, and is attempted only when the entire document is finalized. It is desirable to be able to generate the index without reformatting the entire document. To be able to format the index separately, the global context must be known, which is made possible by the extensible style facility in our design. 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. Some styles require that the index start on an even or odd page number.

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 that escapes `quote`
`page_compositor`	*string*	`"-"`	composite page delimiter

Table 1: Input format parameters.

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 above — permutation, page number merging, subindexing, style handling, and other special effects. In order to achieve format and formatter independence, the index processor must be able to accept raw index terms designated by different keywords and delimiters. Likewise, it must be able to generate output in a specific style so that the result can be processed by the corresponding formatter. The intended tasks can be performed in multiple passes: First the input format file and output style file 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. The remainder of this section discusses the essential attributes for input formats and output styles and points out relevant issues for sorting and generating the entries.

3.1 Input Format

Table 1 is a summary of the input format that consists of a list of <*specifier*, *attribute*> pairs. These attributes are the essential tokens and delimiters needed in scanning the input index file. Default string constants are enclosed in double quotes (`"..."`) and character constants are in single quotes (`'x'`). The user can override the default value by specifying a specifier and a new attribute in the format file or property sheet. The attribute of `keyword` is self-explanatory; `arg_open` and `arg_close` denote the argument opening and closing delimiters, respectively. The meanings of special operators such as `level`, `actual`, and `encap` are described above.

The two range delimiters `range_open` and `range_close` are used 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. This *implicit* range formation can be turned off if an indexed term represents logically distinct concepts in different pages. When the implicit range is disabled, *explicit* page ranges can be enforced by using the two range delimiters `range_open` and `range_close`. Therefore, it is 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 is used to escape 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; `quote` and `escape` must be distinct.

A page number can be a composite of one or more fields separated by the 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, simplifying the sorting of page numbers.

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.

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. These parameters can be further divided into the following groups:

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

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.

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.

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 to be different from those with page numbers.

Table 2 depicts a system that 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 $(1 \leq j \leq n-1)$.

Page Delimiters. Each level has a key/page delimiter that defines what is to be inserted between a key and its first page number. The inter-page delimiter is specified by `delim_n`, while the range designator is given by `delim_r`.

Page Encapsulator. The attributes of `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
```

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.

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 is important.

3.3 Sorting Entries

Entries in the raw index file are sorted primarily on their index keys and secondarily on their page numbers. Index keys are sorted first; within the same index key, page numbers are sorted numerically. Sort keys and numeric page numbers are used in the comparison, while actual keys and literal page fields are entered into the resulting index. In our design, a complete index key is an aggregate of one or more sort keys plus the same or a smaller 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.

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 consisting of all digits. A symbol is a pattern beginning with a character not in the union of the English alphabet and arabic digits or starting with 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. Details of sorting index keys can be found in Reference [18].

There are three basic types of numerals for page numbers: *roman*, *alphabetic*, and *arabic*. The sorting of arbitrary combinations of these three types of numerals (e.g., 112, iv, II-12, A.1.3, etc.) must be based on their numeric values and relative precedence. The attribute of `page_precedence` in Table 2, for instance, specifies the precedence. Again, details of sorting page numbers can be found in Reference [18].

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, followed by 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. The algorithm for generating each entry with appropriate formatter-specific commands or delimiters (i.e. `item_`i's, `delim_`i's, etc.) is described in Reference [18].

4 Placing Index Commands

This section discusses a simple framework for placing index commands in a document. It assumes an interactive editor is available with *string search*, which positions the cursor to a specified pattern, and *query-insert*, which displays a menu of options and, upon the user's selection, inserts a specified key, together with other constant strings (e.g., the index command and its argument delimiters).

We implemented this framework on top of GNU *Emacs* [19] as part of an interactive environment for composing TeX-based documents [5]. The underlying model applies not just to conventional text editors, but to direct-manipulation 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 is `\index{alpha}`. This insertion will be visible in a source-language based system and will be invisible for a direct-manipulation system (or visible as hidden text for its shadow pages).

Before the actual insertion is made, it is desirable to make a confirmation request that presents a menu of options, 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 use 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. These characters can be used in searching to reduce 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 type the key-pattern pair. The same idea also applies to a region of text, which is a piece of continuous text in the document. In *Emacs*, a region 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.

In our system, there is also a special facility to index every author name that appears in the bibliography or references section of a document. This feature 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 term, it is better to display it in the form of last name followed by first and middle names for confirmation, as in

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

This reordering yields last names as primary sort keys. Our name separation heuristic does not always work for multi-word last names. The confirmation prompt allows the user to correct it before insertion.

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 a per document list. If a document includes multiple files, the global function processes 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.

4.4 Extended Framework

A typical scenario for placing index commands under the extended framework is as follows. There are two query-insert modes: one based on a 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, 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, there is a collection of entries that may be represented as a long list for an entire document, or as a short list for a section. Each key-pattern pair in the list is processed as in the former case. Provisions are also available to help construct the list.

With such powerful mechanisms available, there is a tendency to "over index" because it is so easy to be comprehensive. In some cases, "less may be more". Therefore, one should be extremely careful in confirming the insertion of *key* to avoid false drops picked up by the search mechanism.

5 Direct Manipulation

Although our implementation of these facilities was for a source-based system, a number of direct-manipulation systems do support a similar indexing facility. Four systems are described briefly to indicate the applicability of the principles depicted in the previous sections. Also examined is a more elaborate on-line indexing facility made possible using the electronic media.

5.1 Indexing under Direct Manipulation

In Xerox PARC's *Cedar* environment [20], the *Tioga* editor supports an application called *IndexTool* [21] that automatically prepares 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). *IndexTool* takes a selection and creates an index entry attached to the document over the selection range. Index entries can be edited in a separate tool to permit creating the cross references and substitution text. *Tioga* also has a regular expression search capability via *EditTool*, which also permits a wide range of search and replace operations.

In *FrameMaker* 1.0 [15], index tags can be placed and edited using a combination of the *Markers* and the *Search* tools. *Markers* 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, the annotation is the key to appear in the final index. Attributes of page encapsulation discussed previously can also be specified in the *Markers* window. These attributes include explicit page range, fonts, and an option to disable page numbers so that cross references like *see* can be realized. *FrameMaker*'s *Search* can be used to locate the desired pattern in plain or regular expressions. An invisible character `\m` can be specified in *Search* to identify each occurrence of index markers in the document. Whenever a marker is found, the corresponding annotation is 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 above. A processor called *fmBook* can 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 [16], an index command is designated by the "index code" `.i.` The text between `.i.` 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 are "compiled" into the actual index that appears at the end of the document. 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. 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 modes 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.

In *Ventura Publisher* 1.1 [17], a desktop publishing system for documents prepared on any of several word processing programs on the IBM PC, index entries can be placed in the document by invoking a dialogue box at the desired position. An alternative is to use a word processor to enter a markup tag such as

```
<$I<I>alpha<L>[alpha];beta>
```

where `<$I···>` marks ··· as an index term, `<I>alpha<L>` is the actual key in italic, `[···]` designates ··· as the corresponding sort key, and the semicolon is the index level operator. It supports two levels of subindexing as well as *see* and *see also*. Index terms are hidden text; the word `Index` will be displayed by its *Current Selection Indicator* when the cursor is placed at a location where an index term is anchored. The output index style can be specified by changing attributes in a property sheet called `GENERATE INDEX`. The collection of attributes is a proper subset of Table 2. Index processing is executed as a batch job similar to *fmBook* in *FrameMaker*. Searching is unavailable in *Ventura Publisher*, therefore an automated index placing subsystem is not possible.

5.2 Dynamic Indexing and Hypertext

Our proposed implementation of an indexing facility under direct manipulation and the four real systems discussed previously all operate in a multi-pass fashion, much the same as a source-language based system. However, based on its interactive nature, a direct-manipulation document editor can be constructed 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 that is suitable for fast sorting algorithms. Instead, a balanced tree may be the preferred structure. Insertion and deletion reorganize the tree automatically and a traversal of the tree yields the correct ordering. A link between a page number in the index entry and its corresponding actual page is required so that the index needs not 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 a keyword highlighted. Thus indexing becomes a dynamic activity from the reader's point of view. This feature typifies the power of "dynamics" that an electronic document environment is able to offer, which does not exist in traditional static printed material. In addition to dynamic indexing, one can do such hypertext operations as navigation, filtering, summarizing, etc. [22] effectively based on markup tags, embedded annotations, and links among compound objects in the document.

6 Evaluation

6.1 Index Placing Subsystem

An important aspect of our system is the framework for placing index commands in the document. This task has been performed traditionally in an ad hoc fashion. It is unclear how placement is done in the UNIX *troff* environment. Reference [23] does not describe any automated assistance for the author.

Entering index codes in *MicroSoft Word* is awkward because its search mechanism lacks full regular expression support and query-insert mode is unavailable. 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. The premise, however, is that the index pattern has been located and that it is identical to the index key. Locating the pattern may involve extra mouse clicks and keystrokes. Moreover, if the index key and the search pattern are distinct, more keystrokes would be necessary to enter the text for the index key. This situation happens to the marking of each and every instance of index entries. No global scheme is available in *MicroSoft Word*.

Ventura Publisher does not support any searching mechanism; it assumes searching is performed in a front-end word processor. Therefore, systematic index placing in the stand-alone *Ventura Publisher* is difficult.

The situation in *FrameMaker* is somewhat better because of its more powerful keyboard macro registration capability and the support for regular expression search. In *FrameMaker*, 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 to make the whole process efficient.

In our system, it takes only 1 to 3 keystrokes to mark an index entry. Also available is a global scheme for marking index entries in the entire document that 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 that works iteratively on a list of different index entries. In 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 *FrameMaker*, *MicroSoft Word*, and *Ventura Publisher*.

In the production of an index for a book [6], 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 in our experience. With minor modifications, the same facility should work on placing index commands for other formatting languages like *troff* and *Scribe*. Adapting it to direct-manipulation environments is more involved, but given proper editor programming power, the basic principles discussed above should apply.

6.2 Index Processor

The other major portion of our work is a complete index processor implementation. The resulting processor is a single C program called *MakeIndex*. Actually, *MakeIndex* had several predecessors all written as UNIX shell scripts with embedded `sed` [24] and `awk` [25] code. We quickly became unsatisfied with them for various reasons. One of the concerns was efficiency. Interpreted languages such as `sed` and `awk` are satisfactory for rapid prototyping, but they are slow. 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 decision to do a C implementation was made because of C's dynamic storage allocation and data structures. Since we wanted as general a solution as possible, it would be difficult, if not impossible, to implement all the desired features using `sed/awk`. For instance, in `awk` a dynamically linked list of index entries is impossible, and the style handling mechanism with a comprehensive error processing facility is difficult to realize. Originally developed in UNIX C with portability in mind, *MakeIndex* also runs on VAX/VMS, TOPS-20, MS/DOS systems with a few trivial changes. This portability would have been very difficult for an implementation in UNIX shell scripts. Furthermore, *MakeIndex* has more features and is more extensible than tools like *IdxTEX* [26], a LaTeX-specific index processor that runs only on VAX/VMS systems.

Our approach is the direct opposite of that taken by `make.index`, a host of indexing tools [23] 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. The 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 analyzing the tasks involved in index processing. Parameters are derived to form a table-driven style handling facility. 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 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 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*, for example, has an indexing subsystem that supports a subset of the functionality described above. 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. Likewise, *MakeIndex* can be integrated with direct-manipulation document development or publishing systems by binding their descriptive markup tags to the attributes of *MakeIndex*'s input format and output style.

To summarize the differences in approach between *MakeIndex* and `make.index`, let us examine the points of comparison: speed, size of the programs, ease of use, portability, and extensibility. *MakeIndex* is an order of magnitude faster. The `awk` programs of `make.index` are less than 100 lines of code, while *MakeIndex* is large: 6,000 lines of C code and a binary of 70K bytes. *MakeIndex* is a self-contained C program, adapted from our original UNIX implementation, to run on a variety of machines from micros to mainframes. The `make.index` system will run on any UNIX system which has `awk` so it too spans many systems, although it is UNIX-specific while *MakeIndex* is not.

MakeIndex is easy to use. It is monolithic and covers almost every conceivable case likely to occur. We actually examined many of the books in the Addison-Wesley Computer Science Series and found that more than 95% of the cases could be handled by the program. The `make.index` suite, on the other hand, handles the standard cases and all customizations must be done by the user by modifying `awk` code. With *MakeIndex*, user changes will very rarely occur. If they should occur, and we have never had them happen, then the user must change the C source code. The question as to whether it is easier to modify `awk` code or C code is left to the reader. Our answer is given in *MakeIndex*.

7 Acknowledgements

We would like to thank Leslie Lamport for his collaboration in the design of *MakeIndex*. He provided the initial specification and did some serious testing. Charles Karney furnished the patches for *MakeIndex* to run under VMS. Nelson Beebe ported *MakeIndex* to TOPS-20 and MS/DOS. Some useful feedback came from Art Werschulz who tried out both *MakeIndex* and the index placing facility with $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX. We are grateful to Richard Beach for his description of *Tioga*'s index processing facility, to Ravi Sethi for the information on *troff*'s index processing tools, and to Ventura Software, Inc. for giving us documentation on their indexing mechanism. Thanks also go to Ethan Munson and the anonymous referees for their helpful comments on earlier drafts of this paper.

References

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

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

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

[4] *The Chicago Manual of Style*, 13th edition, 1982. University of Chicago Press, Chicago, Illinois.

[5] Pehong Chen and Michael A. Harrison. Integrating noninteractive document processors into an interactive environment. Technical Report 87/349, Computer Science Division, University of California, Berkeley, California, April 1987. Submitted for publication.

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

[7] J. F. Traub, G. W. Wasilkowski, and H. Woźniakowski. *Information-Based Complexity.* Academic Press, New York, New York, 1988.

[8] 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), Hillsdale, New Jersey, 1986. Lawrence Erlbaum Associates, Inc.

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

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

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

[12] Brian K. Reid. *Scribe: A document specification language and its compiler.* PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, October 1980. Available as technical report CMU-CS-81-100.

[13] 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.

[14] *8010 STAR Information System Reference Library, Release 4.2.* Xerox Office Systems, El Segundo, California. 1984.

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

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

[17] *Ventura Publisher — Professional Publishing Program, Reference Guide, version 1.1.* Ventura Software, Inc., Salinas, California. July 1987.

[18] Pehong Chen and Michael A. Harrison. Automating index preparation. Technical Report 87/347, Computer Science Division, University of California, Berkeley, California, March 1987.

[19] Richard M. Stallman. *GNU Emacs Manual, Fifth Edition, Version 18.* Free Software Foundation, Cambridge, Massachusetts, December 1986.

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

[21] Richard J. Beach. Personal communication.

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

[23] Jon J. Bentley and Brian W. Kernighan. Tools for printing indexes. *Electronic Publishing*, 1(1), June 1988. Also available as Computer Science Technical Report No. 128, AT&T Bell Laboratories, Murray Hill, October 1986.

[24] Lee E. McMahon. Sed – a non-interactive text editor. Computer Science Technical Report No. 77, AT&T Bell Laboratories, Murray Hill, August 1978. Also available in UNIX User's Manual.

[25] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[26] Richard L Aurbach. Automated index generation for LaTeX. *TUGBoat*, 8(2):201–209, July 1987.