xindy

Definition of an Indexing Model and its Implementation

Roger Kehr

May 1997

Bericht TI-11/97
Institut für Theoretische Informatik
Fachbereich Informatik
Technische Hochschule Darmstadt

Abstract

This report describes a new indexing model and its implementation. The model is the result of an analysis of existing indexes and an evaluation of several index processors, mainly *MakeIndex* and *International MakeIndex*, for which we have analysed their features and weaknesses in practical applications.

We have identified two orthogonal dimensions in an index. The vertical dimension consisting of index entries and keywords is mostly well understood. The theoretical part of our work mainly contributes to a formalisation of the horizontal dimension of indexing, namely the location references and mechanisms involved in their processing.

Based on our indexing model we have implemented the xindy-system, an index processor that is a nearly full implementation of our model. Additionally, it contains a powerful declarative event-based dispatching scheme to support complex markup strategies.

We describe our model and some of the implementation details that may be of general interest, especially the declarative object-oriented nature of the user interface.

1 Introduction

> Sometimes it is desirable to index words that don't actually appear on the page. . . . For example, Appendix I lists page 1 under 'beauty', even though page 1 only contains the word 'beautiful'. (The author felt that it was important to index 'beauty' because he had already indexed 'truth'.)
>
> Donald E. Knuth, The TEXbook (1984)

Today, the need to index information becomes more and more important, to guide readers in finding the desired information efficiently and effective. For large documents indexes are traditionally one form of service that helps finding the desired information.

Compiling an index is still a tedious work. First of all the indexer has to decide what items should appear in the index. Furthermore it has to be decided to which locations in the document the reader should be guided first, separating important from less important occurrences of items in a document.

Books are often not indexed by the authors themselves. For textbooks it is necessary that the indexer has sufficient knowledge of the problem area to make appropriate indexing decisions. Unfortunately, the index is very often the most neglected part of a book.[1]

Most of nowadays word processors are still weak in supporting the production of high-quality indexes, which might be one reason for the deficiencies found in many textbooks. Our work is an approach to give a model of what an index is and how a good indexing system should operate. Based on this model we present xindy, our implementation of this indexing model.

The rest of this paper is organised as follows: section 2 describes the data flow between a document preparation system and the indexing system and we characterise the involved processing phases. In section 3 we analyse the structure of indexes, thus identifying the components an index consists of and achieving detailed insight into the processing of location references. Section 4 describes the current implementation. We present details of the processes involved in the transformation of an index into its final form and how the internal representation inside the model is tagged with markup information and fed back to the document preparation system. Necessary design decisions in the implementation of the xindy-system are described and lessons learned from the project are reported.

[1] Especially as it can be used to express feelings (as shown by the quotation of D. E. Knuth), as well as humour (see index entry for *kludges* in [17]).

2 Data Flow in the Indexing Process

Current document preparation systems can be roughly divided into two categories: (a) markup-based batch-processing systems such as TEX/LATEX [10, 11] or the `nroff` family of document formatters [12], and (b) direct manipulation word processors typically operating under some windowing system. One of the most widespread used index processor in the first group is *MakeIndex* [2, 3]. The indexing capabilities of the latter group are less powerful than *MakeIndex* and not controllable from outside. Therefore, we have concentrated on indexing functionalities in the first group. Figure 1 describes the typical data flow in this group of systems.

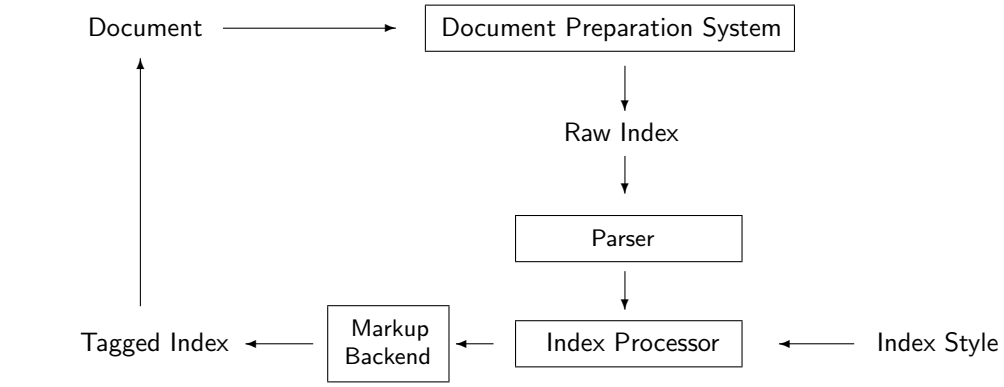

Figure 1: Typical data flow in document preparation systems

The document is processed by the document preparation system that extracts the indexing information, the so-called *raw index*. The raw index comes in a system dependent format containing information about the raw index entries.

The *parser* reads the raw index and transforms it into a representation suitable for the index processor. That means, it must be adopted to the format of the document preparation system and the index processor.

The *index processor* reads two different data streams: (1) the *index style*, describing how the index is to be processed and tagged, and (2) the raw index. Typical processing tasks are: (a) merging and sorting the index entries according to some alphabet, (b) accumulating all location references with the same keyword into an index entry, (c) decomposing and classifying location references, and (d) merging and sorting the location references optionally followed by a range-building phase.

After the index has been processed, its internal representation must be fed back into the document preparation cycle. This is done through the *markup backend* that outputs the internal format tagged with appropriate markups defined in the index

style in a representation that can be used by the document preparation system to typeset the index.

From an abstract point of view the indexing process can be described as a homomorphism, mapping the raw index to the tagged index. Figure 2 illustrates this view.

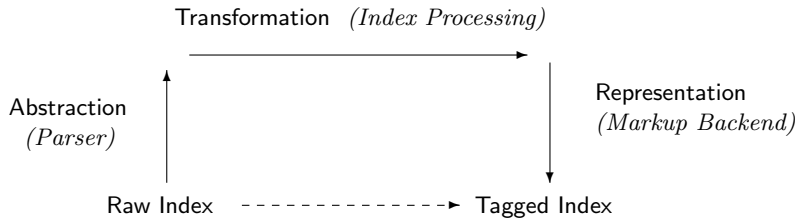

Figure 2: Index processing as a homomorphism

The raw index is mapped into an internal representation through an *abstraction process*. This task is effectively done with a parser. It transforms the raw index in a representation suitable for processing the index. The indexing process is a *transformation* yielding an abstract representation of the index. In our system the result of the transformation is a tree structure. An appropriate output form is achieved through the *representation mapping* done with the markup backend.

The *MakeIndex* system does not cleanly separate the three mappings of this homomorphism. In *MakeIndex* the parser is part of the indexing system. It can be configured to accept a certain range of raw index formats, but essentially the formats are rather restricted because it was designed primarily as an index processor for the TEX-system and its raw index format. Additionally, the markup is in parts directly derived from the raw index. Thus, *MakeIndex* operates on a level that more or less directly maps the raw index to the tagged index. We have removed the parser from the core indexing system and defined a general raw index specification format.

In our specification format a *raw index entry* consists of a keyword and a raw location reference. A *keyword* consists of a list of strings. A *string* is a word over the underlying *document alphabet*. The *raw location reference* is a string.

Additionally, we have made the representation mapping highly controllable by the user. This is a major improvement in the design of index processors.

In this report we mainly concentrate on a description of the transformation mapping and a brief description of the representation mapping. A detailed description of the representation mapping can be found in [9].

3 The Indexing Model

3.1 Definition of an Index

Typically, indexes are collections of ordered keywords each one followed by a list of *location references*. Usually the index is part of the indexed document and the location references represent locations within that document. At some stage in the document preparation process the index is generated from the document information and fed back to the document preparation system.

Figure 3 shows an example index. Despite the fact that such confusing indexes probably do not exist in practice, we recognise the two-dimensional structure of an index.

---

trees
    AVL, 2.3
    natural, A-1, 2.1
Fibonacci queues, *see* priority queues
search
    binary, 11, 11a, 1.3
    sequential, A-2, 1.2
    ordered, 1.2.1, *see also* unordered search
    unordered, 1.2.2
sorting
    quick sort, 37–41
priority queues
    Fibonacci, 3.3

Figure 3: Sample index with section numbers and page numbers

In the *vertical dimension* the index is structured by the keywords, since they actually serve as the key for the reader's search. The keywords form a hierarchy which is often visualised by the level of indentation. This is not always the practice. For example, the Chicago Manual of Style [4] recommends the so-called *flush-and-hang style* or the *run-in indented style* that are visualisations of the appearance of index hierarchies. As opposed to hierarchical indexes there is often need for so-called *flat* indexes.

Along the *horizontal dimension* we identify the list of *location references*. A location reference is an object that serves as a reference to some location within the document. We can observe different types of locations in documents such as page numbers, section numbers, parts of the appendix, cross references, etc. Most indexes contain only references to pages but there are many application areas in which pages are of less concern or don't even exist.

3.2 Index Entries

An index entry consists of a keyword which is a list of strings and a list of location references. Most indexes have a recursive structure since an index entry can possibly be composed of other (sub-)index entries. Thus index entries may form a recursive structure.

A keyword consists of a list of *strings*. Strings are *character sequences* over the underlying *alphabet* that is used in the document preparation system such as ASCII or the ISO Latin family. Strings are denoted using the quote characters "...". An example keyword consisting of two strings might be ("search" "sequential"). We use parenthesis to denote list-based structures. A keyword can be interpreted as the *path* in the tree structure to the corresponding location references. This view raises two basic questions: (a) how do we recognise that two keywords and therefore strings are equal, and (b) how do we order them?

We first discuss the ordering problem. Strings are usually ordered according to the lexicographic order of the sequence of characters. The order of characters in the underlying document preparation system often does not correctly represent the language specific lexicographic order, being a problem for most of the European languages. For instance, in many Roman languages the accented letters such as *á*, *à*, *ã*, *â*, *ä*, etc. are sorted like the corresponding letter without an accent. Sorting based on the encoding of these letters in the ISO Latin-1 alphabet would frequently yield undesired results. Another problem is that in many languages a letter is actually composed of two or more letters of the underlying document alphabet. For example, in Hungarian there are letters such as *Cs*, *Ly*, *Ny*, and Spanish has the letters *Ch*, *Ll*. Furthermore, a keyword might contain formatter dependent markup information that must be ignored when comparing keywords. This problem is discussed in detail in [15] and a scheme called *keyword mappings* is proposed. A *merge key* is generated from the initial key through the *merge mapping*. From the merge key the so-called *sort key* is derived through the *sort mapping*. Keyword mappings have been incorporated into version three of *MakeIndex* named *International MakeIndex* [16] and are implemented using string and regular expression substitutions. With these mapping mechanism an initial string s with its lexicographic order l_s can be mapped onto a string t having a lexicographic order l_t that better represents the order of the language specific alphabet.

The problem of equality is solved with the merge mapping. Two keywords are *equal* iff they are mapped onto the same merge key. The sort key is then used to order the keywords solving the ordering problem.

The problem of *letter groups* has been solved in *International MakeIndex*, too. Letter groups usually appear as separate groups in the tagged index, and thus must be identified correctly. Letter groups are identified using a comparison of a prefix

of a sort key with a list of all letter groups declared in the index style. This model assumes that letter groups are built using a prefix of the sort key. Other models are not supported.

Though keyword mappings can be used to describe (a) language dependent sorting orders, (b) language dependent letter groups, and (c) to appropriately handle the formatting information that might be part of the keywords itself, it is still weak from the users perspective. Complex sorting schemes such as the French sorting rules described in the ISO 14651 standard *International String Ordering* [5] are hard to express in the current model. The *International MakeIndex* model is essentially a *string rewriting system* implemented with string and regular expression substitutions. The most important weakness with such a system is that it operates at the character level and does not work for more abstract objects, which might simplify the specification of sort rules. The ISO standard describes a different method for sorting strings that is based on a table-lookup specification of sorting rules which in some cases simplifies the specification of sorting rules.

We have incorporated the keyword mapping scheme from *International MakeIndex* with minor extensions into xindy. A new scheme is under development that unifies and extends the concepts of the *International MakeIndex* and the ISO standard.

3.3 Location References

The second part of an index entry is the set of location references. Mostly locations pointed to by the index are page numbers. In many application areas this *physical location* of an indexed item is of less interest, since often documents are built according to a more *logical* structure the index must refer to. For example, in Hypertext systems there are no page numbers. We give some examples of these logical structures:

Regular Structures

- Chapter-based pages of the form ([arabic]–[arabic]): *2–8*, *2–12*, *4–7*, *4–9*.
- Texts of Law: **1** §436, §446; **2** §546.
- URLs: `http://www.iti.informatik.th-darmstadt.de/xindy/`.

Variable Structures

- Manual pages of the form ([arabic]{[alpha]}): *11*, *11a*, *17*, *17c*.
  This kind of irregular structures consisting of mandatory and optional parts[2] sometimes occurs in manuals that have new pages inserted between old

[2]Denoted with curly braces {...}.

ones. This preserves the overall structure by inserting, for instance, page *11a* between page *11* and *12*.

- Bible verses: *Genesis 1, 31*; *Exodus 1, 7*; *Leviticus 2, 3*; *Psalm 47*.
- Hierarchical Sections of the form ([arabic]{.[arabic].[...]}): *1.1*, *1.2.1*, *2.1*.

A structural description of location references is done with the concept of *location classes*.

3.4 Location classes

The detailed analysis of the requirements on location references and their formal definition is the most important new aspect on processing indexes. Typically, the document preparation system represents the location references as a simple string. Thus, the index processor needs to parse the location reference string and identify the components of the location reference conforming to some description. Our model is based on the idea that the location structures appearing in a document must be explicitly defined in the index style. A so-called *location class* intensionally describes all possible location references matching this class. The index processor then tries to match each location reference against its known location classes and decomposes its structure, if a match was found.

The description of a location class C is a list of *alphabets* $(\alpha_0, \alpha_1, \ldots, \alpha_n)$. Typical alphabets we encountered are: (a) arabic numbers, (b) roman numerals, (c) letters, (d) words over the document alphabet such as '*Genesis*' or '`http`'. Thus, there are alphabets that are actually sequences of elements of *finite length*, e.g. the letters {'A', ..., 'Z'} or the alphabet {'*Genesis*', '*Exodus*', '*Leviticus*'}. A sequence of finite length can be declared by simply listing all of its elements, thus $\alpha_i = \{l_0, l_1, \ldots, l_k\}$ with l_j being the letters of the alphabet. The sorting order inside an alphabet is simply derived from the order of its letters ($l_i < l_j \Leftrightarrow i < j$). An alphabet consisting of only one letter defines a *separator*, i.e. a string that is used to separate several components of a location reference. Typical examples are dashes, dots, and blanks ('–', '.', '␣').

An example of a sequence of *infinite length* is the set of arabic numbers. Sequences of infinite length must at least be *enumerable* to be processed by our system. They cannot be entirely enumerated and thus must be represented as a function, mapping a string to an ordinal number. xindy, has several built-in *enumerable sequences* such as arabic numbers and roman numerals. If there is a need to define more such sequences, this can be done with a user-definable function implementing the necessary mapping. In the following the notion of alphabet covers both, alphabets and enumerable sequences.

A location reference $L = s_1 \dots s_n$ *matches* a location class C iff $\forall i.s_i \in \alpha_i$. Using alphabets to declare the structure of a location class allows to directly derive an ordinal number for each s_i separately. A location reference can then be represented by the sequence of ordinal numbers of all α_i. This sequence is called the *sort key* of a location reference. This enables one to define a natural sorting order on a location class based on the *lexicographic order* of the sort key.
For example, bible verses can be described in the index style using the following definition:

```
(define-alphabet "bible-chapters" ("Genesis" "Exodus" "Leviticus"))
(define-location-class "bible-verses"
  ("bible-chapters" :sep[3] "␣" "arabic-numbers" :sep "," "arabic-numbers"))
```

This effectively matches all of the examples *Genesis 1, 31*; *Exodus 3, 12*; and *Leviticus 2, 3*. The sort key of the location reference '*Exodus 3, 12*' is then the sequence of ordinal numbers (2 3 12).[4] Sorting bible verses lexicographically based on sort keys is now a trivial task. Since some of the identified location structures have optional parts, such as the irregular structures presented above, our model allows the definition of optional suffixes as well. These location classes are named *variable location classes*. The example of the manual with inserted pages is one instance of this type of location classes.

Another problem is the potential ambiguity in the matching process. Given the location classes ([arabic][alpha]) and ([arabic][roman]), the following location references match both classes: *1c*, *2i*, *7m*, since the set of the lowercase roman numerals $\{c, i, l, m, v, x\}$ and the set of the lowercase Latin letters $\{a, \dots, z\}$ are not disjoint. We argue that this situation is unlikely to occur in practice since the reader would be confused by such location references, making this a rather pathological case.[5]

Location classes like index entries can have a recursive structure, too. Often location references of the structure ([Alpha].[arabic]) are typeset as *A 1,3,7; B 5,12* instead of *A.1, A.3, A.7, B.5, B.12*. The example shows of two location references each one containing other (sub-)location references, whereas the latter consists of five different location references. xindy offers additional declarative means to define *recursive location classes*.

3.5 Attributes

In practice location references are often typeset using different font-shapes, informing the reader of the occurence of a keyword in different meanings. For instance, in

[3] The argument `:sep` declares the following argument to be a separator.

[4] Assuming the first element of an alphabet has ordinal number '1' we obtain *Exodus*→2, *3*→3, and *12*→12.

[5] We hope so, since we actually found no meaningful counter-example.

mathematical texts one wants to optically distinguish the definition of an entity from its usage. Systems such as *MakeIndex* defined the concept of *encapsulators* used to encapsulate the location reference in a markup that caused the document preparation system to typeset the location reference as needed.

In xindy one can attach a so-called *attribute* to a location reference. It is a tag that is used for index processing purposes as well as markup tasks. The markup mechanism is described in more detail in section 4.3. We continue with a description of the processing schemes for location references offered in our model.

3.6 Attribute Groups

The analysis of indexes has shown that location references of different location classes usually appear separated from each other in an index entry. Actually, it would be of dubious value to freely mix, for instance, page numbers with section numbers. The list of location references can therefore be partitioned into subsets for each occurring location class.

Another observation is that some indexes separate the location references of a location class under certain circumstances based on criteria depending on the attributes of a location reference. For example, we might want all references to a definition of a term to appear before the references to its usage. To illustrate these ideas we use the font-shape roman for location references with the attribute *default* and the shape boldface for *definitions*. The difference between the two versions would look like this:

a) 7, **10**, 11, **12**, **14**, 15, 17

b) **10**, **12**, **14**, 7, 11, 15, 17

Sequence a) is sorted according to the sort key of the location references. In sequence b) all definitions appear first, thus the attributes define a sorting criteria which is of higher priority than the sort key. We describe these properties with the so-called *attribute groups* that are an abstraction of the observed phenomenon. An attribute group is defined by a set of attributes belonging to it, such that the defined attribute groups constitute a partition on the set of all attributes.[6] The appropriate definitions in the index style for both examples are shown here:

a) `(define-attributes ( ("definition"   "default") ))`

b) `(define-attributes ( ("definition") ("default") ))`

[6]We have not modeled the case that an attribute appears in more than one attribute group, though under certain circumstances it might be an useful extension.

The argument of the declaration `define-attributes` must be a list of all attribute groups. The only element in example a) is the attribute group consisting of both attributes. In contrast, example b) defines two attribute groups each consisting of only one attribute. In this notation it is possible to define attributes groups as needed. Location references are sorted for each attribute group separately based on the position of an attribute in an attribute group.

3.7 Relations on Location References

Beyond simply sorting the location references within an attribute group there are other tasks that must be performed. The most important one is that successive location references should under certain circumstances be joined to form a *range*. For example, the location references *11*, *12*, *13*, *14* should be joined into the range *11–14*.

Successor Relationship. To be able to join successive location references we need an exact definition of the successor relationship among location references. We have already introduced a total ordering on the location references using the sort key of a location reference, but in general this is not enough to derive the successor relationship. It may work for page numbers consisting of arabic numbers or roman numerals, but in case of hierarchically structured location classes it is not sufficient and may depend on the actual document as we show now.

For example, we have page numbers of the form ([arabic]{[alpha]}) matching the location references *11, 11a, 12*. The corresponding sort keys are (11), (11 1) and (12). We can perhaps infer that (11) could be written as (11 0) and then conclude that *11a* is a successor of *11*, but we actually don't know if *11a* and *12* are successors. As long as we do not know if there is another location[7] between *11a* and *12* we cannot build the range *11–12*, though it might be desirable in the concrete case.

What actually causes the problem is a missing bijective mapping from the sort key to the set of natural numbers, for which a well-defined successor relationship exists. If this mapping would exist, we could verify if *11a* and *12* are mapped onto two successive natural numbers to decide if *12* directly follows *11a*. There are two solutions to this problem: (a) the author of a document explicitly declares *12* to be a successor of *11a*, tedious but under certain circumstances acceptable, or (b) the document preparation system generates this information automatically and adds it to the raw index as well. We call this kind of missing information *document knowledge* since it is contained in the document itself and must be made available to the index processor for entirely solving the successor problem.

[7] Here we actually mean *location* and not *location reference*.

The Principle of Superseding. Till now we have concentrated on building ranges among location references with the same attribute. The next problem is how to solve this problem in an attribute group consisting of a mixture of location references with different attributes. First of all we observe that the location references **11** and *11* might indicate two different types of occurrences of a term on page eleven. But how should this appear in the index? Sometimes it may be desirable that there exists a natural precedence among the attributes of an attribute group. For example, we want to drop the location reference pointing to an usage of a term on a particular page in favour of its definition on the same page. This saves space in the resulting index and emphasises the importance of a definition of a term over its usage. But for other documents the author may want to let both references coexist simultaneously.

Thus the attributes of an attribute group must define a superseding relation indicating what location references with certain attributes can be dropped in favour of the same location references with other attributes.

Virtual Location References. Sometimes a range cannot be built from a set of location references with the same attribute, due to missing location references necessary to complete a sequence. For example the location references *11, 12, 14, 15* cannot be joined to a range because *13* is missing. In case of an additional location reference **13**, one could wish to build the range *11–15* and show the **13** separately resulting in the following sequence: **13**, *11–15*.

This can be accomplished if the location references with the attribute *definition* would be seen as *default* location references in the phase of range building as well. We call these location references *virtual location references*, because they do not exist as real objects but are rather a way to simplify the process of range building. A virtual object can be seen as the child of a real object from which it is derived. Thus, virtual location references appear if there is a relation among the attributes appearing in an attribute group.[8] In our case there exists the virtual location reference [*13*] behaving as a normal *default* location reference that helps to form the range *11–15*. If no range can be built, the virtual location references disappear. Another option can be to eliminate the original location reference (the one the virtual location reference is a child of), if the virtual one helps building a range. There are two ways to describe the relations among the attributes: (a) the so-called *merge-to* relation M between a source attribute and a target attribute, and additionally (b) a *drop-if-merged* relation D, for which $D \subseteq M$ must hold. Table 1 gives examples of possible combinations for some specific relations.

Examples 1–4 characterise the situation in which the attributes *definition* and *default* appear in two separate attribute groups, whereas in the examples 5–8 both

[8] Actually it may be a relation among attributes appearing in different attribute groups as well.

No.	*Ranges*				*Location references*
	none	*allowed*			11 13 14 15 17 25 **12 15 25**
			merge-to		
				drop-if-merged	
1	○				11 13 14 17 **12 15 25**
2		○			11 13–15 17 **12 15 25**
3		○	○		11–15 17 **12 15 25**
4		○	○	○	11–15 17 **25**
5	○				11 **12** 13 14 **15** 17 **25**
6		○			11 **12** 13–15 **15** 17 **25**
7		○	○		11–15 **12 15** 17 **25**
8		○	○	○	11–15 17 **25**

Table 1: Range building in presence of virtual location references

are in the same attribute group and attribute *definition* also supersedes attribute *default*. Without this restriction there exist even more alternatives. As we can see the additional relations may help to compress the list of location references under certain circumstances.

Dropping Across Hierarchies. The principle of dropping location references in favour of a more compact representation has to be generalised to variable location classes. Location references could be dropped across hierarchies if they are subsumed by location references of a higher level. For example, one may wish to transform the location references *1, 2, 2.1, 2.2, 2.2.1, 2.3, 3* of class ([arabic].{[arabic].[arabic]}) into one of the following representations:

- Building of ranges separately for some layers:

  1–3, 2.1–2.3, 2.2.1 or *1–3, 2.1, 2.2, 2.2.1, 2.3* .

- Joining at higher levels is done in favour of joining at lower levels:

  1–3, 2.1–2.3 or just *1–3* .

- Not everything is joined and the explicit enumeration of remaining location references is varied:

  1, 2, 2.1–2.3, 3

  1, 2, 2.1–2.3, 2.2.1, 3

  1, 2, 2.1, 2.2, 2.2.1, 2.3, 3 .

We will not further investigate the exact semantics of these examples, since the rules for the different variants especially in conjunction with virtual location references are complicated. One possible formalisation can be found in [8].

3.8 Cross References

Cross references are references to redirect the readers interest to other index entries. Often they are typeset as '*see*...' if there are no location references in this index entry or '*see also*...' in case the reader should also look at another index entry.

Cross references were accomplished in *MakeIndex* using the encapsulators presented previously. We have made cross references separate objects in our indexing model. The user may define cross reference classes in the index style. A cross reference can be attached an attribute in the raw index specification format identifying the class of a cross reference. This class attribute can be used to assign specific markup to cross references and allows for a much more flexible markup and grouping mechanism than other index processors.

As an additional feature we distinguish between *checked* and *unchecked* cross reference classes. Members of the former group are checked, if they point to an existing entry in the index, thus avoiding the annoying presence of cross references pointing to non-existent index entries.

4 Implementation

Like *MakeIndex* and *International MakeIndex*, xindy is a *monolithic* system. In [1] a *pipeline-based* system is presented and the authors argue in favour of using UNIX text processing tools such as `awk` to write a tailored index processor for each application. They emphasise the functional decomposition of each member of the pipeline and compare it with *MakeIndex*. In fact, the indexes they processed with their tools are simple compared to the model we presented. We believe that developing an index processor with the capabilities of our indexing model will not be possible by using pipelines. Additionally, we wanted to implement a system that is usable by many people, not only by experienced UNIX-programmers.

4.1 Why Common Lisp?

We started using C++ as the programming language of our choice. We aimed at a prototype implementation to verify the new aspects of our indexing model in practice. We have found C++ to be an inadequate language for productively building such prototypes and switched to COMMON LISP [17]. All necessary data structures

are part of the core language ready for use. The powerful macro-mechanism enabled us to implement a system without writing a dedicated parser. The LISP-interpreter is well suited for this purpose. This has encouraged us to write the production system in LISP as well.

We have chosen the freely available `clisp`-implementation[9], and have extended it with the GNU `rx` regular expression library[10] for the keyword mappings.

The system was written using the literate programming system `noweb` [13]. It consists of about 4 500 lines of LISP code and 600 lines of C code. A parser for the transformation from the TEX-specific raw index in the format used by xindy has been implemented using 150 lines of `lex` code. The printed `noweb` code covers about 150 pages of text. As a comparison, *MakeIndex* is written in 4 300 lines of C.

4.2 Design Decisions

The Raw Index. The raw index specification format understood by xindy currently consists of one polymorphic command that can be used to describe an index entry consisting of the main key, an optional print key, the location reference, its associated attribute, cross references and other options. Each front-end to xindy must produce the format of the raw index interface. Due to the length of the specification we do not describe it here. The details can be found in [7].

The Index Style. As shown in figure 1 the index processor is configured with the index style. The index style language consists of about 35 different commands, most of them are used to specify the markup of the final index. For a detailed description of all commands we refer to [7]. Since a complete description of an index style can be rather complex we have adopted a module concept that enables the decomposition of a complete index style into submodules. This allows the reuse of components simplifying the task of writing style files. We have implemented a set of predefined styles that can be used in an ad-hoc manner. Though compatibility with *MakeIndex* was not a design goal the distribution contains tools and modules, that make xindy mostly behave as a plug-in for *MakeIndex*.

The superseding relation is implicitly implemented using the attribute groups. We have implemented the superseding relation as the transitive hull over the successor relation of the list defining an attribute group. For example, the attribute group (a_1, a_2, a_3) implicitly defines the relation $\{(a_1, a_2), (a_1, a_3), (a_2, a_3)\}$. This is in our opinion an useful simplification of the model covering almost all practical needs. Building ranges is only allowed on the last hierarchy layer of a location reference.

[9] Available at `ftp://ma2s2.mathematik.uni-karlsruhe.de/pub/lisp/clisp/`.
[10] Available at `ftp://prep.ai.mit.edu/pub/gnu/`.

For variable location classes it is not supported due to the lack of experience with useful application domains. The merge-to and drop-if-merged relations are implemented.

4.3 Tagging the Index using Context-Based Markup

The tagging process directly operates on the tree representation as illustrated in figure 4. Boxed items are components that are directly derived from the raw index. The other nodes are created in the processing phase and serve to structure the index.

We have chosen to support an environment-based style of defining markup. This kind of markup is used in document preparation systems such as TEX, SGML (e.g. its instance HTML) and many others. The markup algorithm simply traverses the index tree and when a node is entered or left a *markup event* is generated. The user specifying the markup in the index style needs to establish appropriate *event bindings* that emit meaningful formatting info for the document preparation system.

Event bindings can be specialised relative to the context in which the event was generated. This scheme is mostly inspired by the STIL-project [14]. At event generation time the traversing algorithm is in a context that consists of the path in the index tree it came across from the root to the current node.

The event dispatcher must decide which of the markup schemes is the most specialised one that matches the current context. This kind of multi-argument dispatching is directly expressible in the multi-method dispatching scheme of CLOS [6]. Internally, the system generates CLOS-methods at run-time when reading the index style that specialise on the given arguments. At markup time the method dispatcher of CLOS is used to find the appropriate method.

The user interface represented by the index style is therefore itself of object-oriented nature. The user declares objects such as alphabets and location classes, of which location references are instances. Markup can be attached orthogonally to these objects using multi-argument dispatching. For users not familiar with this kind of operation we have added a feature that traces the whole markup phase so that generated events are shown and the markup is reported that is emitted in response to these events. For a detailed description of the markup model see [9].

4.4 Open Problems

A markup specific problem we encountered is that many flat indexes are typeset in the following way: *index, 1–10; ~processor, 5–9, 13; ~style, 6, 8, 13*. Thus repetitions of keywords are indicated with an *abbreviation sign* (the tilde character in our case). Allowing markup schemes of this kind can be implemented by making the information that a keyword is repeated available in the markup context. But

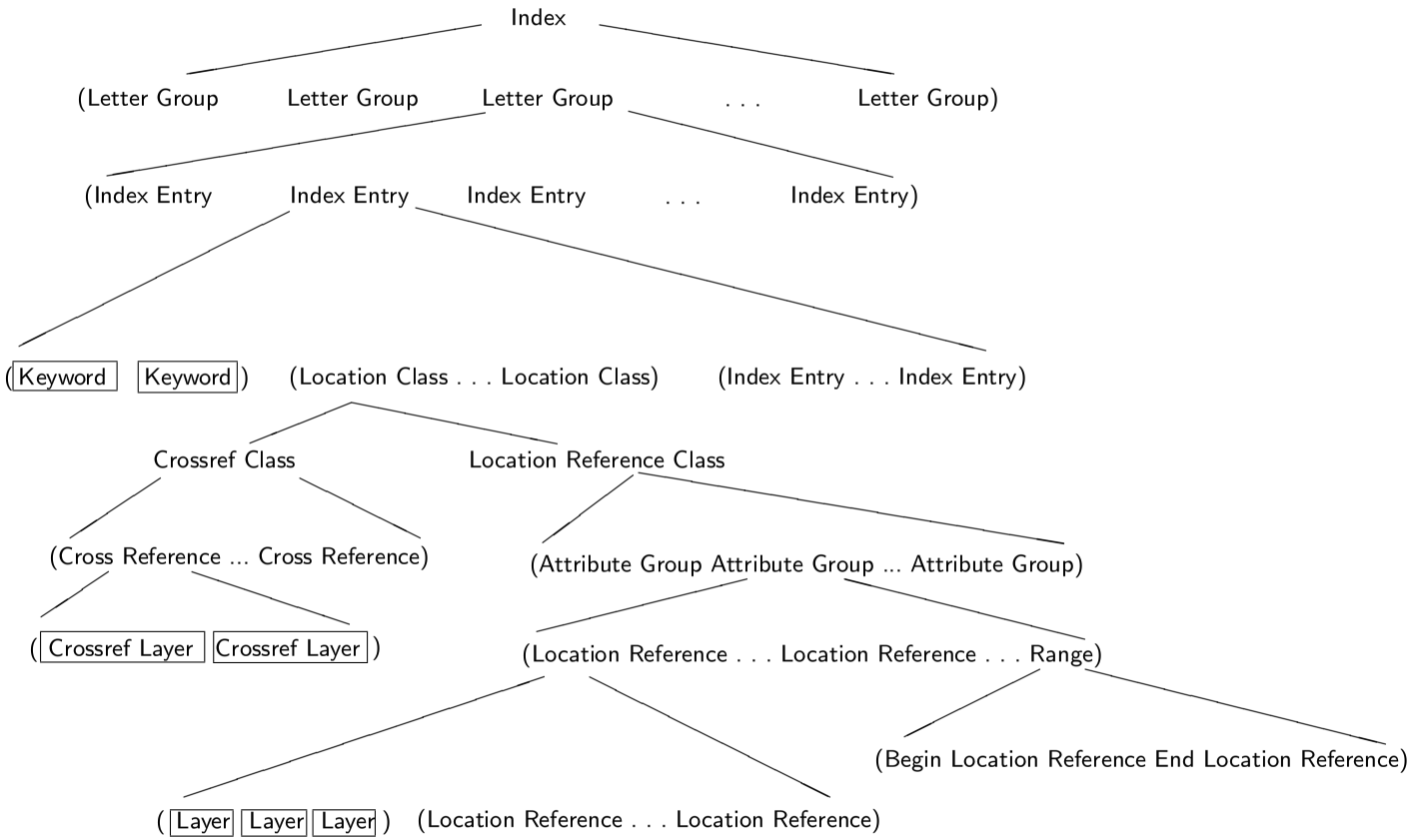

Figure 4: Skeleton tree structure of an index

for indexes appearing in textbooks this is only one part of the solution. Usually the abbreviation signs are always suppressed at the beginning of a new page in the document. This simplifies the readers search for a keyword in the index. We argue that suppressing abbreviation signs is mainly a problem that has to be solved by the document preparation system but further investigation of the interaction of document preparation systems and index processors is necessary to solve this problem.

As already noted we are currently exploring new sorting mechanisms based on an object-oriented view on letters to simplify the user-interface of our system for complex sorting schemes.

5 Conclusion

An index consists of a two-dimensional structure represented as a tree. Figure 4 illustrates the structure of the components an index consists of. The vertical dimension, the index entries and their associated keywords are quite well understood due to the results of the *International MakeIndex* project [15]. The model of the keyword mappings covers many needs. Our main contribution is a formalisation of the objects and processes that participate in the horizontal dimension, namely alphabets, location classes, attributes, attribute groups, the superseding relation, the merge-to relation and the drop-if-merged relation. We have identified the involved objects and have presented appropriate processing schemes. The details of this analysis can be found in [8].

From an abstract point of view there are several topics the vertical and horizontal dimensions share. Along both dimensions we have to solve a *classification problem*. In terms of location references we must decide the location class of a location reference, and for keywords we must decide the equality of two keywords following a *normalisation* step. In terms of keywords the keyword mappings are a solution to this problem, whereas in the horizontal dimension the different relations (superseding, merge-to and drop-if-merged) are solutions to it.

Another correspondence occurs with the sorting problem, which in both cases can be solved using appropriate mappings. In the vertical dimension we use a mapping from keywords, consisting of a list of words over the document alphabet, to another list in which each element is generated using the merge and sort mappings. This list is then sorted lexicographically. The same process appears at the horizontal dimension, except that we use a list of ordinal numbers to represent the sort key of a location reference. It is possible for the user to define arbitrary alphabets along both dimensions.

Based on this model we have implemented x̊indy, a system that is more flexible than former systems. It is tailorable to a wide range of application domains, ranging

from multi-language support to quite complex document structures such as the bible or HTML-documents, which can now be processed with our system. Its markup scheme based on a highly declarative multi-argument dispatching strategy has proven to be an adequate mechanism of describing context-based markup.

The implementation, a manual and a tutorial are available via WWW under the URL `http://www.iti.informatik.th-darmstadt.de/xindy`. The author is reachable via electronic mail under `kehr@iti.informatik.th-darmstadt.de`.

6 Acknowledgements

I would like to thank Joachim Schrod and Klaus Guntermann for their inspiring discussions, their valuable comments, and precise corrections and who made me feel comfortable in their working group. I would also like to thank Gabor Herr who was as excellent adviser in many implementation questions especially concerning Lisp. Without him this project would not be in its current state. Finally, I would like to thank Prof. Waldschmidt who has given many helpful advices to improve this report and who made it possible to finish this project.

7 References

The following books and papers were referenced in this report.

[1] J. L. Bentley and B. W. Kernighan. Tools for Printing Indexes. *Electronic Publishing Origination, Dissemination, and Design*, 1(1):3–18, Apr. 1988.

[2] P. Chen and M. A. Harrison. Automating index preparation. Technical Report 87/347, Computer Science Division, University of California, Berkeley, CA, USA, Mar. 1987. This is an extended version of [3].

[3] P. Chen and M. A. Harrison. Index Preparation and Processing. *Software–Practice and Experience*, 19(9):897–915, Sept. 1988. The LaTeX text of this paper is included in the `makeindex` software distribution.

[4] *The Chicago Manual of Style*, chapter 18. University of Chicago Press, thirteenth edition, 1982.

[5] ISO/IEC CD 14651 - International String Ordering - Method for comparing Character Strings and Description of a Default Tailorable Ordering, May 1996.

[6] S. E. Keene. *Object–Oriented Programming in Common Lisp–A Programmers Guide to CLOS*. Addison-Wesley, 1988.

[7] R. Kehr. *Manual of the* xindy *indexing system.* The manual is part of the xindy-distribution.

[8] R. Kehr. xindy – Ein flexibles Indexierungsystem. Report of a Project at the Computer Science Department, Technical University of Darmstadt, in German, Available from the author, November 1995.

[9] R. Kehr. A Simple Context-Based Markup Algorithm and its Efficient Implementation in CLOS. Technical report, Technical University of Darmstadt, 1997. To appear.

[10] D. E. Knuth. *The* TEX*book*. Addison-Wesley, 1984.

[11] L. Lamport. LATEX: *A Document Preparation System*. Addison-Wesley, 1986.

[12] `nroff`. The nroff/troff document formatters are usually part of the UNIX system environment.

[13] N. Ramsey. Literate-programming can be simple and extensible. Technical report, Department of Computer Science, Princeton University, Nov. 1994.

[14] J. Schrod. STIL– SGML Transformations in LISP. This system is available at `ftp://ftp.th-darmstadt.de/pub/text/sgml/stil/`.

[15] J. Schrod. An International Version of MakeIndex. *Cahiers GUTenberg*, 10(10–11):81–90, Sept. 1991.

[16] J. Schrod and G. Herr. *MakeIndex Version 3.0*, Aug. 1991.

[17] G. L. Steele Jr. *Common Lisp–The Language*. Digital Press and Prentice-Hall, 12 Crosby Drive, Bedford, MA 01730, USA and Englewood Cliffs, NJ 07632, USA, second edition, 1990.