Narzędzia

xindy — A Flexible Indexing System*

Roger Kehr

Abstract

Whilst MakeIndex *is an index processor which is suitable for the production of indexes in conjunction with many text formatters, its support for non-English languages is weak and a new version called* International MakeIndex *has been presented, for processing international documents. The improvements concentrate on the internationalization of the sorting process for keywords in an index. Though it substantially improves the availability to sort new languages, there are still weaknesses in the processing model largely inherited from* MakeIndex*. Through the experiences gained from the* International MakeIndex *project we implemented a new index processor* xindy *that (a) improves the sorting of index entries at a finer granularity than* International MakeIndex*, (b) offers new mechanisms for processing structured location references besides page numbers and Roman numerals, and (c) allows for complex mark-up schemes.*

Keywords: index processor, *MakeIndex*, *International MakeIndex*, structured location references, context-based mark-up.

1 Existing Systems

The probably most-used index processor in the TEX community is *MakeIndex* [1]. It is independent from the document preparation system, and must be adapted to a particular system using a configuration file usually referred to as the *index style*. *MakeIndex* merges and sorts *index entries*, sorts the *location references* (such as page numbers), builds ranges (if possible and desired), and generates the index with mark-up specified in the index style. Most of its functionality is hard-wired into the application itself, which is sufficient for most of the typical indexes. It works well in conjunction with English texts, but if other languages have to be processed, the sorting model of *MakeIndex* shows its drawbacks. Sorting index entries is based on the lexicographic order of the keywords. If this does not match the intended sorting order of the index, *MakeIndex* cannot be used. The separation of the *print key* from the *sort key* offered in *MakeIndex* is only a partial solution that leads to annoying and error-prone specifications in the document source. This and other problems are discussed in detail in [4].

The *International MakeIndex* [4, 5] system is based on *MakeIndex* and has been enhanced to support user-defined rules for the specification of language-dependent sorting rules, simplifying the treatment of non-English languages. Based on the experiences from the *International MakeIndex* project, we have designed and implemented the index processor xindy [3, 2]. It contributes to the following aspects of indexing which will be described throughout the rest of this paper.

Sorting Model. xindy refines the sorting model of *International MakeIndex* to achieve even better support for complex sorting rules. With this model, the sorting rules for many of the currently spoken languages can be expressed.

Structured Location References. It introduces a clean model of handling structured location references which were supported in *MakeIndex* only on an *ad-hoc* basis. This allows for the specification and correct processing for a large class of existing enumeration schemes.

Context-based Mark-Up. As a result of the new model, the structure of an index has become more complex and a new mechanism for specifying mark-up based on context information has been developed.

2 Sorting Index Entries or How to Sort French

The most obvious problems with *MakeIndex* is its lack of support for sorting index entries of non-English languages. Its sorting mechanism is essentially based on the ISO-Latin alphabet, which is not adequate for most languages. The *International MakeIndex* system has introduced the concept of *merge* and *sort mappings*. These mappings consist of *string rewrite rules* that are applied to a keyword to obtain a new keyword that is used for merging and sorting purposes. The mapping steps are shown in Figure 2.

$$\text{key} \xrightarrow{\textit{merge mapping}} \text{merge key} \xrightarrow{\textit{sort mapping}} \text{sort key}$$

Figure 1: Keyword mappings implemented in *International MakeIndex*.

* Referat prezentowany na X Europejskiej Konferencji TEX-owej (29 Marca – 1 Kwietnia 1998, St. Malo, Francja). Przedruk za zezwoleniem z numeru 28/29 Cahiers GUTenberg.

The merge mapping is used to *normalize* keywords, i.e. to indicate that two different writings for a word should be treated equally. For example, one can define that the character sequences '\"a' and 'ä' (the former is the TEX-notation of the latter) are to be treated equally. After this normalization step, which merges different index entries from the index into a compound index entry, the index entries must be sorted. The sort mapping transfers the merge key into a sort key that reflects the lexicographic order of the index entry. For example, one possible rule is to map '\"a' onto 'ae' which is sometimes useful in German indexes. Hence, the sort rules should be written in such a way that the resulting sort key reflects the order of the index entry correctly. This mapping scheme has been implemented in *International MakeIndex*.

Though this mechanism is a major improvement over the original *MakeIndex*, it still does not cover important cases which often appear in practice. As a running example we sort the French words *cote*, *côte*, *côté*, and *coté*. The French sorting rules [6]—as well as other language sorting rules—have the concept of *sorting phases* that are applied successively to obtain a total order on a given set of keywords. The French rules say that in the first phase the diacritical marks should not be considered at all, and the non-diacritical counterparts should be used instead. This means that for the words above there is no distinction in the first sorting phase at all. In a subsequent phase letters with diacritical marks have to follow letters without diacritical marks. Additionally, the lexicographic order is from right to left. This exactly yields the words in the order shown above. Other languages such as German also have the concept of sorting phases though they usually stay with the left-to-right lexicographic order.

From an abstract point of view, the model needs to be enhanced by the concept of multiple sorting phases and possible variations in the direction in which the lexicographic order should be processed. Figure 2 shows the mapping scheme that is implemented in xindy.

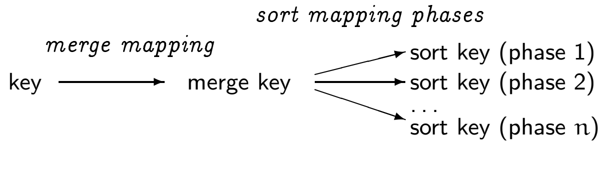

Figure 2: Keyword mappings implemented in xindy.

It supports the user-defined specification of sort rules for several independent sort phases. For sets of keywords that are equal in sort phase n, the sort rules of phase $n+1$ are applied to obtain a new order. This is done successively until a total order on the sort keys is derived or no more sorting phases remain. To achieve a better overview on what happens in a sorting phase, xindy offers means to debug the keyword mappings in detail.

3 Location References or How to Sort Bible Verses

One of the initial reasons for the development of xindy was to study the inherent structure of location references in an indexing model. We define a *location reference* as the entity that references a concrete location in a document. Besides page numbers and Roman numerals one can think of locations such as Bible verses like *Genesis 1:31*; *Exodus 1:7*; *Leviticus 2:3*. An index processor must provide solutions to at least two different aspects of location references: (a) the ability to sort correctly these location references, and (b) to form *ranges* of location references, if possible and desired. Mathematically speaking, (a) requests a *total order* on the location references, whereas (b) needs a *successor-relationship* for location references. The total order enables one to sort the location references unambiguously, and the successor-relationship tests for a potential *joinability* of two location references to form a range.

A closer look reveals that there usually is an inherent structure that gives information about the two relationships. For example, the location references *Genesis 1:31*; *Exodus 1:7*; and *Leviticus 2:3* consist of the name of the book, the number of the chapter, and the number of the verse. If we need to sort these location references, we usually sort them in the first phase according to the book they belong to. The order of the books in the Bible is fixed and does not follow any lexicographic convention. Inside a book, sorting is done first according to the chapter number and then according to the verse number in a chapter. Therefore the structural entities can be described in form of the three *alphabets*:

$$\begin{array}{ll} \textit{book} & \in \{\textit{Genesis, Exodus, Leviticus}\} \\ \textit{chapter} & \in \text{set of Arabic numbers} \\ \textit{verse} & \in \text{set of Arabic numbers} \end{array}$$

For each of these alphabets there exists a total order and a well-known successor-relationship.

Hence, sorting index entries is a problem of lexicographically sorting the components of a location reference. A structural description in this sense is called a *location class* serving as a template for concrete instances of this class, in our case concrete Bible verse references.

To enable xindy to process location references, a definition of the location classes has to be specified—consisting of a sequence of alphabets and separators—as encountered in the raw index. As location references are read, it tries to match the location references (which are available as a plain string) against the location classes it knows about, and in case of a match it is able to decompose the structure into its components. A sample specification of this location class in xindy is as follows:[1]

```
(define-alphabet "bible-chapters"
   ("Genesis" "Exodus" "Leviticus"))

(define-location-class "bible-verses"
   ("bible-chapters" :sep "␣"
    "arabic-numbers" :sep ":" "arabic-numbers"))
```

The argument :sep declares the following argument to be a separator. The first description defines the alphabet *bible-chapters* that consists of the three enumerated letters[2] *Genesis*, *Exodus*, and *Leviticus*. The second definition composes a location class named *bible-verses* based on the new alphabet, the separator characters (solely used for matching the location references), and the built-in alphabet of Arabic numbers. This description essentially defines the *grammar* of the verses occurring in the input. This description enables xindy to correctly sort all instances of the class *bible-verses* and additionally enables it to join location references into ranges, if desired. xindy basically allows for the definition of new alphabets and location classes, which may be of variable length (such as 1, 1.1, 1.1.1, ...) as well. It offers a wide range of specifications how to join location references to form ranges.

A new concept called *location reference attributes* can be used to tag location references with additional information that declares a location reference to be of a particular type, such as a reference to a definition of an item, another for its occurrence, and so on. *MakeIndex* introduced the concept

[1] xindy uses a Lisp notation for the definition of an index style.

[2] A letter is basically a sequence of characters of the underlying document alphabet.

Jak zacząć pracę z xindy

Rozpoczynamy od pobrania pakietu albo z któregoś z węzłów CTAN albo podręcznego archiwum GUST (katalog /indexing/xindy). Pakiet jest dystrybuowany w dwóch archiwach: jednego z plikami wykonywalnymi i pomocniczymi dla konkretnej platformy systemowej (dla Linux-a będzie to xindy-2.0b-i386-linux-elf.tar.gz) oraz drugiego zawierającego pliki niezależne od używanego systemu xindy-2.0b.tar.gz.

Poniżej opiszemy instalację xindy dla Linux-a. Oprócz niej w chwili pisania tego tekstu dostępne były także wersje na inne Unix-y, MS Win 9*/NT a nawet dla DOS-a.

Po rozpakowaniu archiwum xindy-2.0b.tar.gz przechodzimy do utworzonego katalogu xindy. Teraz rozpakowujemy archiwum z plikami systemowymi co spowoduje umieszczenie jego zawartości w katalogu xindy/binaries. Takiej struktury katalogów oczekuje program instalacyjny.

Teraz w katalogu xindy tworzymy dowiązanie: ln -l xindy-2.0b current a potem przechodzimy do katalogu xindy/binaries/i386-linux-elf i piszemy make all. Z kolei make testsuite uruchamia zalecaną serię testów. Wreszcie uruchomienie make install spowoduje zainstalowanie wszystkiego tam gdzie trzeba. (Jeśli chcemy wiedzieć gdzie, to przed make install należy uruchomić make -n install).

Do budowania skorowidzów z plików TEX-owych najlepiej (przynajmniej na początku) skorzystać z programu makeindex4 (zainstalował się razem z xindy). Jest to w istocie skrypt napisany w perl-u „emulujący” program makeindex. Aby posortować plik qq.idx wystarczy napisać makeindex4 qq i po chwili otrzymamy plik qq.ind. Problem w tym, że polskie teksty nie będą poprawnie posortowane.

Reguły sortowania są definiowane w modułach, a nie jak w przypadku makeindex-a zaszyte w programie wykonywalnym. W przypadku makeindex4 wykorzystywane są reguły z pliku makeindex4.xdy znajdującego się w katalogu ~/xindy/modules/tex/ (gdzie „~” oznacza korzeń drzewa biblioteki modułów xindy (np. /usr/local/lib). W przypadku j. polskiego musimy określić co najmniej dwie rzeczy: porządek sortowania znaków oraz podać w jaki sposób posortowany skorowidz ma być dzielony na grupy (najczęściej poszczególne grupy to hasła zaczynające się od tej samej litery – bez względu na to czy jest ona duża czy mała).

Poniższy sposób można określić mianem hakerskiego: do pliku makeindex4.xdy wstawiamy dziewięć poleceń według następującego schematu:

```
(define-letter-group "ą" :after "a" :before "b"
  :prefixes ("a~~"))
```

następnie dla każdej litery łacińskiej wpisujemy 26 następujących poleceń:

```
(define-letter-group "a" :prefixes ("A"))
```

oraz 9 prawie takich samych linijek dla diakrytyków:

```
(define-letter-group "ą" :prefixes ("A~~"))
```

z tym, że dla litery „ź” będzie to: (define-letter-group "ź" :prefixes ("Z~~")) a dla „ż” (define-letter-group "ż" :prefixes ("Ż~~")).

Teraz dla każdej litery łacińskiej wpisujemy: (sort-rule "ą" "Ą"), itd., a dla polskich: (merge-rule "ą" "a~~"), (merge-rule "Ą" "A~~") itd. przy czym znowu dla „ż” jest trochę inaczej, a mianowicie: (merge-rule "Ż" "Ź~~") i (merge-rule "ż" "ź~~"). Tak zmieniony makeindex4.xdy prawidłowo sortuje i dzieli na grupy polskie teksty. Więcej informacji można znaleźć w dokumentacji pakietu.

separate attributes, no ranges	11 13 14 17 **12 15 25**
mixed attributes, no ranges	11 **12** 13 14 **15** 17 **25**
mixed attributes, ranges, not subsumed	11 **12** 13–15 **15** 17 **25**
separate attributes, ranges, subsumed	11–15 17 **12 15 25**
separate, ranges, subsumed and ommitted	11–15 17 **25**

Table 1: Output of location references using different policies.

of *encapsulators* for mark-up purposes. We have generalized this concept to offer more flexibility in the sorting and merging phases, for example to indicate that a definition of an item should subsume its occurrence on the same page, to save space on the resulting output. We give an example of the possibilities to output a sequence of page numbers tagged as definitions and occurrences based on different policies, such as separation of attributes, building ranges where possible, and subsuming occurrences of location references with a definition on the same page. Table 3 illustrates some of the possibilities that can be specified with x̊indy (definitions of items are shown in boldface).

We hope that one might get an impression of what kind of processing location references with x̊indy is possible in general and what different levels of compression in the resulting output can be achieved.

4 Context-Based Mark-Up

In addition to the indexing features introduced, there is a need for a general model to specify mark-up easily. The set of available mark-up tags is relatively limited and fixed in *MakeIndex*. In x̊indy, the final index, after all processing steps have been performed, is internally represented as a tree. Mark-up is implemented with a tree-traversal algorithm that starts at the root node and visits each node of the tree in a depth-first manner. Every time a node is entered or left, an *event* is generated. The user is now able to bind mark-up tags to each of these events. For example, the binding

```
(markup-locref :class "bible-verses"
:open "\\textit{" :close "}")
```

denotes that the location references of class *bible-verses* should be surrounded by the mark-up tags '\textit{' and '}' defining an italicized TeX-mark-up. If the parameter :class would have been omitted, this specification would match all location classes, therefore acting as a default setting. If a mark-up event is raised, the *event dispatcher* is responsible for finding the most specific binding that matches this event. Events are parameterized by information from the context in which they were raised. For example, the events for location references contain information about the current location class, the current attribute, and the depth it is placed in. Bindings can be defined to any subset of the set of parameters. The tag :open is the string to be emitted if a node is entered, whereas :close defines the corresponding binding if a node is left.

At a first glance this scheme sounds more complicated than it is in practice. Debugging facilities exist to help users specifying mark-up bindings. The whole mark-up phase can be traced, events are shown, and the bindings can be seen when they are activated. Usually only a small portion of all possible events actually need bindings. Just to give an impression, Table 4 illustrates what results can be specified with x̊indy by mark-up bindings only.

More examples and detailed descriptions, illustrating how these results can be obtained, are described in the documentation that comes with x̊indy.

5 Implementation, Availability and Distribution

x̊indy is largely implemented in Common Lisp. We have chosen the freely available CLISP-implementation[3], and have extended it with the GNU rx regular expression library[4] for the keyword mappings. 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 x̊indy has been implemented using 150 lines of lex code. As a comparison, *MakeIndex* is written in 4 300 lines of C.

The full implementation is available under the conditions of the GNU General Public License. Its home-page with further links is accessible at our web

[3] Available at ftp://ftp2.cons.org/pub/lisp/clisp/source/.

[4] Available at ftp://prep.ai.mit.edu/pub/gnu/.

standard tagging	*A.1, A.3, A.7, B.5, B.12*
emphasizing the chapters	***A**.1, **A**.3, **A**.7, **B**.5, **B**.12*
additional compression of sections	***A** 1,3,7; **B** 5,12*
standard tagging	*Genesis 1:31*; *Exodus 1:7*
different mark-up for verses	*Genesis **1**(31)*; *Exodus **1**(7)*
verbose mark-up	*Genesis chap. 1, 31*; *Exodus chap. 1, 7*

Table 2: Output of location references with different mark-up.

site `http://www.iti.informatik.tu-darmstadt.de/xindy/`. Source and binary distributions are available at CTAN in directory pub/indexing/xindy/. It is currently available in source and binary distributions for several Unix platforms and OS/2. Efforts to port xindy to Windows 95/NT platforms are underway and are likely to be finished soon. There is lots of documentation available in various formats and more detailed examples describe how to use xindy, especially its treatment of sorting index entries and managing location references.

6 Conclusion

xindy is a new index processor that improves three major aspects of indexing. It offers new means for the specification of sorting rules for the index entries covering languages from English (with its simple sorting model) to French (with rather complex rules). Users will find it a valuable tool that allows them to process indexes in their native language.

Another improvement is achieved by the concept of location references and attributes that allows structured references (such as Bible verses or chapter/page enumeration schemes) to be processed, sorted, and joined in various ways. This is accompanied by a powerful context-based mark-up scheme that offers very fine control over the process of tagging the final index, suitable for different mark-up languages such as TEX, and instances of SGML or XML documents.

From the user interface perspective, the new module scheme allows for the development of index styles by authors that can be reused by end-users with minimal effort. Authors are welcome to contribute to this project by writing modules for different languages and mark-up for different back-ends.

Acknowledgments

I would like to thank Joachim Schrod and Klaus Guntermann for their inspiring discussions essentially in early phases of this project. Further, I would like to thank Gabor Herr, who was as excellent adviser in many implementation questions. Additionally, I would like to thank the participants in our discussions on the xindy mailing list, most notably Chris Rowley, and finally Ulrich Gräf and Prof. Waldschmidt for giving valuable hints for the improvement of this paper.

References

[1] P. Chen, M. A. Harrison, *Index Preparation and Processing*, Softw.–Pract. Exp. 1988, 19(9), p. 897–915. The LaTeX text of this paper is included in the `makeindex` software distribution.

[2] R. Kehr, *A Simple Context-Based Markup Algorithm and its Efficient Implementation in* CLOS. Technical Report TI/12, Computer Science Department, Technical University of Darmstadt, June 1997.

[3] R. Kehr, xindy – *Definition of an Indexing Model and it Implementation.* Technical Report TI/11, Computer Science Department, Technical University of Darmstadt, May 1997.

[4] J. Schrod, *An International Version of MakeIndex.* Cahiers GUTenberg 1991, 10 (10-11), p. 81–90.

[5] J. Schrod, G. Herr, *Makeindex Version 3.0.* Technical Report, Technical University of Darmstadt, August 1991.

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

◇ Roger Kehr
kehr@iti.informatik.tu-darmstadt.de