xindy

A Flexible Indexing System

Roger Kehr, Institut für Theoretische Informatik, TH-Darmstadt
Tutorial, Edition March 1997 for xindy version 1.1

This is a tutorial for the **xindy** *indexing system [5].* **xindy** *is a general purpose index processor that can be configured for a variety of document preparation systems. We will demonstrate step by step the incremental specification of a complete index style. The examples are based on indexes that are delivered with the system so one can get acquainted with it practically.*

Contents

1 About xindy

xindy means flexible **ind**exing s**y**stem. It is an indexing system that can be used to generate book-like indexes for arbitrary document preparation systems. This term includes systems such as TEX [1] and LATEX [2], the Nroff-family or SGML-based systems (e.g. HTML) that process some kind of text and generate indexing information. It is not fixed to any specific system, but can be configured for a wide variety of purposes.

Authors sometimes wish to include an index into their document, but very often their document preparation systems aren't able to produce indexes on their own, or the capabilities of the built-in indexers are insufficient and produce ugly looking results. More often they use separate specialized tools for this purpose, sometimes called *index processors*.

In comparison to other index processors **xindy** has several powerful features that make it an ideal framework for describing and generating complex indexes. Its most interesting features are

Internationality

xindy can be configured to process indexes for many languages with different letter sets and different sorting rules. For example, many roman languages such as Italian, French, Portuguese or Spanish contain accentuated letters such as À, Á, ñ. Other languages from northern Europe have letters like Ä, Ø, æ or ß which often can't even be processed by many index processors let alone sorting them correctly into an index. The **xindy** system can be configured to process these alphabets by defining *sort* and *merge rules* that allow expressing language specific rules. One example of such a rule would be

```
(sort-rule "ä" "ae")
```

defining that a word containing the umlaut-a will be sorted as if it contained the letters `ae` instead. This is one form of how the umlaut-a ("ä") is sorted into german indexes. With an appropriate set of rules on can express the complete rules of a specific language.

User Definable Location Types

Locations are the entities an index entry points to. Often used locations are *page numbers*, *section numbers*, etc. **xindy** allows to define new kinds of types to be composed of basic entities like arabic numbers, roman numerals, letters, etc. which can be used to describe new index types. Examples of these entities can be books that have a page numbering scheme that starts from 1 for each new chapter resulting in page numbers of the form *1-13, 2-15, 2-20*, etc. This type of locations can often be found in computer manuals. A more complex structure is represented by the locations *Psalm 46, 1-8* and *Genesis 1, 31*. Even the structure of bible verses can be described allowing **xindy** to correctly sort and process indexes for documents with a completely different structure.

Highly Configurable Markup

Since a index processor is only one component in a document preparation system it ought to fit smoothly into the complete environment. Many text processing systems use the concept of *environments* that can be used to describe the markup of the text entities. Our approach is based on this concept which has proved to be expressive enough for most applications. As an example one can view systems such as TEX, SGML and its instance HTML, GNU Info, RTF and the Nroff-familiy of document formatters. The markup of an index can be defined for all of these systems in a very comfortable though extremly powerful way.

2 A Basic Introduction

This section incrementally introduces the most important aspects of the system. After reading this chapter you should be able to specify about 80% of the commonly used indexes. The examples are demonstrated with a TEX markup so one can easily typeset the results **xindy** produces. You need LATEX2e and the ISO-Latin enhancements that come with the `inputenc` package to run the following examples. Also the **xindy** system must already be installed.

2.1 Running xindy

Create a new directory somewhere and copy some files from the distribution directory `Doc/tutorial/` by typing

```
eg$ mkdir tutorial
```

```
eg$ cd tutorial
eg$ cp <distrib-dir>/Doc/tutorial/*.raw .
```

with distrib-dir replaced by the actual location. Now create a file style1.xdy with the following content:

```
;; This is a first example using 'xindy'.

(define-location-class "page-numbers" ("arabic-numbers"))
(define-attributes (("definition" "usage")))
```

Now run **xindy** by typing (the switch '-l' is the lowercase letter 'L')

```
eg$ xindy -l ex1.xlg style1.xdy ex1.raw
```

You should see something like

```
This is 'xindy' version 1.1 January 1997 (some-arch).
Copyright (c) 1996,1997 ...

Opening logfile "ex1.xlg" (done)
Reading indexstyle...
Loading module "style1.xdy"...
Finished loading module "style1.xdy".
Finished reading indexstyle.
Finalizing indexstyle... (done)

Reading raw-index "ex1.raw"...
Finished reading raw-index.

Processing index... [10%] [20%] [30%] [40%] [50%] [60%] [70%] [80%] [90%] [100%]
Finished processing index.

Writing markup... [10%] [20%] [30%] [40%] [50%] [60%] [70%] [80%] [90%] [100%]
Markup written into file "ex1.ind".
```

xindy has now successfully compiled the index ex1.raw using your index style style1.xdy. The result is now stored in file ex1.ind. You can view this file but currently it only contains an unreadable mix of data.

But now let's come back to our index style. The syntax of the command is in a Lisp-like form with lots of braces, looking a little bit weird, but you'll soon get used to it. What is the meaning of the two commands we specified? The first command informed **xindy** that we like to process page numbers. We do this by defining a new *location class* named page-numbers. The page numbers consist of arabic-numbers as we might expect but this is not necessarily true—imagine your page numbers consisted of roman numerals instead. When reading the *raw index* contained in file ex1.raw **xindy** checks all locations if they match all known location classes. Since in our example the only known location class is the class of page numbers which are written using arabic digits, all locations will be checked if they are correct page numbers.

The second command tells **xindy** that we use two types of attributes for location references. Most often the locations in an index denote different meanings. For example, in mathematical texts one will distinguish the *definition* of a mathematical term from its *usage*. Sometimes these are typeset using different font shapes such as *italic* or font series such as **boldface**. Each location has an associated attribute which, if it is unspecified, defaults to the attribute `default`. With this command you have made these attributes known to the system, which makes it possible to assign different markup to these attributes later on.

2.2 Adding some Markup

Until now you haven't seen something exciting, so its time to specify some markup. Add the following lines to our index style:

```
(markup-index :open  "~n\begin{theindex}~n"
              :close "~n\end{theindex}~n"
              :tree)

(markup-locref :class "page-numbers" :attr "definition"
               :open  "{\bf " :close "}")

(markup-locclass-list :open "\quad")
(markup-locref-list :sep ", ")
```

Now run **xindy** again and afterwards LaTeX:

```
eg$ xindy -l ex1.xlg style1.xdy ex1.raw
eg$ latex ex1.tex
```

You can view `ex1.dvi` with your prefered viewer (maybe `xdvi` or something else) to get a first impression of your results. Maybe your are not satisfied (for sure you aren't), because it still looks very confusing. What did the above rules tell **xindy**? When you view the file `ex1.ind` which is the result **xindy** generates, you'll recognize some of the *markup tags* you specified. The following is an excerpt of this file:

```
\begin{theindex}
  academia\quad{}{\bf 1}acafetado\quad{}{\bf 2}acalmar\quad{}{\bf 4}
  açafrão\quad{}{\bf 3}indexflat\quad{}1hierarchical\quad{}2
  veryhierarchical\quad{}3impressive\quad{}4saber\quad{}{\bf 7}
  sabor\quad{}{\bf 8}sabão\quad{}{\bf 6}sábado\quad{}{\bf 5}
\end{theindex}
```

First of all you'll see that the file starts with the string `\begin{theindex}` and ends with `\end{theindex}`. Additionally some locations are correctly enclosed into a TeX macro that typesets them in shape boldface, whereas others aren't. The boldface ones are all those locations from the raw index that have the attribute `definition`.

The `:open` and `:close` keyword arguments each take a string as argument. The first one is written to the file when *opening* an enviroment, whereas the latter one *closes* an environment. What we have specified is the markup for the whole index (which is actually printed only once) and the markup for all locations of class `page-numbers` which own the attribute `definition`. Here we have cleanly separated the structured markup from the visual one, allowing an easy redefinition if we decide, for example, to markup the `definition`-locations in italics instead of boldface.

Some words on *keyword arguments* and *switches*. Keyword arguments such as `:open` or `:close` always take exactly one argument which must be positioned right after the keyword separated by a whitespace (a blank or a tab-stop). Switches don't take any arguments. For example, `:tree` in the command `markup-index` is a switch and thus it does not take an argument. We will use this terminology throughout the rest of this document.

The third command caused **xindy** to insert a horizontal space between the keyword and the locations (the TEX command `\quad{}` simply inserts a specific horizontal space). The last command caused **xindy** to separate all location references from each other with a comma followed by a blank, independently of any location class.

As you already may have observed, the tilde sign (~) serves as a *quoting character*.

We continue specifying markup to get a printable result by adding more markup:

```
(markup-indexentry :open "~n  \item "          :depth 0)
(markup-indexentry :open "~n    \subitem "     :depth 1)
(markup-indexentry :open "~n      \subsubitem " :depth 2)
```

This assigns different markup for the different hierarchy layers of the indexentries. Our index is hierarchically organized. It contains items which themselves contain more sub-items which also might contain sub-sub-items. Each layer is started by a different markup which is correctly assigned with the `:depth` keyword argument. The layers are numbered by their *depth* starting from zero.

Now run **xindy** and TEX again and enjoy your first index. It's easy, isn't it! The output `ex1.ind` looks like the following:

```
\begin{theindex}

 \item academia\quad{}{\bf 1}
 \item acafetado\quad{}{\bf 2}
 \item acalmar\quad{}{\bf 4}
 \item açafrão\quad{}{\bf 3}
 \item index
    \subitem flat\quad{}1
    \subitem hierarchical\quad{}2
    \subitem very
      \subsubitem hierarchical\quad{}3
      \subsubitem impressive\quad{}4
 \item saber\quad{}{\bf 7}
 \item sabor\quad{}{\bf 8}
 \item sabão\quad{}{\bf 6}
 \item sábado\quad{}{\bf 5}

\end{theindex}
```

Hmm, as you might have seen there are several problems that need further investigation. The index contains some Portuguese words that are printed correctly but should appear at other positions inside the index. For instance, the word *sábado* should appear before the word *saber* since *á* must be sorted as if it were simply an *a*. The reason why these words are not sorted correctly is simple—the accentuated letters have codes beyond position 128 in the ISO-Latin alphabet. Sorting based on these codes yields this incorrect order.

What to do? We can define for each of the words containing these special characters an explicit *print key*. The print key describes the printed representation of the keyword whereas the *key* or *main key* is used for sorting and merging. A very tedious task which is not a very clever solution since in a non-english language many many words contain these special cases. We follow the way **xindy** offers: *keyword-mappings*.

2.3 Keyword Mappings

What are keyword mappings for? A good question. I'll try to give some answers.

- *Merging of differently written words*. Some text formatting systems allow different writings for the same word. For example, TEX can be used with ISO-Latin characters as well as with its own character sequences. If a document is composed from smaller ones possibly written by different authors using different forms of writing the index entries, the same terms may happen to be written differently and consequently we need a mechanism to identify them as equal.
- *Specifying the sort order*. As outlined in the previous section it is really difficult and error-prone to specify the sort key for each keyword explicitly. Sometimes the sort order is even different for the type of the document, as it happens in German, where two different types of sortings exist, one for everyday indexes and one for dictionaries. The sort order actually defines the position of arbitrary language-specific letters inside of an index.

A detailed elaboration of these ideas can be found in the paper *An International Version of MakeIndex* by Joachim Schrod [3]. It describes the ideas that led to modifications on one of the ancestors of the **xindy** system: `makeindex` [4].

The keyword mappings are as follows. The *merge key* is generated from the *main key* with the so called *merge mapping*. The merge mapping can be specified with the command `merge-rule`. The *sort key* is derived from the merge key using the *sort mapping* specified with the `sort-rule` command. The following scheme shows this mapping process:

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

We will use this command now to define a suitable sort mapping that fits our needs:

```
(sort-rule "à" "a")
(sort-rule "á" "a")
(sort-rule "ã" "a")
(sort-rule "è" "e")
(sort-rule "é" "e")
(sort-rule "ç" "c")
```

These rules define mappings from a keyword to a *normalized* version. In the logfile `ex1.xlg` these transformations are logged so that one can see how these mappings are performed. In this example we do not need any `merge-rule` but we will see applications in further examples.

Running **xindy** and TEXing the result now places the indexentries at the right positions.

The result is now quite satisfying if the index entries weren't clumped together that much. We usually want the different index entries beginning with the same letter be optically separated from the ofhers. This improves readability and there must be a way to accomplish this—the *letter groups*.

2.4 Letter Groups

To group indexentries we must define what indexentries form a group. The clustering is done by matching the keywords' prefixes (taken from the *sort key*) with a user-defined table of prefixes and define appropriate markup that separates the groups from each other. Here it goes.

```
(define-letter-groups
  ("a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
   "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"))

(markup-letter-group-list :sep "~n\indexspace")
```

This defines the given list of letter groups. When forming the letter groups, each letter group is checked if it matches a prefix of the indexentries' sort key. The longest match assigns the index entry to this letter group. If no match was possible the index entry is put into group `default`.

The result now looks much better than before. You have now learned the basic features that you need to specify everyday indexes. In the next chapter we'll continue to make you an expert in indexing.

3 Advanced Features

In the following sections you'll learn more about the features of **xindy**. We'll show you how you can define your own location classes, specify the letter groups in more detail and bring you close to more specfic markup features. After reading this chapter you should be able to master about 95% of the commonly used indexes.

3.1 Location Classes

We continue using a more complex index. Copy the current style to a new file (now `style2.xdy`) and run **xindy** on the raw index `ex2.raw` by typing:

```
eg$ cp style1.xdy style2.xdy
eg$ xindy -l ex2.xlg style2.xdy ex2.raw
```

You should see some error messages indicating that something is unknown to **xindy**. What has happend? The messages should be similar to the following snapshot:

```
 ...
WARNING:
location-reference "B-5" did not match any location-class! (ignored)
WARNING:
location-reference "C-8" did not match any location-class! (ignored)
WARNING:
location-reference "iv" did not match any location-class! (ignored)
WARNING:
location-reference "ii" did not match any location-class! (ignored)
 ...
```

The index contains new, and therefore unknown, location classes. The first one has an appendix-like style, whereas the second one seems to be lowercase roman numerals. **xindy** reported, that it doesn't know these locations and therefore knows nothing about their internal structure. We make them known to the system by adding the following commands to the style file.

```
(define-location-class "roman-pages" ("roman-numerals-lowercase"))
(define-location-class "appendices" ("ALPHA" :sep "-" "arabic-numbers"))
(define-location-class-order ("roman-pages" "appendices"))
```

The first command tells **xindy** that there exist some page numbers that are written with roman lowercase letters. The second one defines the structure of the appendix locations. They consist of three elements, a capital letter, a separator—which is a hyphen in our case—and finally an arabic number. To be able to correctly distinguish separator strings from the names of the known *basetypes*, the argument `:sep` indicates, that the following is a separator. The last command simply says that the locations which are roman numerals shall appear before the appendix locations. So far we know three different basetypes. The built-in basetypes of **xindy** are:

`arabic-numbers`

contains all non-negative numbers beginning with zero: 0, 1, 2, ...

`roman-numerals-uppercase`, `roman-numerals-lowercase`

the roman numerals I, II, III, (IIII/IV), V, ... It contains the old and new roman numeral system. Two versions for uppercase and lowercase letters are supported.

`ALPHA`, `alpha`

The US-ASCII alphabet letters (uppercase and lowercase letters).

`digits`

the digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 in this order.

'Fine,', you'll say, 'but what if my basetypes are completely different?' You're right! But **xindy** offers you to define your own alphabets. For example, you can define a new alphabet by writing

```
(define-alphabet "my-personal-alphabet" ("Hi" "ho" "here" "I" "go"))
```

This is a valid alphabet that consists of 5 *letters*. You can now define a location class

```
(define-location-class "my-personal-class"
    ("my-personal-alphabet" :sep "-" "arabic-numbers"))
```

to match all of the following locations: *Hi-12, ho-2, here-709, I-9, go-42*. **xindy** will recognize them and be able to sort them according to your specification which says that *Hi* comes before all others and *here* is exactly at the third position. So they will be sorted lexicographically, layer by layer, until it can decide which one is before or after the other. We have prepared a concrete example. Do you remember the example we gave when we spoke about indexing bible verses? This exactly matches the situation of such a self-defined alphabet which could look like the following definitions:

```
(define-alphabet "bible-chapters"
    ("Genesis" "Exodus" "Leviticus" "Numbers" "Deuteronomy"
     ... ))
(define-location-class "bible-verses"
    ("bible-chapters" :sep " " "arabic-numbers" :sep "," "arabic-numbers"))
```

This description would match locations like *Genesis 1,3*, *Exodus 7,8*, etc.

Now run LaTeX on `ex2.tex` and view the results. It looks a little bit strange since **xindy** has automatically built ranges of successive locations. The first locations of the index entry *roman* actually denote the range *ii* until *iv*. *Ranges* consist of location references. To typeset them correctly you can specify

```
(markup-range :sep "--")
```

This indicates that location reference forming a range shall be separated by a hyphen. Running **xindy** and LaTeX again gives a better idea of how it should look like. Here is a part of the generated output.

```
 ...
\item appendices\quad{}A-1, A-7, A-11, B-3--B-5, C-1, C-8, C-12,
        C-13, C-22, D-2, D-3, D-5, D-10
 ...
```

3.2 Hierarchical Location Classes

Somehow a lot of space is wasted when looking at the first index entry. Modify the definition of the location class for appendices as follows and add the other commands as well:

```
(define-location-class "appendices"
                       ("ALPHA" :sep "-" "arabic-numbers")
                       :hierdepth 2)
(markup-locref-list           :sep "; " :depth 0 :class "appendices")
(markup-locref-list :open "~" :sep ", " :depth 1 :class "appendices")
```

Run **xindy** and view the output stored in `ex2.ind`. The output looks similar to the following:

```
 ...
\item appendices\quad{}A~1, 7, 11; B~3--5; C~1, 8, 12, 13, 22;
       D~2, 3, 5, 10
 ...
```

You can see that the location references of this class have been transformed into a hierarchical structure caused by the `:hierdepth` argument. Additionally we have specified markup for the layers of this class separately for the depths 0 and 1. The locations at depth 0 are separated by a ';' whereas the ones at depth 1 are separated by a ','.

Maybe you get an impression why we named **xindy** a *flexible* system.

3.3 More about Letter Groups

More problems arise when using languages with different letter schemes. Hungarian is an example. In Hungarian indexes the words beginning with the letters *Cs, Ly, Ny* and more are printed in a separate block. The words beginning with *Ly*, for example, appear behind the words beginning with an *L*. **xindy** allows to define this kind of letter groups as well. Add the following lines to the style file.

```
(define-letter-group "ly" :after "l" :before "m")
(define-letter-group "ny" :after "n" :before "o")

(markup-letter-group :open-head "~n {\bf " :close-head "}"
                     :capitalize)
```

The result looks like the following:

```
 ...
 {\bf Ly}
 \item lyuk\quad{}1
 \item lyukas\quad{}2

 \indexspace

 {\bf M}
 \item maga\quad{}1
 \item magyar\quad{}2

 \indexspace

 {\bf N}
 \item nagy\quad{}1
 \item nagyon\quad{}9
 \item nègy\quad{}4

 \indexspace

 {\bf Ny}
 \item nyelv\quad{}1
 \item nyolc\quad{}8
  ...
```

The result describes what the purpose of the above commands is. It becomes prettier from step to step, doesn't it?

You have now learned most of the features of **xindy**. Go playing around a little bit. For a detailed description of all commands and all their arguments and switches you should reference the manual that comes with this distribution.

4 References

The following books and papers were referenced in this document.

1. D. E. Knuth, *The TEXbook*, Addison-Wesley, 1984

2. Leslie Lamport, LATEX *A Document Preparation System*, Addison-Wesley, 1986

3. Joachim Schrod, *An International Version of MakeIndex*, Cahiers GUTenberg, 10, p81-90, 1991

4. Pehong Chen and Michael A. Harrison, *Automating Index Preparation*, TR-87/347, University of California Berkeley, 1987

5. Roger Kehr, **xindy**, *Ein Flexibles Indexierungssystem*, Studienarbeit FB Informatik, TH-Darmstadt, November 1995.