xindy

A Flexible Indexing System

Roger Kehr, Computer Science Department, Darmstadt University of Technology
Manual, Edition February 1998 for xindy version 2.0

This is the manual of the indexing system xindy*.* xindy *is a powerful and flexible framework for generating indexes for many kinds of documents. This is Edition February 1998 and describes the features of* xindy *version 2.0. The most up-to-date version of this document can be obtained from the* xindy *homepage at http://www.iti.informatik.tu-darmstadt.de/xindy/.*

Contents

1 About this Manual

We assume that the reader is already familiar with the task of an index processor and has basic experience in using a markup-based document preparation systems such as TeX/LaTeX [1, 2], the roff-family or SGML-based systems (e.g. its instance HTML).

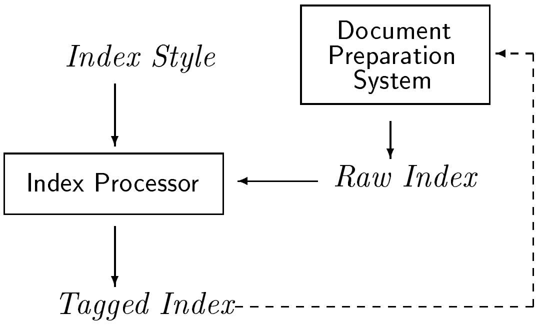
For a basic introduction into the handling and using of xindy refer to the tutorial that comes with this distribution.

This document is structured as follows. The second part gives an overview on the index model that is used by our system and we define general concepts and terms. The third part contains a list of all index style commands, their syntax, and a description of their purpose. The reader should at least have read the first part of the manual to understand the basic terminology to be able to use the command reference. The fourth part describes how xindy can be invoked, what command line options are available and more. The fifth part describes the different processing phases the system runs through.

2 How the Index Processor Works

2.1 Data Flow

Usually document preparation systems produce some form data that describe the *index entries* and the *locations* they point to. This data is called the *raw index* since it consists of raw data which contains structured information about the entries of the index. The raw index is fed into the index processor and is processed according to a specification called the *index style*. This is a user-definable description how the index is to be processed, what sort rules for the keywords should be used, which kind of locations may appear, and finally, what tags should be emitted when writing the index into the output file. The result is sometimes fed back into the document preparation system, as it is in the case of TEX, or used otherwise. The following figure illustrates this embedding into the document preparation process.



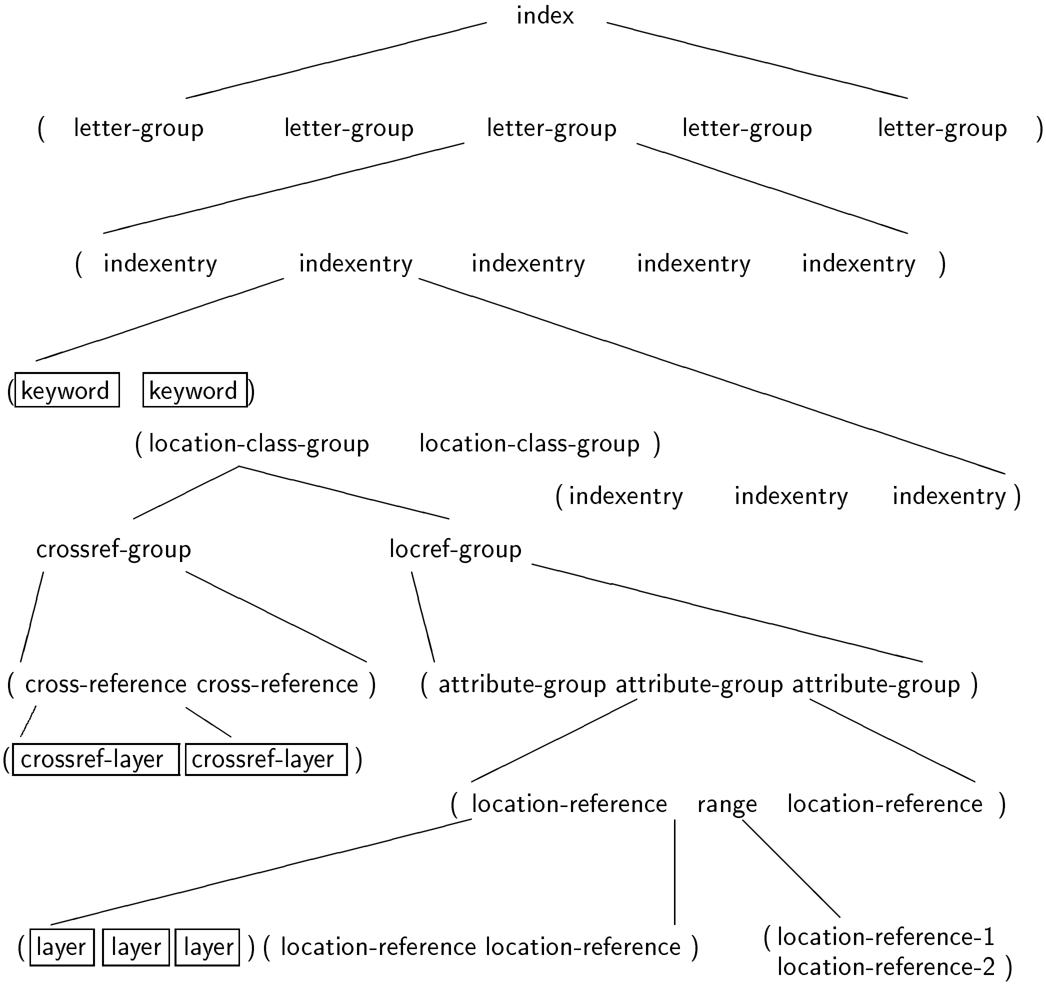
The xindy system is oriented on book-like indexes in a form that is described later in detail. The format of the input data is fixed and a small tool that converts some system dependent raw index in a form suitable for xindy should not be a big hurdle. The user is normally not involved in the correct format suitable for xindy and some tools will make this work invisible to him.

In contrast, the output of the xindy system is highly configurable and can produce a variety of different markups. Additionally, the way xindy processes the index data can be parametrized and controlled in many ways and gives the user a powerful framework for describing complex indexes. The user can tell xindy how it should produce an output according to his wishes or simply use some of the predefined styles that come with the distribution.

2.2 General Concepts and Terms

In the following sections we introduce some terms that are necessary to understand the way xindy works.

The development of xindy started with the devlopment of a model of a *book-like* index. This rather vague term will now be fleshed out by giving a semi-formal description of the model. If the structure of an index conforms to this model it can be processed by xindy. We first present a diagram that describes the structure of a processed index. It is a hierarchical recursive data structure which we will describe now.



Boxed objects contain elements from the raw index such as the keywords or the location reference layers. We start from the root of the diagram and explain the different elements.

index

An index consists of a list of letter-groups.

letter-group

The letter groups serve as containers for indexentries. The letter groups are intended to group all index entries sharing a common keyword prefix, allowing the assignment of a specific markup for each group. For example, indexes are often organized in a way that new index entries beginning with a new letter are separated by a specific markup. The following example illustrates this style.

```
B
  bread      25, 27
  butter     26

M
  marmalade  19
  milk       21
```

Here we have two letter groups for the letters 'B' and 'M' because the keywords 'bread' and 'butter' share the same prefix 'b' and 'marmalade' and 'milk' share the prefix 'm'. xindy can be advised to define arbitrary letter groups with a particular markup for each of them. See command `define-letter-group` for further details.

indexentry

An index entry consists of three components: the keyword and optionally the list of location-class-groups and the sub-indexentries. An example:

```
bread   25-30
   brown   26
   white   27
```

This index entry consists of the keyword 'bread', the location class list '25-30' and the sub-entries 'brown 26' and 'white 27'. The sub-entries themselves are complete index entries as well, forming a recursive data structure.

keyword

The keyword is the identifier of an index entry. It consists of a list of strings being the *key* if the index entry. In our model the keyword constists of a list of strings, since indexes are often organized hierarchically with different layers. In the example

```
bread          25-30
bread, brown      26
bread, white      27
```

we have keywords 'bread', 'bread, brown' and 'bread, white'.

A keyword is actually separated into four different keys. The main-key is the key that is specified from the author who indexes a document. The merge-key is derived from the main key with the so-called *merge mapping*. This mapping is accomplished with a set of rewrite rules. The sort-key is derived from the merge key with another set of rewrite rules called the *sort mapping*. The print-key can be specified by the user explicitely or, if it is ommited, equals the main key. The rewrite rules consist of string substitutions that also include regular expression features.

The following figure describes this:

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

The merge-key is necessary to define the equality of index entry keys. See command `merge-rule` for further details about defining the merge mapping.

The sort-key is necessary to define a total order on the keywords. Since we need a scheme that is independent from any specific character set such as ASCII or ISO-Latin, we must be able to define completely new alphabets. The index entries are then sorted according to the ISO-Latin alphabet. See the command `sort-rule` for an explanation how these mapping can be specified.

The print-key is used for the final markup and contains the string that appears in the generated markup.

location-class-group

There exist two kinds of location class groups namely location-reference-groups and cross-reference-groups.

location-reference-group

consists of a list of attribute-groups.

attribute-groups

Each location-reference-group consists of a list of attribute-groups. Attribute groups are defined by a set of attributes. In the markup phase, all location references are sorted according to the group number their attributes belong to. Additionally all location references of a particular attribute group appear together in the index.

See command `define-attribute-groups` for further information about the handling of attribute-groups.

An attribute group consists of a list of location-references and/or ranges.

location-reference-class, location-reference, location

Each index entry of an index references at least one location in the document. A location is a logical or physical part of a document, such as a single page, a section or subsection, etc. The symbolic name of a location is called a location-reference. The location reference is always represented as a string (a sequence of characters). Each location reference is associated a special tag called attribute. The following examples may give you an idea how location references could look like.

`"25"` represents a page number.

`"Chapter-3"` represents the third chapter of a document.

`"A-I"` represents the first page, written in uppercase roman numerals of the appendix ‘A’, separated by a hyphen.

`"2.3.4"` represents the sub-sub-section 2.3.4 of a document.

As we can see, the location references are often composed hierarchically of smaller entities, such as numbers or names. All location references that belong to the same class form a location-class. A location class is composed of alphabets and separators. See command `define-location-class` for a description how classes can be defined.

alphabet, separator

As we have seen in the previous section, location classes consist of alphabets and separators. xindy has the following built-in set of basic alphabets:

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

`roman-numerals-uppercase`
: the roman numerals I, II, III, (IIII/IV), V, ... It recognizes the old and new roman numeral system.

`roman-numerals-lowercase`
: the same for the lowercase roman numerals.

`ALPHA`
: the US-ASCII alphabet (uppercase letters).

`alpha`
: the US-ASCII alphabet (lowercase letters).

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

The user is free to define new alphabets as he wishes. See the commands `define-alphabet` and `define-enumeration` for further details.

A complete location class can be defined by composing alphabets and separators. The location class of the example `"A-I"` can be defined as follows

```
(define-location-class "appendix"
                       (ALPHA "-" roman-numerals-uppercase))
```

This is a list that completely describes all possible instances of this location class. Other valid members are `A-II`, `B-VI`, etc. Location classes are defined with the command `define-location-class`.

xindy reads the incoming strings representing location references and tries to match them against a previously defined location class. There exist no predefined location classes.

range

A range is a pair of location-references. A sequence of location references following one another is transformed into an range if the length of the sequence exceeds a certain user-definable limit. For example, the location references 11, 12, 13 and 14 may be transformed into the range 11–14. See command `define-location-class` for further details.

cross-reference-group

Container for all cross-references of a particular cross-reference-class.

cross-reference-class, cross-reference

A cross-reference is an unstructured reference. Cross-reference-classes are used to define different classes of cross references. All elements of a cross reference class appear together in the index. A cross reference consists of a list of strings.

attributes

Each location reference contains a so-called attribute. Attributes may be used to give a location reference a different meaning that may influence the processing and markup of this reference. Sometimes location references pointing to semantically different locations such as a definition or a usage of a term are tagged differently. They often appear in *italics* or **boldface** indicating this difference. An attribute may now be used to express this difference. Since xindy needs to know the possible attributes before the index entries are read in, the command `define-attributes` must be used to define a set of attributes and how they interact.

3 Command List

Here is the complete list of xindy's commands that may be used in the index style. The symbol `name` always refers to a string. We separate the commands into the *processing* and *markup* commands. The commands are listed in alphabetical order.

The parenthesis '`[`' and '`]`' denote optional parts of the syntax and '{' and '}' denote the grouping of elements. A vertical bar indicates alternatives. However, the enclosing round braces *are* part of the syntax and must be supplied.

3.1 Processing Commands

Here follows the list of processing commands.

`(define-alphabet name string-list)`

Defines `name` to be the alphabet consisting of all elements of the `string-list`. Examples:

```
(define-alphabet "example-alphabet" ("An" "Example" "Alphabet"))
```

defines an alphabet consisting of exactly three symbols. For the successor relationship holds: `succ("An")="Example"` and `succ("Example")="Alphabet"`. The built-in alphabet `digits` is defined as follows:

```
(define-alphabet "digits"
                 ("0" "1" "2" "3" "4" "5" "6" "7" "8" "9"))
```

`(define-attributes attribute-list)`

Defines all attributes the raw index may contain. Parameter `attribute-list` is a list of list of strings. The nesting level must not be more than 2. So `(..(..)..)` is allowed, whereas `(..(..(..)..)..)` is not.

The list has two kinds of elements: strings and list of strings. A single string is treated as if it were a single element list. So the lists `("definition")` and `( ("definition") )` are equivalent. All elements forming a list are a so-called *attribute group*. The members of a group are written to the output file before any member of the following groups are written.

Examples of valid attributes lists are:

`("definition" "usage")` defines two attribute groups. The first one contains all references with the attribute `definition` and the second one all with the attribute `usage`.

`(("definition" "important") "usage")` defines two attribute groups. The first one contains all references with the attributes `definition` or `important` and the second one all with the attribute `usage`. In the attribute group `("definition" "important")` the attribute `definition` overrides `important`.

```
(define-crossref-class name [:unverified])
```

Defines `name` to be a class of cross references. We distinguish two types of cross reference classes. *Verified* cross reference classes can be checked for dangling references. If for instance a cross reference points to the non-existent keyword 'foo' a warning is issued and the user is advised to correct the invalid cross reference. This is the default. If for some reasons this mechanism must be deactivated the switch `:unverified` can be used to suppress this behaviour.

```
(define-letter-group name [:before lgname] [:after lgname]
                          [:prefixes list-of-prefixes])

(define-letter-groups list-of-letter-groups)
```

This command defines a letter group with name `name`, which must be a string value, grouping all index entries that have a *sort key* beginning with the prefix `name`. The command

```
(define-letter-group "a")
```

is equivalent to the command

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

Using the latter form one can associate more than one prefix with a given letter group. Also further prefixes can be added to an already existing letter group by simply defining the same letter group again. This results not in a redefinition but in adding more prefixes to the currently defined prefixes. Example:

```
(define-letter-group "a")
```

defines a letter group containing all index entries beginning with the string `"a"`.

```
(define-letter-group "c" :after "a")
```

defines a letter group containing all index entries beginning with the string `"c"`. The letter group appears behind the letter group "a"

```
(define-letter-group "b" :after "a" :before "c")
```

inserts letter group "b" between letter group "a" and "c". This allows incremental definition of letter groups by extending already defined ones.

The arguments `:after` and `:before` define a partial order on the letter groups. `xindy` tries to convert this partial order into a total one. If this is impossible due to circular definitions, an error is reported. If more than one possible total ordering can result, it is left open which one is used, so one should always define a complete total order.

The command `define-letter-groups` (with an 's' at the end) is simply an abbreviation for a sequence of `define-letter-group` definitions where the elements are ordered in the ordering given by the list. Example:

```
(define-letter-groups ("a" "b" "c"))
```

equals the definitions

```
(define-letter-group "a")
(define-letter-group "b" :after "a")
(define-letter-group "c" :after "b")
```

See also commands `markup-letter-group-list` and `markup-letter-group` for further information.

```
(define-location-class name layer-list
                       [:min-range-length num]
                       [:hierdepth depth]
                       [:var])
```

Defines `name` to be a location class consisting of the given list of layers. A list of layers consists of names of basetypes and/or strings representing separators. Separators must follow the keyword argument `:sep`. If the keyword `:min-range-length` is specified we define the *minimum range length* to be used when building ranges. The argument `num` must be a positive integer number or the keyword `none` in which case the building of ranges is disallowed. If the switch `:var` is specified the declared class is of type *variable*, i.e. it is a *var-location-class*. Since building of ranges is currently only allowed for standard classes `:var` and `:min-range-length` must not be used together. The keyword argument `:hierdepth` can be used to declare that the location references have to be tagged in a hierarchical form. Its argument `depth` must be an integer number indicating the number of layers the hierarchy does contain. See command `markup-locref-list` for more information. Examples:

```
(define-location-class "page-numbers" ("arabic-numbers")
                       :minimum-range-length 3)
```

Defines the location class `page-numbers` consisting of one layer which is the alphabet `arabic-numbers`. Since the minimum range length is set to 3 the location references 2, 3 and 4 don't form a range because the range length is only 2. But the references 6, 7, 8, and 9 are enough to form a range. Some example instances of this class are `0`, `1`, ... `2313`, etc.

```
(define-location-class "sections" :var
   ("arabic-numbers" :sep "."
    "arabic-numbers" :sep "."
    "arabic-numbers"))
```

defines a variable location class. Valid instances are `1`, `1.1`, `1.2`, `2`, `2.4.5`, but none of `2-3` (wrong separator), `1.2.3.4` (more than 3 layers), `2.3.iv` (roman number instead of arabic one).

```
(define-location-class-order list)
```

Defines the order in which the location classes are written to the output file. The parameter `list` is a list of names of location classes. Examples:

```
(define-location-class-order
    ("page-numbers" "sections" "xrefs"))
```

tells the system that the page numbers should appear before the section numbers and that the cross references should appear at the end. If this command is omitted, the declaration order of the location classes in the index style is implicitly used as the output order. In the case that a location class does not appear in the list, the output may behave unexpectedly, so one should always enumerate all used location classes when using this command.

```
(define-rule-set name
        [ :inherit-from ("rule-set" "rule-set-2") ]
        :rules (<rule>...) )
```

A complete specification of a multi-phase sorting process for a language requires that some rules have to appear in several subsequent sorting phases. Rule sets can be used to define a set of rules that can be instantiated in an arbitrary sorting phase. Basically, they offer means to separate the definition of sorting rules from their instantiation, hence, acting as a wrapper for calls to `sort-rule`. They do not add new functionality that is not already present with `sort-rule`.

A rule can be of the form:

```
<rule> ::= ("pattern" "replacement"
            [:string|:bregexp|:egegexp] [:again])
```

The following incomplete example defines a new rule set of name `isolatin1-tolower` that inherits definitions from rule set `latin-tolower`, overriding or adding the sort rules in the list of `:rules`.

```
(define-rule-set "isolatin1-tolower"

  :inherit-from ("latin-tolower")

  :rules (("À" "à" :string :again)
          ("Á" "á" :string :again)
          ("Â" "â" :string :again)
          ("Ã" "ã" :string :again)
          ("Ä" "ä" :string :again)
          ("Å" "å" :string :again)
          ("Æ" "æ" :string :again)
       ...
         )
...)
```

Rule sets can be instantiated with the command `use-rule-set`. For further descriptions on the sorting model refer to the command `sort-rule`.

```
(define-sort-rule-orientations (orientations...))
```

Defines the order for the different sorting phases. The currently implemented *orientations* are `forward` and `backward`. This command must precede all `sort-rule` commands in an index style. It defines the orientations and implicitly sets the maximum number of sorting phases performed.

For further descriptions on the sorting model refer to the command `sort-rule`.

```
(merge-rule pattern replacement [:again]
                                [:bregexp | :eregexp | :string])
```

Defines a keyword mapping rule that can be used to generate the *merge key* from the *main key* of an index entry. This mapping is necessary to map all keywords that are differently written but belong to the same keyword to the same canonical keyword.

The parameter `pattern` can be a POSIX-compliant regular expression or an ordinary string. The implementation uses the GNU Rx regular expression library which implements the POSIX regular expressions. Regular expressions (REs) can be specified as *basic regular expressions* (BREs) or *extended regular expressions* (EREs). You can use the switch `:bregexp` to force the interpretation of the pattern as a BRE, or `:eregexp` to interpret it as an ERE. If you want `xindy` to interpret the pattern literally, use the switch `:string`. If none of these switches is selected, `xindy` uses an auto-detection mechanism to decide, if the pattern is a regular expression or not. If it recognizes the pattern as a RE, it interprets it as an ERE by default.

The parameter `replacement` must be a string possibly containing the special characters `&` (substitutes for the complete match) and `\1`,...,`\9` (substituting for the n-th submatch. Examples:

```
(merge-rule "A" "a")
```

replaces each occurrence of the uppercase letter '`A`' with its lowercase counterpart.

```
(merge-rule "\~"([AEOUaeou])" "\1")
```

transforms the TeX umlaut-letters into their stripped counterparts, such that '`\"A`' is treated as an '`A`' afterwards.

The following sequences have a special meaning:

'`~n`' : End of line symbol (*linefeed*).

'`~b`' : The ISO-Latin character with the lowest ordinal number.

'`~e`' : The ISO-Latin character with the highest ordinal number.

'`~~`' : The tilde character.

'`~"`' : The double quote character.

Tilde characters and double quotes have to be quoted themselves with a tilde character. The special characters '`~b`' and '`~e`' allow the definition of arbitrary sorting orders by rules. In connection with an additional character every position in the alphabet can be described. E.g. '`m~e`' is lexicographically placed between '`m`' and '`n`'.

Due to efficiency, rules that just exchange characters or substitute constant character sequences are not treated as regular expressions. Therefore, instead of using the rule

```
(merge-rule "[A-Z]" "&x")
```

it is more efficient (though less comfortable) to use

```
(merge-rule "A" "Ax")
(merge-rule "B" "Bx")
...
(merge-rule "Z" "Zx")
```

Usually rules are applied in order of their definition. Rules with a special prefix precede those that begin with a class of characters, so that the search pattern '`alpha`' is checked before '`.*`', but '`auto`' and '`a.*`' are checked in order of their definition.

The first rule from a style file that matches the input is applied and the process restarts behind the substituted text. If no rule could be applied, the actual character is copied from the input and the process continues with the next character.

Sometimes it is necessary to apply rules anew to the result of a transformation. By specifying the keyword argument `:again` in the merge rule the rule is marked as *mutable*, which means that after using this rule the transformation process shall restart at the same place. E.g. the rule

```
(merge-rule "\$(.*)\$" "\1" :again)
```

deletes *all* surrounding '`$`' symbols from the input.

See also command `sort-rule`.

```
(merge-to attr-from attr-to [:drop])
```

A *merge rule* says that the attribute `attr-from` can be used to build ranges in `attr-to`. Both attributes must name valid attribute names. The switch `:drop` indicates, that the original location reference with attribute `attr-from` has to be dropped (removed), if a successful range was built with location references in attribute `attr-to`. A detailed description is given in the section about processing phases.

```
(require filename)
```

This command allows to load more index style modules. The module is searched in the directories defined in the search path. The file is read in and processing of the current file continues. The argument `filename` must be a string. This allows to decompose the index style into several modules that can be included into the topmost index style file. Example:

```
(require "french/alphabet.xdy")
(require "french/sort-rules.xdy")
(require "tex/locations.xdy")
(require "tex/markup.xdy")
```

Submodules can load other submodules as well. If a file is required that was already loaded, the `require` command is simply ignored and processing continues without including this file twice. See also command `searchpath`.

```
(searchpath {path-string | path-list})
```

This command adds the given paths to the list of paths, `xindy` searches for index style files. The argument `path-string` must be a colon-separated string of directory names. If this path ends with a colon the default search path is added to the end of the path list. Example:

```
(searchpath ".:/usr/local/lib/xindy:/usr/local/lib/xindy/english:")
```

adds the specified directories to the search path. Since the last path ends with a colon, the built-in search path is added at the end. Specifying

```
(searchpath ("."
             "/usr/local/lib/xindy"
             "/usr/local/lib/xindy/english"
             :default))
```

yields exactly the same result as the example above. Here `path-list` must be a list of strings and/or the keyword(s) `:default` and `:last`. The keyword `:default` signifies that the default pathnames are to be inserted at the specified position in the list. The keyword `:last` allows to insert the currently active paths at the indicated position. Since this allows to insert the built-in paths at any position and incrementally adding new paths to the search path, this version of the command ist more flexible than the first version.

```
(sort-rule pattern replacement [:run level] [:again])
```

Defines a keyword mapping rule that can be used to generate the *sort key* of an index entry from the *merge key*. This key is used to sort the index entries lexicographically after they have been merged using the merge key.

The argument `:run` indicates that this rule is only in effect a the specified *level* (default is level 0). For a detailed discussion on the definition of sort rules for different layers refer to the documentation about the new sorting scheme (`new-sort-rules`) that comes with this distribution.

See command `merge-rule` for more information about keyword rules.

```
(use-rule-set [:run phase]
              [:rule-set ( <rule-set>... ))
```

This command instantiates the gives rule sets to be in effect at sorting phase `phase`. The order of the rule sets given with argument `:rule-set` is significant. Rule set entries of rule set appearing at the beginning of the list override entries in rule sets at the end of the list.

The following example declares that in phase 0 the rule sets `din5007` and `isolatin1-tolower` should be active, whereas in phase 2 the other rule sets have to be applied.

```
(use-rule-set :run 0
              :rule-set ("din5007" "isolatin1-tolower"))

(use-rule-set :run 1
              :rule-set ("resolve-umlauts"
                         "resolve-sharp-s"
                         "isolatin1-tolower"
                         ))
```

For a discussion on rule sets refer to command `define-rule-set`.

3.2 Markup Commands

The following commands can be used to define the markup of the index. They don't have any influence on the indexing process. Since the markup scheme is characterized by the concept of *environments*, the syntax and naming scheme of all commands follows a simple structure.

The commands can be separated into *environment* and *list-environment* commands. All commands of the first group support the keyword arguments `:open` and `:close`, whereas the second group additionally supports the keyword argument `:sep`. If one of these keyword arguments is missing, the default markup tag is *always* the empty tag. The `:open` tag is always printed before the object itself and the `:close` tag is always printed after the object has been printed. If a list is printed the `:sep`tag is printed between two elements of the list but not before the first element, or after the last one. All commands dealing with a list have the suffix '`-list`' as part of their command name.

Since the number of commands and the heavy usage of *default* and *specialized* tags makes the markup somehow complex (but very powerful) we have added a mechanism to trace the markup tags xindy omits during its markup phase with the command `markup-trace`.

Here follows the list of markup commands in alphabetical order with some of the commands grouped together.

```
(markup-attribute-group-list [:open string] [:close string]
                             [:sep string])

(markup-attribute-group      [:open string] [:close string]
                             [:group group-num])
```

Location class groups consist of lists of attribute groups. The markup of this list can be defined with the command `markup-attribute-group-list`.

To allow different markup for different attribute groups the command `markup-attribute-group` can be specialized on the group number with the keyword argument `:group` which must be an integer number. E.g., given are the groups `("definition" "theorem")` and `("default")` with group numbers 0 and 1, then

```
(markup-attribute-group :open "<group0>" :close "</group0>"
                        :group 0)

(markup-attribute-group :open "<group1>" :close "</group1>"
                        :group 1)
```

can be used to assign different markup for both groups in a SGML-based language.

```
(markup-crossref-list       [:open string] [:close string]
                            [:sep string]
                            [:class crossref-class])

(markup-crossref-layer-list [:open string] [:close string]
                            [:sep string]
                            [:class crossref-class])

(markup-crossref-layer      [:open string] [:close string]
                            [:class crossref-class])
```

A crossref class group contains cross references of the same class. The separator between the classes is defined with the `(markup-locclass-list :sep)`-parameter. A list of cross references can be tagged with the command `markup-crossref-list` that specializes on the `:class` argument.

Each cross reference is determined by a list of layers indicating the target of the cross reference. To define a suitable markup for such a list the command `markup-crossref-layer-list` can be used.

Each layer of a cross reference can be assigned two tags that specialize on the class of the reference, like all other commands.

A suitable markup for a cross reference class `see` within LaTeX2e could look like that:

```
(markup-crossref-list :class "see" :open "\emph{see} "
                                   :sep  "; ")
(markup-crossref-layer-list :class "see" :sep ",")
(markup-crossref-layer :class "see"
                                   :open "\textbf{" :close "}")
```

An example output could look like

... *see* **house**; **garden**,**winter**; **greenhouse**

```
(markup-index [:open string] [:close string]
              [ :flat | :tree | :hierdepth depth ])
```

Defines the markup tags that enclose the whole index via the `:open` and `:close` parameters. Examples:

```
(markup-index :open  "Here comes the index~n"
              :close "That's all folks!~n")
```

defines that the `:open` string is printed before the rest of the index and the `:close` string appears after the index is printed.

Additionally one can specify the form of the generated index. It is possible to produce flat indexes by specifying the switch `:flat`, to generate a tree with the `:tree` switch or any kind of mixture between both by specifying the depth up to which trees shall be built with the parameter `:hierdepth`. Its argument `depth` is the number of layers that can be formed into a tree. Therefore `:flat` is an abbrevation of `:hierdepth 0` and `:tree` is an abbrevation of `:hierdepth max-depth`, with `max-depth` being the maximum number of layers a keyword has. An example: the keywords

```
("tree" "binary" "AVL")
("tree" "binary" "natural")
```

can be transformed in the following ways:

A flat index (`:flat` or `:hierdepth 0`)

```
tree binary AVL
tree binary natural
```

with `:hierdepth 1`

```
tree
   binary  AVL
   binary  natural
```

and a tree (`:tree` or `:hierdepth` > 1)

```
tree
   binary
      AVL
      natural
```

Most often one will create tree-like indexes or ones that are flat.

```
(markup-indexentry-list [:open string] [:close string]
                        [:sep string]  [:depth integer])

(markup-indexentry      [:open string] [:close string]
                        [:depth integer])
```

Letter groups consists of a list of index entries. The command `markup-indexentry-list` defines the markup of these lists. The markup can be specialized on the depth if the index is hierarchically organized. The command

```
(markup-indexentry-list :open  "\begin{IdxentList}"
                        :close "\end{IdxentList}"
                        :sep   "~n")
```

defines that the index entries of all layers are wrapped into the given markup tags. If additionally

```
(markup-indexentry-list :open  "\begin{IdxentListII}"
                        :close "\end{IdxentListII}"
                        :sep   "~n"
                        :depth 2)
```

is defined, all index entry lists of all layers (except layer 2) are tagged according to the first specification, and the index entry list within depth 2 are tagged according to the second rule.

The command `markup-indexentry` defines the markup of an index entry at a given depth. Since index entries may also contain subentries and the markup for subentries may be different in different layers, the optional keyword argument `:depth` can be used to assign different markup for different layers. If depth is ommited the default markup for all possible depths is defined. The top-most index entries have depth 0.

```
(markup-indexentry :open  "\begin{Indexentry}"
                   :close "\end{Indexentry}")
```

defines that the index entries of all layers are wrapped into the given markup tags. If additionally

```
(markup-indexentry :open  "\begin{IndexentryII}"
                   :close "\end{IndexentryII}"
                   :depth 2)
```

is defined, all index entries of all layers (except layer 2) are tagged according to the first specification, and the index entries with depth 2 are tagged according to the second rule.

```
(markup-keyword-list [:open string] [:close string]
                     [:sep string] [:depth integer])

(markup-keyword      [:open string] [:close string]
                     [:depth integer])
```

The print key of an index entry consists of a list of strings. The markup of this list can be defined with the command `markup-keyword-list`. The keyword argument `:depth` may be specified to define the markup of the list at a particular depth.

The keyword of an index entry consists of a list of strings. Each of these components is tagged with the strings defined with the command `markup-keyword`. Since we maybe need different markup for different layers, the optional keyword argument can be used to specialize this markup for some depth.

```
(markup-letter-group-list [:open string] [:close string]
                          [:sep string])

(markup-letter-group  [:open string] [:close string] [:group group-name]
                      [:open-head string] [:close-head string]
                      [:upcase | :downcase | :capitalize])
```

The first command defines the markup of the letter group with name `group-name`. Since the markup of letter groups often contains the name of the letter group as a part of it, the other keyword arguments allow an additional markup for this group name. If one of the parameters `:open-head` and `:close-head` is specified additional markup is added as can be described as follows:

```
<OPEN>
   IF (:open-head OR :close-head)
     <OPEN-HEAD>
       transformer-of(<GROUP-NAME>)
     <CLOSE-HEAD>
   FI
   <INDEXENTRIES...>
<CLOSE>
```

Here, `transformer-of` is a function that possibly transforms the string representing the group name into another string. The transformers we currently support can be specified with the switches `:upcase`, `:downcase` and `:capitalize` which result in the corresponding string conversions. If none of them is specified no transformation is done at all.

The command `markup-letter-group` defines the markup of the list of letter groups.

```
(markup-locclass-list [:open string] [:close string]
                      [:sep string])
```

Each index entry contains a list of location class groups. This markup command can be used to define the markup of this list.

```
(markup-locref [:open string] [:close string]
               [:class locref-class]
               [:attr  attribute]
               [:depth integer])
```

The markup tags of a location reference can be specialized on the three arguments `:class`, `:attr` and additionally, if sub-references are used, `:depth`. Most often one will only use a tag depending on the attribute. For example, all location references with the attribute `definition` should appear in a font series like bold, emphasizing the importance of this location reference; those with the attribute `default` in font shape italic. The markup in this case would not specialize on the depth or any particular class. A valid definition, suitable for a usage within HTML, could look like this.

```
(markup-locref :open "<B>" :close "</B>" :attr "definition")
(markup-locref :open "<I>" :close "</I>" :attr "default")
```

```
(markup-locref-class [:open string] [:close string]
                     [:class locref-class])
```

All location references of a particular location reference class can be wrapped into the tags defined by this command. It specializes on the keyword argument `:class`.

```
(markup-locref-layer      [:open string] [:close string]
                          [:depth integer] [:layer integer]
                          [:class locref-class])

(markup-locref-layer-list [:open string] [:close string]
                          [:sep string]
                          [:depth integer]
                          [:class locref-class])
```

A location reference contains a list of location reference layers. The second markup command can be used to markup this list. It specializes on the class of the location references and the depth (if sub-references are used).

The first command allows to tag the elements of a layer list differently. The first element of this list can is specialisable with `:layer 0`, the next element with `:layer 1`, etc. See the next example for an example.

```
(markup-locref-list [:open string] [:close string] [:sep string]
                    [:depth integer] [:class locref-class])
```

An attribute group contains a list of location references and/or ranges. Additionally a layered location reference itself may contain sub-references that are stored as a list of location references. We specialize the markup for these lists on the location class they belong to with the keyword argument `:class`, and on `:depth` that specializes on the different subentry levels when using location references with sub-references.

Given is a list of location references that have the class description

```
(define-location-class "Appendix"
                       ("ALPHA" :sep "-" "arabic-numbers")
                       :hierdepth 2)
```

This location class has instances like `A-1`, `B-5`, etc. The keyword argument `:hierdepth 2` informs xindy to markup these location references in a hierarchical form. With the commands

```
(markup-locref-list            :sep "; "
                     :depth 0  :class "Appendix")
(markup-locref-list  :open " " :sep ","
                     :depth 1  :class "Appendix")
(markup-locref-layer :open "{\bf " :close "}" :layer 0
                     :depth 0  :class "Appendix")
```

we obtain a markup sequence for some example data that could look like

```
{\bf A} 1,2,5; {\bf B} 5,6,9; {\bf D} 1,5,8; ...
```

```
(markup-range [:open string] [:close string] [:sep string]
              [:class locref-class]
              [:length num] [:ignore-end])
```

A range consists of two location references. Markup can be specified with the `:open` and `:close` arguments and one separator given by the argument `:sep`.

Since both location references are tagged with markup defined by the command `markup-locref` a specialization on attributes or depth is not necessary. Specialization is allowed on the class they belong to, because the separator between two location refences may be different for each location class. Argument `:length` can be used to define different markup for different lengths. In conjunction with `:length` is may be useful not to print the second location reference at all. For example, one wishes to markup ranges of length 1 in the form *Xf.* instead of X–Y. This can be accomplished with the switch `:ignore-end`.

The markup tags for a range *(X,Y)* can be described as follows:

```
<OPEN>
  Markup of location reference X
<SEP>
  IF (not :ignore-end)
     Markup of location reference Y
  FI
<CLOSE>
```

The following tags can be used to define a range of page numbers (given in a location class `page-numbers`) without considering the open and close parameters:

```
(markup-range :sep "-" :class "page-numbers")
```

Location ranges then appear separated by a hyphen in a form like this:

```
 ..., 5-8, 19-23, ...
```

```
(markup-trace [:on] [:open string] [:close string])
```

This command can be used to activate the tracing of all markup commands `xindy` executes. The switch `:on` activates the trace. If `:on` is omitted, the command line flag `-t` can be used as well. All tags which are emitted but not yet defined explicitly by the user are tagged with a symbolic notation indicating the commands that must be used to define this tag. The defaults for the keyword argument `:open` is '`<`' and for `:close` is '`>`'. The beginning of an example output could look like:

```
<INDEX:OPEN>
  <LETTER-GROUP-LIST:OPEN>
    <LETTER-GROUP:OPEN ["a"]>
      <INDEXENTRY-LIST:OPEN [0]>
        <INDEXENTRY:OPEN [0]>
          <KEYWORD-LIST:OPEN [0]>
            <KEYWORD:OPEN [0]>
 ...
```

We use a simple indentation scheme to make the structure of the tags visible. The symbolic tag `<LETTER-GROUP:OPEN ["a"]>` for example indicates that the tag that can be specified with the command

```
(markup-letter-group :open "XXX" :group "a" ... )
```

is emitted at this point in the markup process. By incrementally adding markup commands to the index, more and more tags can be defined until the whole markup is defined. This general mechanism should allow everyone understand the markup process. The best is to start with a small index, define the complete markup and afterwards process the whole index. Additionally one can enclose the symbolic tags into an environment that is neutral to the document preparation system, such as a comment. For TEX this could be

```
(markup-trace :open "%%" :close "~n")
```

or a definition in the TEX document like

```
\def\ignore#1{}
```

combined with the command

```
(markup-trace :open "\ignore{" :close "}")
```

3.3 Raw Index Interface

This section can be skipped if the reader is not interested in adapting xindy to a new document preparation system.

The raw index is the file that represents the index that is to be processed. Since many different document preparation systems may use different forms of index representations, their output must be transformed in a form readable by xindy. We also could have written an configurable parser performing this task, but usually a tool written with some text processing tools such as `perl`, `sed` or `awk` can achieve the same task as well. Therefore, adapting xindy to a completely different system can mostly be done by writing an appropriate raw index filter.

The format of the raw index interface of xindy is defined as follows:

```
(indexentry { :key string-list [:print string-list]
            | :tkey list-of-layers }
            [:attr string]
            { :locref string  [:open-range | :close-range]
            | :xref string-list } )
```

The pseudo variable *string* is a sequence of characters surrounded by double quotes, e.g.

```
"Hi, it's me"  "one"  "a string with two \"double quotes\""
```

are three examples of valid strings. If you need to include a double quote as a literal character, you must quote it itself with a backslash as shown in the third example. A *string list* is simply a list of strings separated by whitespaces and surrounded by round braces. An example of a string list is

```
("This" "is" "a" "list" "of" "strings")
```

So far about the syntax. The semantics of the different elements are described here.

`:key`

The argument *string list* defines the keyword of the index entry. It must be a list of strings, since the keyword may consist of different layers such as `("heap" "fibonacci")`.

`:print`

The optional *print key* defines the way the keyword has to be printed in the markup phase.

`:tkey`

Another possibility to define the keys of an index entry is with the `:tkey` keyword argument. It can be used instead of the `:key` and `:print` arguments. Instead of specifying separately the key and the corresponding print key, we define the keyword by its layers. Each layer consist of a list of one or two strings. The first string will be interpreted as the main key, whereas the second one will become the print key. If the print key is ommited, the main key is taken instead. So the definition

```
:tkey (("This") ("is") ("a") ("bang" "BANG !!!"))
```

is equivalent to

```
:key   ("This" "is" "a" "bang")
:print ("This" "is" "a" "BANG !!!")
```

`:locref`

The reference an index entry describes can be a *location reference* or a *cross reference*. The switch `:locref` describes a location reference. Its optional arguments are `:open-range` and `:close-range`. The *string* that must be supplied must somehow encode the location reference. It might look like the string `"25"` representing the page number 25, or `"Appendix-I"` representing the first appendix numbered in uppercase roman numerals.

`:open-range`,`:close-range`

These are switches that do not take any arguments. They describe the beginning and ending of a *range*, starting or ending from the location reference that is given by the argument `:locref`. If they are supplied, the location reference may have influence on the way ranges are build.

`:xref`

These arguments choose the second alternative. The argument *string list* of parameter `:xref` describes where the index entry should point to.

`:attr`

This parameter may be used to tag a location reference with a certain attribute or it names the class of a cross reference. It may also used to associate different markup for different attributes in the markup phase. If this parameter is omitted or is the empty string, the indexentry is declared to have the attribute `default`.

Some examples:

```
(indexentry :key ("airplane") :locref "25" :attr "default")
```

defines an index entry with the key 'airplane' indexed on page '25'. This index entry has the attribute `default`.

```
(indexentry :key ("house") :xref ("building") :attr "see")
```

defines a cross reference with the key 'house' pointing to the term 'building'. This cross reference belongs to the cross reference class `see`.

```
(indexentry :key ("house") :xref ("building") :open-range)
```

is an invalid specification, since `:open-range` mustn't be used together with cross references.

4 Processing Phases

4.1 The Startup Phase

After the system is started, xindy reads in the index style that is passed as a command line argument. Each `require` command is executed and the internal data structures representing the index style are built up. The index style must completely be read in before the raw index can be read.

4.2 The Processing Phase

The processing phase starts with reading the complete raw index. The name of the raw index file must be passed via the command line. All index entries are read in and pre-processed. All attributes and cross reference classes are checked if they are already known to the system. All strings representing location references are matched against all known location classes. Appropriate warnings are issued, if errors are encountered.

After the raw index has completely been read in, the location references of each index entry are merged, separated and sorted and the building of ranges takes place. This phase is the most complex one and we will describe it in detail.

1. All location references are separated according to the class they belong to. These groups are called *location class groups*. Possible groups are all defined location and crossref classes. See the commands `define-location-class` and `define-crossref-class` for a description how these classes can be defined.

   The classes are sorted according to an order that can be defined with the command `define-location-class-order`.

2. The further processing of each location class group is different for the location classes and the crossref classes.

   - Cross references are sorted lexicographically based on the ISO-Latin alphabet.
   - To illustrate the processing of location references we assume the following list:

     13, *14*, *15*, *18*, **12**, **13**, **14**, **16**, 14, 16

     The location references in italics own the attribute 'important, those with in boldface have attribute 'definition', and all others are own the attribute 'default'. Imagine, the attribute groups were defined with the commands

     ```
     (define-attribute-groups (("definition" "important")
                               ("default")))

     (merge-to "definition" "default" :drop)
     ```

     See commands `define-attributes` and `merge-to` for a detailed description.

     The substitution rules are applied. This means that location references **13** and **14** with attribute 'important' are *substituted* by the location references *13* and *14* with attribute 'definition'. Substitution means removing from the list of location references.

     Substitution occurs because the definition of the attribute groups implicitly defines `"definition"` *substitutes* `"important"`.

     The resulting list is now

     13, *14*, *15*, *18*, **12**, **16**, 14, 16

     The `merge-to` rules are applied. Their meaning is to make location references appear with another attribute as well, but only in the function of supporting the building of ranges. They disappear after the ranges are built. The location references that cause new location refererences to be added are called *parents*, whereas the new ones are called *childs*. The example rule results in the adding of all refernces with attribute 'definition' to the attribute 'default' which results in the list

     13, *14*, *15*, *18*, **12**, **16**, (13), 14, (15), 16, (18)

     The childs are put in parenthesis since they may only be used to build up ranges.

     For each attribute we now try to build ranges. Since the switch `:drop` was specified we must start with the attribute 'default', because a successful merging of location references may result in dropping the parents. This results in the range '13–16'. The childs (13) and (15) were used in the building of ranges, so their parents *13* and *15* have to be removed from the list of location references. This step would be omitted if the switch `:drop` were not specified. After unsucessfully trying to build more ranges and dropping the location references *13*, *15* and (18)–which was only meant to build ranges–we obtain the list

     14, *18*, **12**, **16**, 13–16

     Finally the attributes are brought into the right order. In our example the location references of the first attribute group are merged and sorted lexicographically resulting in two attribute groups

     (**12**, *14*, **16**, *18*) (13–16)

After all index entries have been processed the letter groups are formed and the index entries and location references are transformed into tree like structures as defined in the index style.

4.3 The Markup Phase

After the index has completely been processed, the markup phase traverses the tree-like structure of the index. Each step triggers the appropriate markup events resulting in the emitting of markup tags. This phase can be traced by using the command line option `-t`.

5 Invoking xindy

5.1 Command Line Options

The following command line options are accepted:

```
xindy  [-h] [-t] [-v] [-l logfile] [-o outfile]
       [-L n] [-f filterprog]
       indexstyle raw-index
```

The argument `indexstyle` names a file, containing the index style description. The argument `raw-index` names a file, containing the raw index. Both arguments are mandatory.

`-h`
: Gives a short summary of all command line options.

`-l`
: Writes helpful information into the specified `logfile`. For example, the keyword mappings are written into this file, so one can check if the intended mappings were actually performed this way.

`-o`
: Explicitly defines the name of the `output` file. If not given, the name of the `raw-index` is used with its extension changed to '`.ind`' (or added, if it had no extension at all).

`-t`
: Enters tracing mode of the symbolic markup tags. The format of the emitted tags can be defined with the command `markup-trace`.

`-L`
: Set the xindy logging-level to n.

`-f`
: Run `filterprog` on `raw-index` before reading. The program must act as a filter reading from stdin and writing to stdout. The most obvious use of this option in conjunction with TeX is to run `-f tex2xindy` on the index file prior to reading the entries into xindy.

`-v`
: Shows the version number of xindy.

Errors and warnings are reported to `stdout` and additionally to the logfile if `-l` was specified.

5.2 Search Path

The system uses the concept of a *search path* for finding the index style files and modules. The searchpath can be set with the environment variable `XINDY_SEARCHPATH` which must contain a list of colon-separated directories. If it ends with a colon, the built-in searchpath is added to the entire searchpath. See the command `searchpath` for further details.

6 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.
6. Joachim Schrod and Gabor Herr, *MakeIndex Version 3.0*, Technical University Darmstadt, 1991
7. Roger Kehr, xindy – *A Flexible Indexing System*, Proceedings of the EuroTEX'98, St. Malo, France, March 1998