Librarian

Paul Isambert
zappathustra@free.fr

Introduction

Librarian is a set of macros that handles bibliographic references without BibTEX. It reads bibliographic files, extracts information about entries, and sorts lists of entries according to that information and the user's specifications. TEX code can then be written to typeset citations and bibliographies or more generally to write the equivalent of `bst` files, and absolutely no knowledge of the BibTEX language is needed, nor is BibTEX itself.

The package is low-level in the sense that it doesn't do much more than what's said above and offers little syntactic sugar to typeset bibliographies. It is meant to make as few meaningful decisions as possible, where 'meaningful decisions' means operations that influence the output or what the user can do. The way *librarian* extracts information is immaterial (barring efficiency, of course) to the typesetting of a bibliography, which depends strictly on the user's knowledge in TEX programming. On the contrary, predefined styles, for instance, would be easier indeed but also more limited.

Besides, *librarian* is format-independent, relying on plain TEX's usual allocation macros only (which are similar in other formats, as far as I can tell). Thus, a well-written bibliographic style should work equally well in all formats; by 'well written' I mean that it does not use any format-specific commands, and at worst defines aliases to those commands at the beginning of the file.

The first part of this document is a reference manual where *librarian* is explained thoroughly. The second part spells out a pair of examples of bibliographic styles; the second example gives details on the important parts of the file `authoryear.tex`, which is distributed with the package and can be used as is.

Reference manual

How to load librarian

In plain, the package is loaded in the good old way:

```
\input librarian
```

In LaTEX you must use the associated style file:

```
\usepackage{librarian}
```

And in ConTEXt you must load *librarian* with the third-party file:

```
\usemodule[librarian]
```

Commands

`\Cite{<entry key>}{<list>}{<code1>}{<code2>}`

This adds *<entry key>* to *<list>* (if the latter doesn't exist, it is created on the fly), unless it has already been added previously. If *<entry key>* has already been read in the bibliography (most likely in a previous compilation), `\EntryKey` is set to *<entry key>* and *<code1>* is executed. Otherwise, *librarian* will read the bibliographic files to retrieve information about *<entrykey>*, and for the moment *<code2>* is executed (without redefining `\EntryKey`). With complex bibliographic styles, this command is better embedded in macros.

`\BibFile{<list of files>}`

The *<list of files>* is a comma-separated list of bibliographic files in which *librarian* will dig information about the requested entries. The `.bib` extension is implicit, but you can give another one, as in `\BibFiles{mybib,myotherbib.tex}`. The information is passed to the next compilation thanks to an external file called *<jobname>*`.lbr` (the culprit to delete if things seem wrong to you). If there remains no requested entry then the files aren't read. `\BibFile` can occur several times in a document, but on each call, only those entries that have been `\Cite`'d before will be searched for in the bibliographic files, because calls to `\Cite` aren't passed from one compilation to the next. Most complex bibliographic styles generally require three compilations to get everything right.

`\SortingOrder{<list of fields>}{<name parts>}`

This sets the order in which lists will be sorted. The fields in *<list of fields>* are the ones you find in BibTEX entries, plus the ones *librarian* adds to entries (like `name`), and they should be separated by commas. Any field can be prefixed with '-', which indicates that when it comes to this field entries will be sorted in reverse

order. The sorting works as follows: when comparing two entries, *librarian* compares their values for the first field; if they are the same, it tries the second field, and so on. If *<list of fields>* is exhausted, the entries are sorted according to the order of citation in the document relative to the list that's being scanned (i.e. the value of `\EntryNumber`). The *<name parts>* are concatenated letters, with 'f' denoting first name, 'l' denoting last name, 'v' referring to the von part and 'j' referring to the junior part; it sets how names are compared, and although there seems to be only two meaningful orders (`vlfj` and `lfvj`), differing in how the von part is taken into account, you can do whatever you want. Thus, `\SortingOrder{name,year,title}{vlfj}` sorts entries in the 'author year' style, with titles differentiating entries by the same author(s) written the same year, and with the von part taking precedence in the last name. Instead, `\SortingOrder{name,-year}{lfvj}` will produce a bibliography sorted by names, with most recent references first, and similar references sorted according to their order in the document, while the von part doesn't matter in last names (and thus Ludwig van Beethoven is sorted before Franz Schubert). In any case, *<name parts>* have nothing to do with how names are typeset in the bibliography. (In these examples, `name` is used instead of `author` or `editor` on purpose. See below about the `name` field.) By default, *librarian* doesn't sort entries.

`\SortList{`*`<list>`*`}`
This sorts *<list>* according to the latest value of `\SortingOrder`. Thus different lists can be sorted with different orders. If `\SortList` is called before `\BibFile`, then new entries will be sorted on the next compilation only. From one compilation to the next, *librarian* retains the previous sorted entries and sorts only new entries, and entries that are equal according to `\SortingOrder` (because their relative order may have changed). This saves time (especially in LuaTEX where the many calls to `\directlua` are quite time-consuming). However, if the value of `\SortingOrder` when `\SortList` is called has changed from the previous compilation, all entries are sorted again.

`\ReadList{`*`<list>`*`}`
This loops over entries in *<list>*, and for each it stores its key in `\EntryKey` and calls `\MakeReference`, which by default does nothing, and should be redefined to typeset the entry. For each entry you can use the `\ifequalentry` conditional, which is true if the current entry is part of a set of entries that are equal with regard to `\SortingOrder` and are thus distinguished by citation order only, and false otherwise. If the list is unsorted (i.e. `\SortingOrder` is empty), then `\ifequalentry` is always false, even though all entries are equal.

`\SortDef{`*`<command>`*`}{`*`<definition>`*`}`
This defines *<command>* as *<definition>* at sorting time. Indeed, macros shouldn't hinder sorting. However, `\TeX` won't be read as 'TeX' and *TEXbook* won't be sorted after *Tao Te Ching* but rather after *Typographic Investigations*, because macros separate letters. So `\SortDef\TeX{tex}` is the proper way to get good sorting (everything is sorted in lowercase, so no need to use 'TeX'). `\SortDef` can't define with parameter text, but a macro can be defined to another one taking arguments, although it is generally unnecessary: all braces are removed at sorting time. It is probably very important to redefine accents (*librarian* doesn't), e.g. `\SortDef\'{}` is vital if you want to see Bézier between Bernoulli and Boole.

`\WriteInfo{`*`<arg>`*`}`
`\WriteImmediateInfo{`*`<arg>`*`}`
Since *librarian* reads and rewrites an external file on each compilation, you can use it too. This writes *<arg>* to that file, so that it is made available for the next compilation; *<arg>* is expanded as in any `\write` statement. `\WriteInfo` writes at output time, whereas `\WriteImmediateInfo` writes immediately.

`\Preamble`
This holds everything that has been encountered in `@preamble` entries when reading bibliographies. It isn't passed from one compilation to the next.

`\CreateField{`*`<field>`*`}`
If a field is unknown to *librarian* (there is no restriction in bibliographic entries), *librarian* complains when asked to retrieve it with the commands below. This command just enables it to work with unknown fields. By default, *librarian* knows the common BibTEX fields and its own, of course.

\AbbreviateFirstname
This turns first names into initials, e.g. *John* into *J.* and *Jean-Luc* into *J.-L.* In order to get *Ch.* from *Charles*, the name in the bibliographic file should be '{Ch}arles', which is already the case for BibTEX. Note that *librarian* sorts full names and thus *J(ohn) Doe* and *J(ack) Doe* are different in this respect, even though they'll be indistinguishable in your bibliography.

The following macros all come in pairs made of \Command and \CommandFor, where \Command is \CommandFor with *<entry key>* set to \EntryKey. All along, I refer to bibliographic entries, naturally, and *field* and *value* are to be understood in that context (see the next two sections for details).

\RetrieveField{*<field>*}
\RetrieveFieldFor{*<field>*}{*<entry key>*}
This produces the value of *<field>* for *<entry key>*. The command uses \lowercase, so you should use it with care, and basically avoid it unless for typesetting fields. The following command should be preferred if anything must be manipulated.

\RetrieveFieldIn{*<field>*}{*<command>*}
\RetrieveFieldInFor{*<field>*}{*<entry key>*}{*<command>*}
This one defines *<command>* as the value of *<field>* for *<entry key>*. If *<field>* doesn't exist in *<entry key>*, or else if *<entry key>* doesn't exist, then *<command>* is defined as an empty command and is equivalent to \empty if the latter is defined as \def\empty{}. Quite convenient for testing.

\EntryNumber{*<list>*}
\EntryNumberFor{*<entry key>*}{*<list>*}
This returns *n* if *<entry key>* is the *n*th entry in *<list>*, i.e. if it is the *n*th entry to be cited for the first time in *<list>*.

\EntryNumberIn{*<list>*}{*<command>*}
\EntryNumberInFor{*<entry key>*}{*<list>*}{*<command>*}
This defines *<command>* as *n*, where *n* is defined as above. (Beware that *<command>* is a macro whose expansion produces a number, not an integer denotation.) This is safer than the previous command for the same reason as with \RetrieveFieldFor, and is emptily defined under the same circumstances.

\ReadName{*<code>*}
\ReadNameFor{*<entry key>*}{*<code>*}
For each person in the name field for *<entry key>*, this stores the first name in \Firstname (abbreviated if \AbbreviateFirstname was called), the last name in \Lastname, the von part in \Von and the junior part in \Junior. There is also a number \NameCount, which holds the place of the person in the name field (i.e. the first author has \NameCount=1). Finally, *<code>* is called with those values. There are aliases \ReadNames and \ReadNamesFor. \ReadAuthor, \ReadAuthorFor, \ReadEditor and \ReadEditorFor, and the versions with plural as well, are defined equivalently for the author and editor fields respectively. You can still use \RetrieveField{author}, for instance, but those fields aren't designed for typesetting in their basic form, hence this command.

\TypesetField{*<field>*}{*<command>*}{*<code>*}
\TypesetFieldFor{*<field>*}{*<entry key>*}{*<command>*}{*<code>*}
If *<field>* isn't empty in *<entry key>*, *<command>* is executed with the value of *<field>* as its argument. So *<command>* should be a macro that takes one (and only one) argument. If *<field>* is empty, *<code>* is executed instead.

Fields

Beside the fields that are found in bibliographic entries, *librarian* adds some of its own, which can be convenient for dealing with references. Here they are.
The entrytype field contains the type of the entry as recorded in bib files at the beginning of each entry after the @ sign. Thus

```
@Book{Coover2002,
  ...
}
```

has 'book' as the value for entrytype. Note that lowercase is always used, whatever the actual case in the bib file. The number of authors in the author field is the value of the authornumber field, each author being distinguished from the next by the customary 'and'. An equivalent field, editornumber, does the same for editor. Most importantly, the name field holds the same value as author if there is such a field in the entry, or the value of editor it it exists and there is no author. Thus edited books can be cited along with normal books.

That's why name appeared in \SortingOrder above: if it had been author, then collective works wouldn't have been sorted correctly. Along with name, namenumber holds the value either of authornumber or of editornumber, according to the same rule. Finally, nametype takes one out of two values: author or editor. It takes the former if the name field is set to the value of the author field, and the latter if it is set to the value of the editor field. Thus, when referencing collective works in a bibliography, you can add '(ed.)' for instance after the value of name if nametype is editor.

A crash course in bibliographic entries (and how librarian reads them)

This section is not necessary to the understanding of *librarian* but it can be useful if in trouble. Suppose you have the following entry:

```
@Book{Coover2002,
  author = {Coover, Robert},
  title = {The Adventures of Lucky Pierre},
  subtitle = {Director's Cut},
  publisher = "Grove Press",
  Address = {New York},
  year = 2002,
}
```

Then *librarian* stores all the fields for the 'coover2002' entry key. This key is in lowercase, and thus you can't distinguish two entries by case only (this is how BibTEX works). This doesn't mean that you can't use \Cite{Coover2002}{list}: *librarian* does the necessary conversion. The entrytype field is set to 'book', in lowercase once again, and this time you must be careful: 'book' is a value, and since you'll probably test it to know how the reference should be typeset in the bibliography, you shouldn't compare it to, say, 'Book', of course. And to finish with lowercase, 'New York' is the value of the address field, not Address, but once again *librarian* does the conversion so you can call \RetrieveField{Address}{cooVEr2002}. Finally, the subtitle field is no customary field, but you can use it. Up to there, *librarian* works exactly like BibTEX.

But there are differences too. For one thing, *librarian* has no restriction on the type of the entry, so you could have:

```
@novel{Coover2002,
  author = {Coover, Robert},
  title = {The Adventures of Lucky Pierre},
  subtitle = {Director's Cut},
  publisher = "Grove Press",
  Address = {New York},
  year = 2002,
}
```

That's perfectly fine with *librarian*, although not with BibTEX. There are no restrictions on fields either, and unlike BibTEX fields aren't labelled as required or optional or ignored; in a bibliographic style you can issue an error message if a field is missing, but *librarian* won't do it for you. And there are annoying differences too. They occur in the way *librarian* reads the values of fields. Here's how it works: if the first character (disregarding space) after the '=' sign is a {, then everything up to the next balanced } is taken as the value, and everything up to the comma is discarded. (The last value of the entry may have no comma, but *librarian* adds one before reading the entry.) If the first character is a ", then everything up to the next ", with balanced braces, is taken as the value, and everything up to the comma is discarded once again. Finally, if the first character is none of the above, then *librarian* takes everything up to the next comma as the value (while balancing braces, of course). The latter case differs from BibTEX. Normally, only numerical values (like the value of year in our example) can be given as such, unless strings are used. But *librarian* is totally unable to understand strings. The following

```
@string{ny = "New York"}
@Book{Coover2002,
  author = {Coover, Robert},
  title = {The Adventures of Lucky Pierre},
  subtitle = {Director's Cut},
  publisher = "Grove Press",
  Address = ny,
  year = 2002,
}
```

would yield 'New York' as the value of address in BibTEX, whereas with *librarian* it gives 'ny' (and the @string entry is simply ignored). In a situation as simple as the present one, you can make do with

macros in you document, i.e. a macro which when fed with 'ny' returns 'New York' (problems arise if 'ny' is the *real* value of a field, though), but it becomes more serious when you concatenate strings with the # operator. No way to hide the truth: *librarian* will get everything wrong. So if you're heavy on strings, better stick to BibTEX.

Examples

Although *librarian* doesn't use many commands, it might be hard to understand them from scratch. This section details a pair of example styles so you can get familiar with how things work. The first example is stripped to the minimum just to learn the basic use of *librarian*'s macros. The second example is an *Author (Year)* style with some refinements, dealing in particular with entries that have the same author and the same year and should thus be distinguished by a suffix, and it also shows how to handle cross-references. This style is thoroughly defined in the file `authoryear.tex`, distributed with *librarian*.

Remember, though, that what you can do in *librarian* is what you can do in TEX. There's nothing arcane in *librarian*, and what you can achieve depends really on your skill in TEX. So don't feel limited; for instance, suppose you want to make a list of your publications like:

> *2010*
> 'My fascinating paper', *Journal of Science*, volume 55 (22), 210–224.
> *The book I finally wrote*, Good Publishing House: Cityville.
> *2009*
> 'A talk on something', presented at...

That is, you want the year to be set above the entries, not repeated each time: then, for each entry, retrieve its `year` in a macro, and on the next one, compare with the current year. If it has changed, print the new value. (Of course entries should be sorted by date, otherwise it doesn't make sense.)

Another example: *librarian* doesn't abbreviate journal titles. But if you think in TEX, it is easily solved: just create a function from a journal title to its abbreviation (if any), and call it on the value of the `journal` field... and so on and so forth...

A simple unsorted list with numerical references, which gets sorted later

Our first example simply doesn't sort entries, and they are cited with the number they get when they appear. To make things a little funnier, though, we'll have a case of multiple bibliographies, e.g. one bibliography per chapter. In each chapter we set `\currentchapter`, and `\cite` is as follows:

```
\def\cite#1{%
  \Cite{#1}{\currentchapter}%
       {[\EntryNumber{\currentchapter}]}{[??]}%
  }
...
\def\currentchapter{chapter1}
```

And at the end of the chapter we'll have:

```
\ReadList{\currentchapter}
```

whereas the `\BibFile` command will appear at the end of the book. What `\cite` does is: store the entry in the list associated with the chapter, print '[??]' if there's no information associated with that entry (probably because it hasn't been read in the bibliographic file yet) and '[n]' instead, where *n* is its number in that list, e.g. '[3]' if that's the third entry in that chapter.

When `\ReadList` is called, we need to have `\MakeReference` ready, so here it is:

```
\def\empty{}
\def\MakeReference{%
  \par \noindent
  \llap{[\EntryNumber{\currentchapter}]~}%
  \RetrieveFieldIn{namenumber}\tempcount
  \ifx\tempcount\empty
    \def\tempcount{0}%
  \fi
  \ReadName\makerefname.
  \RetrieveField{title}.
  \RetrieveField{year}.\par
  }
```

This produces '[n]' in the left margin (with plain TEX's `\llap`

macro), and prints the name, the title, the year, separated by dots and a space. To put it simply, that's an utterly boring citation style, which doesn't take the entry type into account. The thing about `\tempcount` is: we'll need it in `\ReadName` with `\makerefname` to check whether we're at the last author. So we need it as a number; however, on the first compilation, since `\BibFile` appears after `\ReadList`, entries have no `namenumber` field, and thus `\tempcount` is empty... which is a very bad idea in an `\ifnum` test. In that case we set it to 0. Now here how names are typeset:

```
\def\space{ }
\def\makerefname{%
  \ifnum\NameCount>1
    \ifnum\NameCount=\tempcount\relax \space and \else , \fi
  \fi
  \Firstname
  \unless\ifx\Von\empty \Von\space \fi
  \space\Lastname
  \unless\ifx\Junior\empty , \Junior \fi
  }
```

We check for the existence of the von and junior parts and put them conditionally. Names are separated by a comma except the last one which takes 'and' instead. Note the crucial `\relax` after `\tempcount`: the latter is a macro expanding to a number, and it would gobble the next `\space` if `\relax` wasn't there. I could have swapped `\NameCount` and `\tempcount`, since the former is a real integer denotation, but we would have missed this fascinating conversation.
Now we're done with that style. Suppose however we want sorted entries. For instance:

```
\SortingOrder{name,year,title}{lfvj}
```

Now, citing entries with the citation number isn't very convenient. Instead, we want a number associated with the entry in the sorted list. We need a new count, and on each chapter it should be reset to 0. For instance:

```
\newcount\entrycount
...
\def\currentchapter{chapter24}
\entrycount=0
```

Next, we redefine `\cite` as follows:

```
\def\cite#1{%
  \Cite{#1}{\currentchapter}%
  {[\csname\EntryKey @\currentchapter\endcsname]}%
  {[??]}%
  }
```

Now `\cite` calls a command associated with the key (note that `\EntryKey` is useless, we could have used `#1` directly) and the chapter, which holds the number we want. The latter was set in `\MakeCitation`, redefined as:

```
\def\MakeReference{%
  \par \noindent
  \advance\entrycount1
  \WriteImmediateInfo{%
    \noexpand\expandafter\def
      \noexpand\csname\EntryKey @\currentchapter
      \noexpand\endcsname{\the\entrycount}%
    }%
  \llap{[\the\entrycount]~}%
% ... same as before
  }
```

This means: on each entry, advance `\entrycount` and store its value as the expansion of the command seen above, thanks to `\WriteImmediateInfo`. The rather clumsy code in the latter just means that the following will be written to the external file:

```
\expandafter\def\csname coover2002@chapter24\endcsname{13}
```

And of course, it is the value of `\entrycount` that we put into the left margin. Note that this style takes three compilations to get everything right: in the first one, a new entry can't be sorted, because `\SortList` appears before `\BibFile`. In the second compilation, the entry is sorted and its number is written to the external file. In the last one, that number is read and used when the entry is cited. Here's the style it produces:

Wallace's last book [2] was published eight years after his cult novel [1].
...
[1] David Foster Wallace. Infinite Jest. 1996.
[2] David Foster Wallace. Oblivion. 2004.

An Author (Year) style, with similar entries and cross-references

The basic idea here is to have a command `\cite{entry key}` that produces an *Author (Year)* citation in the main document (rather *Name (Year)*, but whatever). References are thus sorted according to names and then year, and if two entries are equal with regard to these fields, they should be sorted according to their relative order of appearance in the document and distinguished with a suffixes, e.g. 'Wallace (2003a)' and 'Wallace (2003b)'. To do so we'll use `\ifequalentry` and `\WriteImmediateInfo`.

However, such a simple `\cite` is definitely not very fun. So let's make it harder: it will take any number of entry keys and separate them with commas (e.g. 'Wallace (1996), Coover (2002)') unless the `name` field is identical, in which case it won't be repeated and the year will be put between the same parentheses as the previous entry (e.g. 'Wallace (1996, 2003)'). So `\cite` actually calls a recursive macro (`\readcite`) on its argument. Here's how it goes. (Since *librarian* requires eTEX, we can use it too; if you don't know anything about it, the only important thing is that you can prefix a conditional with `\unless` to reverse its evaluation; i.e. `\unless\ifwhatever A\else B\fi` is the same as `\ifwhatever B\else A\fi`. It's useful to write `\unless\ifwhatever` instead of `\ifwhatever\else` when you're interested in the false condition only.)

```
\def\terminator{\terminator} \def\empty{}
\def\cite#1{%
  \def\prevauthor{}%
  \readcite#1,\terminator,%
  }
\def\readcite#1,{%
  \def\temp{#1}%
  \ifx\temp\terminator
    )%
    \let\tail\relax
  \else
    \let\tail\readcite
    \unless\ifx\temp\empty
      \Cite{#1}{main}\makecitation{}%
    \fi
  \fi\tail
  }
```

Since we don't know what the next entry will be, if any, the final parenthesis is typeset when `\readcite` meets the terminator. (We take care of empty entries in case one writes '`\cite{entry1,entry2,}`' by mistake, which becomes '`entry1,entry2,,\terminator,`' for `\readcite`.) Note that in `authoryear.tex`, '(' and ')' are actually replaced by macros (`\leftcitemark` and `\rightcitemark`) that are defined as parentheses; it simply makes the switch to another style easier.

All entries are added to the `main` list, and at the end of the document we'll have to write:

```
% Perhaps some \SortDef's?
\BibFile{myfiles}
\SortingOrder{name,year}{lfvj}
\SortList{main}
\ReadList{main}
```

But back to citation. `\readcite` calls `\makecitation` in case there are fields for the entry. Here it is. If the current author is the same as in the previous entry (recorded in `\prevauthor`), then we just add a command and a space to the previous date:

```
\def\makecitation{%
  \RetrieveFieldIn{name}\temp
  \ifx\temp\prevauthor ,
```

On the other hand, if authors differ (unless the previous one is empty, which means we're on the first citation), we close the parenthesis (always added by the next entry to the previous one, remember?), adds a comma and a space to separate entries, and typeset the name with `\ReadName` (which will make use of `\tempcount`). We also typeset an opening parenthesis preceded by a hard space to welcome the upcoming year.

```
  \else
    \unless\ifx\prevauthor\empty ),\fi
    \RetrieveFieldIn{namenumber}\tempcount
    \ReadName\makecitename~(%
  \fi
```

And in any case we set `\prevauthor` to the current value of `name`, typeset the year and add the suffix associated with that entry if it exists (it is created when we use `\ReadList`), somehow like the entry number in the previous example.

```
  \RetrieveFieldIn{name}\prevauthor
  \RetrieveField{year}\csname\EntryKey @suffix\endcsname
  }
```

Now here's how names are typeset: they are separated by commas, except the last one which is separated by *and* (e.g. 'Jackson, Jameson and Johnson (1980)'). Besides, if there are more than 3 authors, only the first one is typeset, followed by *et alii*. The number 3 is of course totally arbitrary. Finally, each name is typeset as `\Von~\Lastname` with the `\Von` part optional, of course (maybe that makes too many hard spaces, though). Remember that `\tempcount` holds the value of the `namenumber` field and that `\NameCount` is the place of the author under investigation in the entire field. Hence, if they are more authors than allowed:

```
\chardef\namelimit=3
\def\makecitename{%
  \ifnum\tempcount>\namelimit
    \ifnum\NameCount=1
      \unless\ifx\Von\empty \Von~\fi
      \Lastname\space {\italics{et alii}}%
    \fi
```

Otherwise we typeset all names preceded by different markers: nothing for the first author, *and* for the last one, and commas in between.

```
  \else
    \unless\ifnum\NameCount=1
      \ifnum\NameCount=\tempcount\relax \space and
        \else , \fi
    \fi
    \unless\ifx\Von\empty \Von~\fi
    \Lastname
  \fi
  }
```

And we're done with the `\cite` command. We could add variation, e.g. `\citeauthor` to typeset the `name` field, `\citeyear` for the date, remove parentheses when needed, etc., but basically we've done the hardest part. The `\makecitename` macro shown here is slightly simpler than the real one in `authoryear.tex`. The latter first checks whether `\Lastname` is '`others`', in which case it typesets the *et alii* phrase (a basic feature of BibTEX).
Now we have to define `\MakeCitation`. It begins with `\ReadName` called with `\makerefname` as in the previous example. Even though `\makrefname` is slightly different (the first author is typeset as *von Last, Jr, First* whereas the other authors remain in the *First von Last, Jr* style), I won't show it here, because there's nothing much to learn.

```
\def\MakeReference{%
  \par\noindent
  \RetrieveFieldIn{namenumber}\tempcount
  \ReadName\makerefname
```

In `\MakeReference` still, we add a usual marker for collective works (`\editor` is simply defined as '`editor`' elsewhere).

```
  \RetrieveFieldIn{nametype}\temp
  \ifx\temp\editor
    \RetrieveFieldIn{namenumber}\temp
    \ifnum\temp>1 \space (eds.)\else \space (ed.)\fi
  \fi
```

Then we typeset the date between parentheses with a suffix, which will be set by `\comparentries` (and actually probably empty). This suffix is meant to distinguish entries that are equal according to `\SortingOrder`.

```
  \space (\RetrieveField{year}%
```

```
        \csname\EntryKey @suffix\endcsname)
  \compareentries
```

Typesetting the rest of the entry depends on the entry type, which is fed to `\typesetref`. At the end of the entry we call `\conditionalstop`, a stop that appears if and only if there was no stop before (see `authoryear.tex` for details).

```
  \RetrieveFieldIn{entrytype}\temp
  \typesetref\temp \conditionalstop
  }
```

The `\comparentries` macro tests whether we are in equal entries relative to sorting and if so gives them a suffix.

```
\newcount\sameentrycount
\def\compareentries{%
  \ifequalentry
    \advance\sameentrycount1
    \WriteImmediateInfo{%
      \noexpand\expandafter\def
        \noexpand\csname\EntryKey @suffix\noexpand\endcsname
        {\toletter}}%
  \else \sameentrycount=0 \fi
  }
```

The `\toletter` macro turns `\sameentrycount` into a letter and is quite uninteresting, so it isn't shown here. Note that this version of `\compareentries` works because the first argument of `\SortOrder` is set to '`name,year`', and hence two references written the same year by the same author are equal according to *librarian*, which can set the `\ifequelentry` conditional to the requested value. However, if `\SortOrder` was '`name,year,title`', then not only would the same two entries be sorted according to their titles, but most importantly they wouldn't be equal at all as far as *librarian* is concerned (unless they have identical titles, of course), although in this bibliographic style they should be seen as such. In that case, `\compareentries` should be more refined and store each entry's `name` and `year` fields in a macro to be compared to the next one.
But we were calling `\typesetref` on the `entrytype` field. It launches a macro which typesets the rest of the entry according to its type, and `\createtype` is meant to define such a macro:

```
\def\typesetref#1{%
  \ifcsname#1@entrytype\endcsname
    \csname#1@entrytype\endcsname
  \else
    \errmessage{Unknown entry type: `#1'}%
  \fi
  }
\def\createtype#1{%
  \expandafter\def\csname#1@entrytype\endcsname
  }
```

Before creating entries, here are some useful macros to be used with `\TypesetField`. Indeed, to typeset references properly, punctuation should be added if and only if the corresponding field exists.

```
\def\booktitle#1{\setbooktitle{\RetrieveField{#1}}}
\def\setbooktitle#1{\italics{#1}}
\def\articletitle#1{\setarticletitle{\RetrieveField{#1}}}
\def\setarticletitle#1{`#1'}
\def\addcomma#1{, #1}
\def\addjournal#1{\addcomma{{\setbooktitle{#1}}}}
\def\addcolon#1{: #1}
\def\addpar#1{(#1)}
\def\addbook#1{, in \setbooktitle{#1}}
\def\addeditor#1{%
  \RetrieveFieldIn{editornumber}\tempcount
  , edited by \ReadEditor\makeedname}
```

The `\italics` macro in `\setbooktitle` should be set to whatever you use to typeset italics. Now, we can create entries:

```
\createtype{book}{%
  \booktitle{title}%
  \TypesetField{publisher}\addcomma{}%
  \TypesetField{address}\addcolon{}%
  }
\createtype{article}{%
  \articletitle{title}%
  \TypesetField{journal}\addjournal{}%
```

```
  \TypesetField{volume}\addcomma{}%
  \TypesetField{number}\addpar{}%
  \TypesetField{pages}\addcomma{}%
  }
\createtype{incollection}{%
  \articletitle{title}%
  \TypesetField{crossref}\crossref{%
    \TypesetField{booktitle}\addbook{}%
    \TypesetField{editor}\addeditor{}%
    \TypesetField{pages}\addcomma{}%
    \TypesetField{publisher}\addcomma{}%
    \TypesetField{address}\addcolon{}%
    }%
  }
```

... and many more! That's the basic job of a bibliographic style. Note that we make good use of `\TypesetField`, but not with the `title` field, because it is very unlikely to be missing. Now, give a look at the `incollection` entry; once the title is typeset, it reads as follows: call the `\crossref` macro on the value of the `crossref` field if it has any, otherwise, typeset the information about the book. And `\crossref` is:

```
\def\crossref#1{%
  , in \cite{#1}%
  \WriteImmediateInfo{\noexpand\Cite{#1}{main}{}{}}%
  }
```

Since the value of the `crossref` field is an entry key, we simply `\cite` that entry. However, this is not enough; `\SortList` can deal with entries if and only if they are cited before it is called; on the other hand, `\ReadList`, where the cross-reference happens, must appear after `\SortList`. That's why we pass a `\Cite` command to the next compilation; note that the third argument does nothing.
Thus, supposing the reference '`Gaddis1994`' has a `crossref` field to '`Gass2000`', `\cite{Gaddis1994}` in a document will produce the following in the bibliography:

> Gaddis, William (1994) 'Old Foes with New Faces', in Gass and Cuoco (2000).
> Gass, William H. and Lorin Cuoco (eds.) (2000) *The Writer and Religion*, Southern Illinois University Press: Carbondale.

What if we don't want to see the '`Gass2000`' reference in the bibliography and instead want something like this:

> Gaddis, William (1994) 'Old Foes with New Faces', in *The Writer and Religion*, edited by William H. Gass and Lorin Cuoco, Southern Illinois University Press: Carbondale, 2000.

Then `\crossref` should be something like this:

```
\def\crossref#1{%
  \bgroup
    \lowercase{\def\EntryKey{#1}}%
    \TypesetField{title}\addbook{}%
    \TypesetField{editor}\addeditor{}%
    \TypesetField{pages}\addcomma{}%
    \TypesetField{publisher}\addcomma{}%
    \TypesetField{address}\addcolon{}%
    \TypesetField{year}\addcomma{}%
  \egroup
  \WriteImmediateInfo{\noexpand\Cite{#1}{crossref}{}{}}%
  }
```

That is: we apply the same macros to a different entry by redefining `\EntryKey` temporarily (hence the group). Note that this redefinition must happen in `\lowercase`, because there we're away from *librarian*'s handling of entry keys. Note also that we still `\Cite` the entry in the external file, because the citation is necessary to retrieve fields in the bibliographic files; however the entry is added to a secondary list '`crossref`' that we'll never sort nor read.

Typeset with LuaTEX v.0.6
in Garamond Pro (Robert Slimbach)
and Inconsolata (Raph Levien)