BIBTOOL

A Tool to Manipulate BibTeX Files

Version 2.55

BibTool Manual

Gerd Neugebauer

Abstract

BibTeX provides an easy to use means to integrate citations and bibliographies into LaTeX documents. But the user is left alone with the management of the BibTeX files. The program BibTool is intended to fill this gap. BibTool allows the manipulation of BibTeX files which goes beyond the possibilities—and intentions—of BibTeX. The possibilities of BibTool include sorting and merging of BibTeX data bases, generation of uniform reference keys, and selecting of references used in a publication.

This file is part of BibTool Version 2.55

Gerd Neugebauer
Im Lerchelsböhl 5
64521 Groß-Gerau (Germany)

Net: http://www.gerd-neugebauer.de/

E-Mail: gene@gerd-neugebauer.de

Contents

1. Introduction

The user's manual is divided into two parts. In this first part the big picture on BibTool is shown. The next chapter after this one is then devoted to the nitty gritty details.

1.1. Related Programs

BibTeX [Lam94, Pat88a, Pat88b] is a system for integrating bibliographic information into LaTeX [Lam94] documents. BibTeX is designed to serve exactly this purpose. It has shown that various tasks in relation with managing bibliographic databases are not covered by BibTeX. Usual activities on bibliographic databases include

- inserting new entries
- editing
- using citations in documents
- sorting and merging of bibliographic data bases
- extraction of bibliographic data bases

Since only the integration in documents is covered by BibTeX several utilities emerged to fill the gaps. We will sketch some of them shortly.

BibTeX is a program by Oren Patashnik to select publications used in a LaTeX document and format them for inclusion into this document. This program should be part of each TeX installation.

bibclean is a program by Nelson H.F. Beebe to pretty-print BibTeX files. It also can act as syntax checker. The C sources can be compiled on several systems.

bibindex/biblook is a pair of programs by Nelson H.F. Beebe to generate an index for a BibTeX file and use it to perform a fast look-up of certain entries. The programs so far run only under UNIX.

bibsort is a UNIX shell script by Nelson H.F. Beebe to sort a BibTeX file.

bibextract is a UNIX shell script by Nelson H.F. Beebe to extract entries from a BibTeX file which are used in a LaTeX document.

lookbibtex/bibdestringify are Perl scripts by John Heidemann to extract entries from a BibTeX file which are used in a LaTeX document and to remove strings from a BibTeX file.

bibtools is a collection of UNIX shell scripts by David Kotz to add and extract entries to bibliographic databases. Several small programs are provided to perform special tasks.

bibview is a Perl script by Dana Jacobsen to extract entries from a BibTeX file which are used in a LaTeX document.

BibCard is a program by William C. Ogden running under X11/xview which provides a means to edit bibliographic databases.

hyperbibtex Something similar for Macintosh computers.

xbibtex/bibprocess/bibsearch are programs by Nicholas J. Kelly and Christian H. Bischof running under X11 which provides a means to edit bibliographic databases, add fields to a BibTeX file and extract certain entries from a BibTeX file.

bibview is an X11 program by Holger Martin, Peter Urban, and Armin Liebl to search in and manipulate BibTeX files. It is similar to BibCard and hyperbibtex.

tkbibtex is a BibTeX file browser with support for editing, searching sorting and merging. Written by Peter Corke in Tcl/Tk it runs under Unix and Windows.

bibdb Editor for BibTeX files that runs under Dos and Windows.

qbibman is a graphical user interface by Ralf Görtz utilizing BibTool as underlying library. It is written in C++ and uses Qt.

Barracuda an X11 Editor for BibTeX files, written in C++ and Qt.

BibTeX-**Mode** is an extension of the editor GNU-Emacs to provide means to edit BibTeX files. Several useful operations are provided. There is also a BibTeX-Mode for the Emacs-like JED-Editor.

btOOL is a Perl library to access BibTeX files. It is implemented in Perl and C and has been written by Greg Ward.

This is a selection of some programs I have heard of. I have tested some of them and I have skipped through the documentation of others. Thus the description may be too short or incomplete. Some additional information can be found in [GMS94, Chapter 13].

Most of those utilities are tailored towards a particular operating system and thus they are not available on other platforms. Most of these program are made to perform a single task. Often they can not be configured to suit a personal taste of a user.

Still there are some points not covered by the utilities mentioned above. BibTool tries to provide the missing features and integrate others into a single tool.

1.2. Using BibTool—Some Instructive Examples

BibTool has been developed on UN*X and UN*X-like machines. This has influenced many of the design decisions. Version 1 was controlled using numerous command line options. This way of controlling has been supplemented in version 2 by the concept of a

resource file. This resource file allows the modification of the various internal parameters determining the behavior of BibTool.

When BibTool has been compiled correctly there should be an executable file named `bibtool`[1]. We will assume that you are running BibTool from a command line interpreter. There you can simply issue the command

```
bibtool
```

Now BibTool will start reading from the standard input lines obeying the rules of a BibTeX file.[2] The entries read are pretty-printed on the standard output. It is obvious that this behavior is not very useful in itself. The origin of this kind of interface lies in the concepts of UN*X where many commands can act as filters.

Usually we do not intend to use BibTool in this way. Thus we need a way to specify an input file. This is simply done by adding the file name as argument after the command name like in

```
bibtool file.bib
```

The result of this command can at once be seen on the screen. The contents of the file `file.bib` is pretty printed.

Now that we have seen the simplest case of the application of BibTool we will see the case of a useful application of BibTool. This application is the sorting and merging of BibTeX databases.

1.2.1. Sorting and Merging

BibTeX files can be sorted by specifying the command line option `-s`. The given files are sorted according to the reference key. Several files can be given at once in which case BibTool will sort and merge those files.

```
bibtool -s file1.bib file2.bib
```

With the command line option the files are sorted in reverse ASCII order.

```
bibtool -S file1.bib file2.bib
```

If you want to sort the BibTeX files according to the authors then the following invocation should do the trick:

```
bibtool -- sort.format="%N(author)" file1.bib file2.bib
```

[1] Maybe with an additional extension.
[2] We assume that no resource file can be found. Resource files will be described later.

This means that the sorting order is determined by the (normalized) author field. Note that single quotes encapsulating the sort.format are necessary to prevent the command line interpreter to gobble the special characters.

1.2.2. Key Generation

Once you have a reference and you insert it into a BibTeX file you have to assign a reference key to it. The problem is to find a key which is unique and meaningful, i.e. easy to remember. The easiest way to remember a key is to use an algorithm to create it and remember the algorithm—which is the same for all keys.

One algorithm which comes to mind is to use the author and (an initial part) of the title. Alternatively we can use the author and the year. But the problem is with industrious authors writing more than one publication per year. The necessary disambiguation of such references is not very intuitive. However, BibTool has the capability to describe desired keys. Thus, the alternatives described above can be realized.

For this section we want to use the following BibTeX entry as our example:[3] Suppose it is contained in a file named sample.bib.

```
@ARTICLE{article-full,
   author = {L[eslie] A. Aamport},
   title = {The Gnats and Gnus Document Preparation System},
   journal = {\mbox{G-Animal's} Journal},
   year = 1986,
   volume = 41,
   number = 7,
   pages = "73+",
   month = jul,
   note = "This is a full ARTICLE entry",
}
```

First, we want to see how we can make keys consisting of author and title. This is one of my favorite algorithms thus it is rather easy to use it. You simply have to run the following command:

```
bibtool -k sample.bib -o sample1.bib
```

After the command has colpleted it's work the following entry can be found in the output file sample1.bib:

```
@Article{         aamport:gnats,
  author        = {L[eslie] A. Aamport},
  title         = {The Gnats and Gnus Document Preparation System},
  journal       = {\mbox{G-Animal's} Journal},
  year          = 1986,
  volume        = 41,
  number        = 7,
```

[3] Shamelessly stolen from the BibTeX xamples.bib file.

```
  pages          = "73+",
  month          = jul,
  note           = "This is a full ARTICLE entry"
}
```

You see that the reference key has been changed. It now consists of the last name and the first relevant word of the title, separated by a colon. Sometimes it might be desirable to incorporate the initial names as well. This can be achieved by the command

```
bibtool -K sample.bib -o sample1.bib
```

The resulting reference key is `aamport.la:gnats`. The initials are appended after the first name. Thus the usual lexicographic order on the keys will (hopefully) bring together the publications of the same first author.

Another alternative is to use the author and the year. This can be achieved with the following command:[4]

```
bibtool -f %n(author):%2d(year) sample.bib -o sample1.bib
```

The resulting key is `Aamport:86`. Note that the last example works as desired for our sample file. But for a real application of this technique a deep understanding of the key generation mechanism as described in section A.9 is necessary.

1.2.3. Normalization

BibTool can be used to normalize the appearance of BibTeX databases. As an example we can consider the different forms of delimiters for fields. BibTeX allows the use of of braces or double quotes. Now it can be desirable to use one style only. For this purpose the rewriting facility of BibTool can be applied.

```
bibtool --  'rewrite.rule={"^\"\([^#]*\)\"$" "{\1}"}' -o out.bib
```

Since this seems to be rather cryptic we will have a closer look at this example. First we have to mention that the outer quotes are there because the UN*X shell (csh, sh, bash,...) treats some characters special and we want to avoid this to happen to the rewrite rule given. A similar quoting mechanism might be required for all command line interpreters.

The rewrite rule is applied to any field. The first string—called pattern—which is enclosed in double quotes is matched against the contents of the field. If a match is found then the matching sub-string is replaced by the replacement text in the second string.

The pattern is a regular expression like the ones used in Emacs. The first character is the hat (^). This character anchors the match at the beginning of the line. The last

[4] Note that some command line interpreters (like the UN*X shells) require the format string to be quoted (enclosed in single quotes).

character is the dollar sign which anchors the end at the end of the field value. Thus only complete matches are considered.

Since we want to find those fields whose values are enclosed in double quotes they are given after the hat and before the dollar. To avoid a misinterpretation as the end of the pattern they have to be quoted with the backslash (\).

Next we have the parentheses \(...\). They are instructions to memorize the matching sub-string in a register. Since it is the first instruction of this kind the register number 1 is used.

Now we come to the point where we have to specify the contents of the string. For this purpose we use a character class—written as [...]. Since the first character in this class specification is a hat this class consists of all characters but those given after the hat. Thus all characters but the hash sign (#) are allowed.

The star (*) after the character class indicates that an arbitrary number of characters of this class are allowed.

We have used the complicated construction with a character class to avoid wrong results which would have resulted when this rewrite rule is applied to a concatenated field value like the following one:

```
author = "A. U. Thor" # " and " # "S. O. Meone"
```

Such fields are left unchanged by the rewrite rule given above. We could have used the point (.) instead of the character class since the point matches any character. But this would have let to the syntactic wrong result:

```
author = {A. U. Thor" # " and " # "S. O. Meone}
```

But we have to complete the explanation of the rewrite rule. The remaining part is the replacement text. Here we just have to note that the sub-string \1 is not copied verbose but replaced with the contents of the first register. This register contains the contents of the field without the delimiting double quotes.

Thus we have a solution to our initial problem which is conservative in the sense that it sometimes fails but never produces a wrong result.

1.2.4. Extracting Entries for a Document

BibTool can be used to extract the references used in a document. For this purpose BibTool analyzes the .aux file and takes the information given there. This includes the names of the BibTeX files. Thus no BibTeX files have to be given in the command line. Instead the .aux file has to specified—preceded by the option -x.

```
bibtool -x document.aux -o document.bib
```

The second option `-o` followed by a file name specifies the destination of the output. This means, instead of writing the result to the standard output stream the result is written into this file.

1.2.5. Extracting Entries Matching a Regular Expression

BibTool can be used to extract the references which fulfill certain criteria. Those criteria can be specified utilizing regular expressions.[5] As a special case we can extract all entries containing a certain sub-string of the key:

```
bibtool -X tex all.bib -o some.bib
```

This instruction selects all entries containing the sub-string `tex` in the key. The second option `-o` followed by a file name specifies the destination of the output. Thus instead of writing the result to the standard output stream the result is written into this file.

Next we want to look up all entries containing a sub-string in some of its fields. For this purpose we search for the string in all fields first:[6]

```
bibtool -- select{"tex"} all.bib -o some.bib
```

Note that the comparison is not done case sensitive; however this can be customized (see page 34).

Finally we want to select only those entries containing the sub-string in anyone of certain fields. For this purpose we simply specify the names of those fields in the *select* instruction:

```
bibtool -- select{title booktitle $key "tex"} all.bib -o some.bib
```

This example extracts all entries containing the sub-string `tex` in the title field, the booktitle field, or the reference key.

After we have come so far we can say that the first example in this section is in fact a short version of the following command:

```
bibtool -- select{$key "tex"} all.bib -o some.bib
```

As a simple case of extraction we might want to extract all books from a bibliography. This can be done with the following command:

```
bibtool -- select{@book} all.bib -o some.bib
```

A similar method can also be applied for other entry types.

[5] Those features are only usable if the regular expression library has been enabled during the configuration of BibTool—which is the default.

[6] Note that some command line interpreters (e.g the UN*X shells) might need additional quoting of the select instruction since it contains special characters.

Note Usually cross-referenced entries are not selected automatically. This can result in incomplete—and thus incorrect—BibTeX files. To avoid this behavior use the following command:

```
bibtool -- select{book} -c all.bib -o some.bib
```

1.2.6. Translating ISO 8859-1 Characters

Sometimes you need to translate some special characters into BibTeX sequences. Suppose you have edited a BibTeX file and by mistake used those nice characters that are incompatible with standard ASCII as used in BibTeX. You can use BibTool to do the trick:

```
bibtool -r iso2tex -i iso.bib -o ascii.bib
```

1.2.7. Correctly Sorting Cross-referenced Entries

BibTeX has a restriction that a cross-referenced entry has to come after the referencing entry. This can be achieved by putting all entries containing a field "crossref" before those containing none. As second sorting criterion we want to use the reference key.

This can be achieved with a resource file containing the following instructions

```
sort.format = {{%1.#s(crossref)a#z}$key}
sort.reverse = off
sort = on
```

The magic is contained in the first instruction. Thus we will examine it in detail:

`%1.#s(crossref)`
: This formatting instruction does not produce any output but simply acts as condition to determine whether or not to include the following string. The condition counts the allowed characters (`#s`) of the field `crossref` and compares this number with the given interval $[1, \infty]$ (`1.`).

  Thus it detects those entries containing a non empty crossref field.

`%1.#s(crossref)a`
: If the condition holds then the string `a` is used as part of the sort key.

`{%1.#s(crossref)a#z}`
: If the first condition fails then the next alternative after the hash mark (`#`) is considered. This is the string `z` which will always succeed and thus be included into the sort key.

  Thus this construction will produce `a` if a crossref field is present and not empty or `z` otherwise.

`{%1.#s(crossref)a#z}$key`
: Finally the reference key (`$key`) is appended to the characterizing initial letter.

The sorting according to ascending ASCII order will bring all the entries with crossref fields to the beginning.

1.3. Interfacing BibTool with Other Programming Languages

BibTool can be used as a means for other programming languages to access BibTeX data bases. In this course BibTool reads the BibTeX file and prints it in a normalized form which makes it easy for the host programming language to parse it and get the information about the entries and fields.

In addition BibTool can already preselect several entries or do other useful transformations before the host programming language even sees the contents. Thus it is fairly easy to write a CGI script (e.g. in Perl) which utilizes BibTool to extract certain entries from a BibTeX data base and presents the result on a HTML page.

Currently the distribution of BibTool contains frames of programs in Perl and Tcl which can be used as a basis for further developments.

I am working towards making BibTool a linkable library of C code. As one step into this direction the exported functions and header information has been documented. This documentation is contained in the distribution.

A tight integration of BibTool into another programming language is possible. As an experiment into this direction I have chosen Tcl as the target language. The result is BibTcl which is contained in the distribution of BibTool.

1.4. Getting BibTool, Hot News, and Bug Reports

Usually BibTool can be found on the CTAN or one of its mirrors. Thus you can get BibTool via ftp or extract it from a CDROM containing a dump of the CTAN. The CTAN (Comprehensive TeX Archive Network) consists of the following major sites (and many mirrors):

`www.dante.de`
`www.tex.ac.uk`

BibTool can be found in the following directory:

`tex-archive/biblio/bibtex/utils/bibtool`

BibTool is hosted in a public repository at Sarovar. The repository contains the released sources as well as the development versions. The repository can be found at

`http://sarovar.org/projects/bibtool/`

I have set up a WWW page for BibTool. It contains a short description of the features and links to the documentation and the current downloadable version in source form. The URL is:

http://www.gerd-neugebauer.de/software/TeX/BibTool/

In addition, this page contains a description of the current version of BibTool and a list of changes in the last few releases.

If you encounter problems installing or using BibTool you can send me a bug report to my email address `gene@gerd-neugebauer.de`. Please include the following information into a bug report:

- The version of BibTool you are using.
- Your hardware specification and operating system version.
- The C compiler you are using and its version. (Only for compilation and installation problems)
- The resource file you are using. Try to reduce it to the absolute minimum necessary for demonstrating the problem.
- A *small* BibTeX file showing the problem.
- The command line options of an invocation of BibTool making the problem appear.
- A short justification why you think that the behavior is an error.

I have had the experience that compiling this information has helped me find my own problems in using software. Thus I could fix several problems before sending a bug report.

On the other side I have unfortunately also had the experience that I have got complains about problems in my software. After several questions it turned out that the program had not been used properly.

Oh, sure. There have been bugs and I suppose there are still some bugs in BibTool. I am grateful for each hint which helps me eliminating these bugs.

1.5. Contributing to BibTool

As you might have read BibTool is free software in the sense of the Free Software Foundation. This means that you can use it as a whole or parts of it as long as you do not deny anyone to have the sources and use it freely. See the file `COPYING` for details.

If you feel morally obliged to provide compensation for the use of this program I have the following suggestions.

- Proofread this documentation and report any errors you find as well as additional material to put in.

- Provide additional contributed pieces to BibTool. For instance useful resource files which could be included into the library.
- Write a useful program and release it to the public without making profit, preferably under an Open Source license like the `GNU General Public License` or the GNU artistic license.

A. Reference Manual

This part of the documentation tries to describe all commands and options. Since the behavior of BibTool can be adjusted at compile time not all features may be present in your executable. Thus watch out and complain to the *installer* if something is missing.

A.1. Beware of the Command Line

Be aware that command line interpreters have different ideas about what to do with a command line before passing the arguments to a program. Thus it might be necessary to carefully quote the arguments. Especially if the command contains spaces it is very likely that quoting is needed.

For instance in UN*X shells it is in general a good strategy to enclose command line arguments in single quotes (`'`) if they contain white-space or special characters like `\`, `$`, `&`, `!`, or `#`.

Instead of excessively using command line arguments it is preferable and less error-prome to put the major configuration into a resource file and just include this resource file on the command line. Details on this are described in the next section.

A.2. Command Line Usage and Resource Files

BibTool can be controlled either by arguments given in the command line or by commands given in a file (or both). Those command files are called resource files. If BibTool is installed correctly you should have the executable command `bibtool` (maybe with an additional extension). Depending on your computer and operating system you can start BibTool in different ways. This can be done either by issuing a command in a command line interpreter (shell), by clicking an icon, or by selecting a menu item. In the following description we will concentrate on the use in a UN*X like shell. There you can type simply

```
bibtool
```

Now BibTool is waiting for your input. As you type BibTool reads what you type. This input is interpreted as data conforming BibTeX file rules. The result is printed when BibTool is finished. You can terminate the reading phase with your End-Of-File character (e.g. Control-D on UN*X, or Control-Z on MS-D*S)

This application in itself is rather uninteresting. Thus we come to the possibility to give arguments to BibTool. The simplest argument is `-h` as in

```
bibtool -h
```

This command should print the version number and a short description of the command line arguments to the screen.

The next application is the specification of resources. Resource files can be given in the command line after the flag `-r`.

```
bibtool -r resource_file
```

In this way an arbitrary number of resource files can be given. Those resource files are read in turn and the commands contained are evaluated. If no resource file is given in the command line BibTool tries to find one in standard places. First of all the environment variable `BIBTOOLRSC` is searched. If it is defined then the value is taken as a list of resource file names separated by colon (UNIX), semicolon (DOS), or comma (Amiga). All of them are tried in turn and loaded if they exist. If the environment variable is not set or no file could be loaded successfully then the default resource file (usually the file `.bibtoolrsc`) is tried to be read in the home directory (determined by the environment variable `HOME`) or the current directory.

The resource files are searched similar to the searching mechanism for BibTeX files (see section A.3). The extension `.rsc` is tried and a search path can be used. This search path is initialized from the environment variable `BIBTOOL`. Initially only the current directory is on the search path. The search path can also be set in a resource file (for following resource file reading). This can be achieved by setting the resource resource.search.path.

```
resource.search.path = path
```

When an explicit resource file is given in the command line the defaults are not used. To incorporate the default resource searching mechanism the command line option `-R` can be used:

```
bibtool -R
```

Now let us consider some examples. Suppose that the current directory contains a default resource file (named .bibtoolrsc) and an additional resource file my_rsc.

The following invocation of BibTool uses only the resource file *my_rsc*:

```
bibtool -r my_rsc -i sample
```

If you want to initialize the resources from the default resource file before you can use the `-R` *before* the inclusion of the resource file:

```
bibtool -R -r my_rsc -i sample
```

If you add the `-R` argument after the resource specification then the default resource is evaluated after your resource file. Thus settings are potentially overwritten:

```
bibtool -r my_rsc -R -i sample
```

Additionally note that resource files are evaluated at once whereas input files are read in one chunk at the end. Thus you can not specify one set of parameters to be used for one file and another set of parameters for the next file. This is impossible within one invocation of BibTool[1].

As a consequence of this behavior the last example is equivalent to the following invocations:

```
bibtool -r my_rsc -i sample -R
```

```
bibtool -i sample -r my_rsc -R
```

Now we have to describe the commands allowed in a resource file. The general form of a resource command is of the form

name = {*value*}

name is the resource name which conforms the rules of BibTeX reference keys. Thus *name* can be composed of all characters but white-space characters and

```
"  #  %  '  (  )  ,  =  {  }
```

Resource names are currently composed of letters and the period. The next component is an optional equality sign (=). The equality sign is recommended as it helps detecting syntax problems. White-space characters surrounding the equality sign or separating resource name and resource value are ignored. The resource value can be of the following kind:

- A number composed of digits only.
- A string conforming the rules of resource names, i.e. made up of all but the forbidden characters described above.
- A string containing arbitrary characters delimited by double quotes (") not containing double quotes. Parentheses and curly brackets have to come in matching pairs.
- A string containing arbitrary characters delimited by curly brackets ({}). Parentheses and curly brackets have to come in matching pairs.

[1] This might be changed in the next major revision (3.0).

You can think of resource names as variables or functions in a programming language. Resource commands simply set the variables to the given value, add the value to the old value, or initiate a action. There are different types of resources

- Boolean resources can take only the values on and off. The values on, t, true, 1, and yes are interpreted as the same. For those values the case of the letters is ignored. Thus true and TrUe are the same. Every other value else is interpreted as off.
- Numeric resources can take numeric values only.
- String resources can take arbitrary strings.

Usually white-space characters are ignored. There is one exception. The characters % and # act as comment start characters if given between resource commands. All characters to the end of the line are ignored afterwards.

Now we come the description of the first resource available. To read in additional resource files the resource file may contain the resource

```
resource {additional/resource/file}
```

Thus the resource given above has the same functionality as the command line option -r described above. Path names should be specified in the normal manner for your operating system.

One resource command useful for debugging is the print resource. The resource value is immediately written to the error stream. The output is terminated by a newline character. Certain translations are performed during the reading of a resource which can be observed when printing. Each sequence of white-space characters is translated into a single space.

To end this subsection we give an example of the print resource. In this sample we also see the possibility to omit the equality sign and use quotes as delimiters.

```
print "This is a stupid message."
```

Finally we can note that the commands given in a resource file can also be specified on the command line. This can be achieved with the command line option -- The next command line argument is taken as a resource command.

```
bibtool -- resource_command
```

This can be used to issue resource commands which do not have a command line counterpart. One example we have already seen. The print instruction can be used from the command line with the following

```
bibtool -- print{hello_world}
```

Summary

Option	*Resource command*	*Description*
`-h`		Show a list of command line options.
`-R`		Immediately evaluate the instructions from the default file.
	print {*message*}	Write out the text *message*.
`-r` *file*	*resource* = *file*	Immediately evaluate the instructions from the resource file *file*.
	resource.search.path	List of directories to search for resource files.
`--` *rsc*	*rsc*	Evaluate the resource instruction *rsc*.

A.3. Input File Specification and Search Path

An arbitrary number of input files can be specified. Input files can be specified in two ways. The command line option `-i` is immediately followed by a file name. Since no restriction on the file name is applied this can also be used to specify files starting with a dash.

```
bibtool -i input_file
```

The resource name input can be used to specify additional input files.

```
input {input_file}
```

Input files are processed in the order they are given. If no input file is specified the standard input is used to read from.

Depending on the special configuration of BibTool there are two ways of searching for BibTeX files. The native mode of BibTool uses a list of directories and a list of extensions to find a file. Alternatively the kpathsea library can be used which provides additional features like the recursive searching in sub-directories. First we look at the native BibTool searching mechanism.

The files are searched in the following way. If the file is can't be opened as given the extension `.bib` is appended and another read is tried. In addition directories can be given which are searched for input files. The search path can be given in two different ways. First, the resource name bibtex.search.path can be set to contain a search path specification.

```
bibtex.search.path = {directory1:directory2:directory3}
```

The elements of the search path are separated by colons. Thus colons are not allowed as parts of directories. Another source of the search path is the environment variable `BIBINPUTS`. This environment variable is usually used by BibTeX to specify the search

path. The syntax of the specification is the same as for the resource bibtex.search.path. To check the appropriate way to set your environment variable consult the documentation of your shell, since this is highly dependent on it.

To allow adaption to operating systems other than UN*X the following resources can be used. The name of the environment bibtex.env.name overwrites the name of the environment variable which defaults to `BIBINPUTS`.

```
bibtex.env.name = {ENVIRONMENT_VARIABLE}
```

The first character of the resource env.separator is used as separator of directories in the resource bibtex.search.path and the environment variable given as bibtex.env.name.

```
env.separator = {:}
```

The default character separating directories in a file name is the slash (`/`). The first character of the resource dir.file.separator can be used to change this value.

```
dir.file.separator = {\}
```

Note that the defaults for env.separator and dir.file.separator are set at compile time to a value suitable for the operating system. Usually you don't have to change them at all. For instance for MSD*S machines the env.separator is usually set to `;` and the dir.file.separator is usually set to `\`.

If the kpathsea library is used for searching BibTeX files then some of the resources described above have no effect. They are replaced by their kpathsea counterparts. Most probably you are using the kpathsea library already in other TeX related programs. Thus I just have to direct you to the documentation distributed with the kpathsea library for details.

Summary

Option	*Resource command*	*Description*
	bibtex.env.name={*var*}	Use the environment variable *env* to add more directories to the search path for BibTeX (input) files.
	bibtex.search.path={*path*}	Use the list of directories *path* to find BibTeX (input) files.
	dir.file.separator={*c*}	Use the character *c* to separate the directory from the file.
	env.separator={*c*}	Use the character *c* to separate directories in a path.
`-i` *file*	*input*{*file*}	Add the BibTeX file *file* to the list of input files.

A.4. Output File Specification and Status Reporting

By default, the processed BibTeX entries are written to the standard output. This output can be redirected to a file using the command line option `-o` as in

```
bibtool -o output_file
```

The resource name output.file can also be used for this purpose.

```
output.file = {output_file}
```

No provisions are made to check if the output file is the same as a input file.

A second output stream is used to display error messages and status reports. The standard error stream is used for this purpose.

The messages can roughly be divided in three categories: error messages, warnings, and status reports. Error messages indicate severe problems. They can not be suppressed. Warnings indicate possible problems which could (possibly) have been corrected. They are displayed by default but can be suppressed. Status reports are messages during the processing which indicate actions currently performed. They are suppressed by default but can be enabled.

Warning messages can be suppressed using the command line option `-q`. This option toggles the Boolean quiet value.

```
bibtool -q
```

The same effect can be obtained by assigning the value on or off to the resource quiet:

```
quiet = on
```

Status reports are useful to see the operations performed. They can be enabled using the command line option `-v`. This option toggles the Boolean verbose value.

```
bibtool -v
```

The same can also be achieved with the Boolean resource verbose:

```
verbose = on
```

Another output stream can be used to select the string definitions. This is described in section A.13 on macros.

Summary

Option	*Resource command*	*Description*
`-o` *file*	*output.file* {*file*}	Direct output to the file *file*.
`-q`	*quiet*=*on*	Suppress warnings. Errors can not be suppressed.
`-v`	*verbose*=*on*	Enable informative messages on the activities of BibTool.

A.5. Parsing and Pretty Printing

The first and simplest task we have to provide on BibTeX files is the parsing and pretty printing. This is not superfluous since BibTeX is rather pedantic about the accepted syntax. Thus I decided to try to be generous and correct as many errors as I can.

Each input file is parsed and stored in an internal representation. BibTeX simply ignores any characters between entries. BibTool stores the comments and attaches them to the entry immediately following them. Normally anything between entries is simply discarded and a warning printed. The Boolean resource pass.comments can be used to change this behavior.

```
pass.comments = on
```

If this resource is on then the characters between entries are directly passed to the output file. This transfer starts with the first non-space character after the end of an entry.

The standard BibTeX styles support a limited number of entry types. Those are predefined in BibTool. Additional entry types can be defined using the resource new.entry.type as in

```
new.entry.type {Anthology}
```

This option can also be used to redefine the appearance of entry types which are already defined. Suppose we have defined *Anthology* as above. Afterwards we can redefine this entry type to be printed in upper case with the following option:

```
new.entry.type {ANTHOLOGY}
```

Each undefined entry type leads to an error message.

When a database is printed the different kinds of entries are printed together. For instance all normal entries are printed en block. The order of the entry types is determined by the resource print.entry.types. The value of this resource is a string where each character represents an entry type to be printed. If a letter is missing then this part of the database is omitted. The following letters are recognized—uppercase letters are folded to their lowercase counterparts if they are not mentioned explicitly:

a The aliases of the database.

c The comments of the database which are not attached to an entry.

i The includes of the database.

m The modifies of the database.

n The normal entries of the database.

p The preambles of the database.

$ The strings (macros) of the database.

S The strings (macros) of the database which are used in the other entries.

s The strings (macros) of the database where the resource print.all.strings determines whether all strings are printed or the used ones only.

The following invocation prints the preambles and the normal entries only. This can be desirable if the macros are printed into a separate file.

```
print.entry.types {pn}
```

The internal representation is printed in a format which can be adjusted by certain options. Those options are available through resource files or by specifying resources on the command line.

print.line.length This numeric resource specifies the desired width of the lines. lines which turn out to be longer are tried to split at spaces and continued in the next line. The value defaults to 77.

print.indent This numeric resource specifies indentation of normal items, i.e. items in entries which are not strings or comments. The value defaults to 2.

print.align This numeric resource specifies the column at which the '=' in non-comment and non-string entries are aligned. This value defaults to 18.

print.align.key This numeric resource specifies the column at which the '=' in non-comment and non-string entries are aligned. This value defaults to 18.

print.align.string This numeric resource specifies the column at which the '=' in string entries are aligned. This value defaults to 18.

print.align.preamble This numeric resource specifies the column at which preamble entries are aligned. This value defaults to 11.

print.align.comment This numeric resource specifies the column at which comment entries are aligned.[2] This value defaults to 10.

[2]This is mainly obsolete now since comments do not have to follow any syntactic restriction.

print.comma.at.end This Boolean resource determines whether the comma between fields should be printed at the end of the line. If it is off then the comma is printed just before the field name. In this case the alignment given by print.align determines the column of the comma.

print.equal.right This Boolean resource specifies whether the = sign in normal entries is aligned right. If turned off then the = sign is flushed left to the field name. This value defaults to on.

print.newline This numeric resource specifies the number of newlines between entries. This value defaults to 1.

print.terminal.comma This Boolean resource specifies whether a comma should be printed after the last record of a normal entry. This contradicts the rules of BibTeX but might be useful for other programs. This value defaults to off.

print.use.tab This Boolean resource specifies if the TAB character should be used for indenting. This use is said to cause portability problems. Thus it can be disabled. If disabled then the appropriate number of spaces are inserted instead. This value defaults to on.

print.wide.equal This Boolean resource determines whether the equality sign should be forced to be surrounded by spaces. Usually this resource is off which means that no spaces are required around the equality sign and they can be omitted if the alignment forces it.

suppress.initial.newline This Boolean resource suppresses the initial newline before normal records since this might be distracting under certain circumstances.

The resource values described above are illustrated by the following examples. First we look at a string entry.

```
                    print.align.string                          print.line.length
```

Next we look at an unpublished entry. It has a rather long list of authors and a long title. It shows how the lines are broken.

```
                    print.align.key
@Unpublished{       unpublished-key,
  author          = "First A. U. Thor and Seco N. D. Author and Third A. Uthor
                    and others",
  title           = "This is a rather long title of an unpublished entry which
                    exceeds one line",
  note            = "Some useless comment"
}
  print.indent      print.align                                 print.line.length
```

The field names of an entry are usually printed in lower case. This can be changed with the resource new.field.type. The argument of this resource is an equation where left of

the '=' sign is the name of a field and on the right side is it's print name. They should only contain allowed characters.

```
new.field.type { author = AUTHOR }
```

This feature can be used to rewrite the field types. Thus it is completely legal to have a different replacement text than the original field:

```
new.field.type { OPTauthor = Author }
```

String names are used case insensitive by BibTeX. BibTool normalizes string names before printing. By default string names are translated to lower case. Currently two other types are supported: translation to upper case and translation to capitalized case, i.e. the first letter upper case and the others in lower case.

The translation is controlled by the resource symbol.type. The value is one of the strings `lower`, `upper`, and `cased`. The resource can be set as in

```
symbol.type = upper
```

The macro names are passed through the same normalization apparatus as field types. Thus you can force a rewriting of macro names with the same method as described above. You should be careful when choosing macro names which are also used as field types.

The reference key is usually translated to lower case letters unless a new key is generated (see section A.9). In this case the chosen format determines the case of the key. Sometimes it can be desirable to preserve the case of the key as given (even so BibTeX does not mind). This can be achieved with the Boolean resource preserve.key.case. Usually it is turned off (because of backward compatibility and the memory used for this feature). You can turn it on as in

```
preserve.key.case = on
```

If it is turned on then the keys as they are read are recorded and used when printing the entries. The internal comparisons are performed case insensitive. This is not influenced by the resource preserve.key.case. Especially this holds for sorting which does not recognize differences in case.

Summary

Option	*Resource command*	*Description*
	new.entry.type{*type*}	Define a new entry type *type*.
	new.field.type{*type*}	Define a new field type *type*.
	pass.comments=*on*	Do not discard comments but attach them to the entry following them.
	preserve.key.case=*on*	Do not translate keys to lower case when reading.
	print.align.comment=*n*	Align comment entries at column *n*.
	print.align.key=*n*	Align the key of normal entries at column *n*.
	print.align.string=*n*	Align the = of string entries at column *n*.
	print.align=*n*	Align the = of normal entries at column *n*.
	print.comma.at.end=*on*	Put the separating comma at then end of the line instead of the beginning.
	print.indent=*n*	Indent normal entries to column *n*.
	print.line.length=*n*	Break lines at column *n*.
	print.print.newline=*n*	Number of empty lines between entries.
	print.use.tab=*on*	Use the TAB character to compress multiple spaces.
	print.wide.equal=*off*	Force spaces around the equal sign.
	suppress.initial.newline=*on*	Suppress the initial newline before normal records.
	symbol.type=*type*	Translate symbols according to *type*: upper, lower, or cased.

A.6. Sorting

The entries can be sorted according to a certain sort key. The sort key is by default the reference key. Sorting can enabled with the command line switches `-s` and `-S` as in

```
bibtool -s
```

```
bibtool -S
```

The first variant sorts in ascending ASCII order (including differentiation of upper and lower case). The second form sorts in descending ASCII order. The same effect can be achieved with the Boolean resource values sort and sort.reverse respectively.

```
sort = {on}
sort.reverse = {on}
```

The resource sort determines whether or not the entries should be sorted. The resource sort.reverse determines whether the order is ascending (off) or descending (on) ASCII

order of the sort key. The sort key is initialized from the reference key if not given otherwise.

Alternatively the sort key can be constructed according to a specification. This specification can be given in the same way as a specification for key generation. This is described in section A.9 in detail.

The associated resource name is sort.format. Several formats are combined as alternatives.

```
sort.format = {%N(author)}
sort.format = {%N(editor)}
```

Those two lines are equivalent with the single resource

```
sort.format = {%N(author) # %N(editor)}
```

This means that the sort key is set to the (normalized) author names if an author is given. Otherwise it tries to use the normalized editor name. If everything fails the sort key is empty.

Let us reconsider the unprocessed example on page 8. Without any sort.format instructions this entry would sorted in under "article-full". With the sort.format given above it would be sorted in under "Aamport.LA".

Note that in ASCII order the case is important. The uppercase letters all come before the lowercase letters.

Usually the keys are folded to lower case during the normalization. Thus the lower case variants are also used for comparison. The resource preserve.key.case can be used to print cased keys as they are encountered in the input file. This feature can be combined with the Boolean resource sort.cased to achieve sorting according to the unfolded reference key:

```
preserve.key.case = {on}
sort.cased = {on}
```

Beside the normal entries the macros (string entries) are sorted. This happens in per default. The resource sort.macros can be used to turn off this feature as in

```
sort.macros = {off}
```

An example of sorting can be seen in section 1.2.1 on page 7.

Summary

Option	*Resource command*	*Description*
-S		Enable sorting of entries in reverse sorting order.
-s	*sort*	Enable sorting of entries.
	sort.cased=on	Use the cased form of the reference key for sorting.
	sort.format{spec}	Add disjunctive branch *spec* to the sort key specifier.
	sort.macros=off	Turn off the sorting of string entries.
	sort.reverse=on	Reverse the sorting order.

A.7. Regular Expression Matching

BibTool makes use of the GNU regular expression library. Thus a short excursion into regular expressions is contained in this manual. Several examples of the application of regular expressions can be found also in other sections of this manual.

A concise description of regular expressions is contained in the document regex-0.12/regex.texi contained in the BibTool distribution. In any cases of doubt this documentation is preferable. The remainder of this section contains a short description of regular expressions.

Note that the default regular expressions of the Emacs style are used.

Ordinary characters match only to themselves or their upper or lower case counterpart. Any character not mentioned as special is an ordinary character. Among others letters and digits are ordinary characters.

For instance the regular expression *abc* matches the string *abc*.

The period (.) matches any single character.

For instance the regular expression *a.c* matches the string *abc* but it does not match the string *abbc*.

The star (*) is used to denote any number of repetitions of the preceding regular expression. If no regular expression precedes the star then it is an ordinary character.

For instance the regular expression *ab*c* matches any string which starts with a followed by an arbitrary number of b and ended by a c. Thus it matches *ac* and *abbbc*. But it does not match the string *abcc*.

The plus (+) is used to denote any number of repetitions of the preceding regular expression, but at least one. Thus it is the same as the star operator except that the empty string does not match. If no regular expression precedes the plus then it is an ordinary character.

For instance the regular expression *ab+c* matches any string which starts with a followed by one or more b and ended by a c. Thus it matches *abbbc*. But it does not match the string *ac*.

The question mark (`?`) is used to denote an optional regular expression. The preceding regular expression matches zero or one times. If no regular expression precedes the question mark then it is an ordinary character.

For instance the regular expression *ab?c* matches any string which starts with a followed by at most one b and ended by a c. Thus it matches *abc*. But it does not match the string *abbc*.

The bar (`\|`) separates two regular expressions. The combined regular expression matches a string if one of the alternative separated by the bar does.

Note that the bar has to be preceded by a backslash.

For instance the regular expression *abc\ |def* matches the string *abc* and the string *def*.

Parentheses (`\(\)`) can be used to group regular expressions. A group is enclosed in parentheses. It matches a string if the enclosed regular expression does.

Note that the parentheses have to be preceded by a backslash.

For instance the regular expression *a\(b\ | \d)c* matches the strings *abc* and *adc*.

The dollar (`$`) matches the empty string at the end of the string. It can be used to anchor a regular expression at the end. If the dollar is not the end of the regular expression then it is an ordinary character.

For instance the regular expression *abc$* matches the string *aaaabc* but does not match the string *abcdef*.

The hat (`^`) matches the empty string at the beginning of the string. It can be used to anchor a regular expression at the beginning. If the hat is not the beginning of the regular expression then it is an ordinary character. There is one additional context in which the hat has a special meaning. This context is the list operator described below.

For instance the regular expression *^abc* matches the strings *abcccc* but does not match the string *aaaabc*.

The brackets (`[]`) are used to denote a list of characters. If the first character of the list is the hat (`^`) then the list matches any character not contained in the list. Otherwise it matches any characters contained in the list.

For instance the regular expression *[abc]* matches the single letter strings *a*, *b*, and *c*. It does not match *d*.

The regular expression *[^abc]* matches any single letter string not consisting of an a, b, or c.

The backslash (\) is used for several purposes. Primarily it can be used to quote any special character. Thus if a special character is preceded by the backslash then it is treated as if it were an ordinary character.

If the backslash is followed by a digit d then this construct is the same as the d^{th} matching group.

For instance the regular expression *(an)\1as* matches the string *ananas* since the first group matches *an*.

If the backslash is followed by the character `n` then this is equivalent to entering a newline.

If the backslash is followed by the character `t` then this is equivalent to entering a single `TAB` character.

A.8. Selecting Items

A.8.1. Extracting by `aux` Files

BibTool includes a module to extract BibTeX entries required for a document. This is accomplished by analyzing the `aux` file of the document. The `aux` file is usually produced by LaTeX. It contains the information which BibTeX files and which references are used in the document. Only those entries mentioned in the `aux` file are selected for printing. Since the BibTeX files are already named in the `aux` file it is not necessary to specify an input file.

To use an `aux` file the command line option `-x` can be given. This option is followed by the name of the `aux` file.

```
bibtool -x file.aux
```

Multiple files can be given this way. As always the same functionality can be requested with a resource. The resource extract.file can be used for this purpose.

```
extract.file {file.aux}
```

A small difference exists between the two variants. the command line option automatically sets the resource print.all.strings to `off`. This has to be done in the resource file manually.

Note that the extraction automatically respects the cross-references in the selected entries. Thus you will get a complete BibTeX file—unless some references can not be resolved and an error is produced.

One special feature of BibTeX is supported. If the command `\nocite{*}` is given in the LaTeX file then all entries of the bibliography files are included in the bibliography. The same behavior is imitated by the extracting mechanism of BibTool.

An example of extracting can be seen in section 1.2.4 on page 10.

A.8.2. Extracting with Sub-string Matching

The simplest way of specifying an entry—except by giving its key—is to give a string which has to be present in one of the fields or pseudo-fields. The resource select.by.string can be used to store a selection rule which is applied at the appropriate time later on. If several rules are supplied then any entry matching one of the rules is selected. Thus different rules act as alternatives. This includes rules with regular expressions as described in section A.8.3.

The simplest form of the resource select.by.string is to specify a single string to search for. This string has to be enclosed in double quotes. Since the argument of the resource has to be enclosed in braces we get the following funny syntax:

```
select.by.string {"some string"}
```

This operation selects all entries containing `some string` in one of the normal fields. The search can be restricted to specific fields or extended to pseudo-fields by specifying those fields before the search string. An arbitrary number of white-space separated fields can be given there. Thus the general syntax for this resource is as follows:

```
select.by.string {field_1 ... field_n "string"}
```

To make this selection operation more flexible it is possible to determine whether or not the comparison against the value of a field is performed case sensitive. This can be done with the Boolean resource select.case.sensitive. Since the selection is performed after all resources have been read the value of this resource is only considered then. Thus it is not possible to mix case sensitive and non case sensitive selections as with regular expressions (see section A.8.3).

During the matching of the search string against the value of a field BibTool ignores certain characters. Thus it is possible to hide irrelevant details like braces or spaces. The characters to ignore are stored in the resource select.by.string.ignored. As a default the following resource command is performed implicitly:

```
select.by.string.ignored {"{} []"}
```

As for the resource select.case.sensitive the evaluation of the resource select.by.string.ignored is performed just before the comparisons are carried out. Thus it is not possible to use several rules with different ignored sets of characters.

In addition to the functionality described above the resource select.by.non.string can be used to select all entries for which the match against the given field fails. The general form is the same as for select.by.string:

select.by.non.string {*field*$_1$ ... *field*$_n$ "*string*"}

Note Cross-references are not considered unless select.crossrefs is set.

A.8.3. Extracting with Regular Expressions

Another selecting mechanism uses regular expressions to select items. This feature can be used in addition to the selection according to aux files. The regular expression syntax is identical to the one used in GNU Emacs. For a description see section A.7.

The resource select allows to specify which fields should be used to select entries. The general form is as follows:

select {*field*$_1$ ... *field*$_n$ "*regular_expression*"}

If no field is specified then the regular expression is searched in each field. If no regular expression is specified then any value is accepted; i.e. the regular expression `"."` is used.

Any number of selection rules can be given. An entry is selected if one of those rules selects it. The select rule selects an entry if this entry has a field named *field* which has a sub-string matching *regular_expression*. The field can be missing in which case the regular expression is tried to match against any field in turn.

The pseudo fields `$key`, `$type`, and `@`*type* can be used to access the key and the type of the entry. See page 50 for details. The routines used there are the same as those used here.

Analogously to the negation of the string matching the regular expression matching can be negated. The resource to perform this functionality is select.non. The general form is

select.non {*field*$_1$ ... *field*$_n$ "*regular_expression*"}

The Boolean resource select.case.sensitive can be used to determine whether the selection is performed case sensitive or not:

select.case.sensitive = {off}

Note that the selection does not take place immediately. Instead all selection rules are collected and the selection is performed at an appropriate time later on. The different selection rules are treated as alternatives. Thus any entry which matches at least one of the rules is selected. Nevertheless the value of the resource select.case.sensitive is used which is in effect when the selection rule is issued. Thus it is possible to mix case sensitive rules with non-case sensitive rule.

A regular expression can be specified in the command line using the option `-X` as in

```
bibtool -X regular_expression
```

The fields compared against this regular expression are given in the string valued resource select.fields. Initially this resource has the value `$key`. In general the value is a list of fields and pseudo fields to be considered. The elements of the list are separated by spaces. If the list is empty then all fields and the key are considered for comparison.

Thus the following setting means that the regular only the fields `author` and `editor` are considered when doing a selection.

```
select.fields = {"author editor"}
```

Without changing the resource select.fields the command line given previously is equivalent to the (longer) command

```
bibtool -- select{$key "regular_expression"}
```

Note that the resources select.case.sensitive and select.fields are used for all regular expressions following their definition until they are redefined. This means that it is possible to specify that some comparisons are done case sensitive and others are not done case sensitive.

Finally the resource extract.regex can be used as in

```
extract.regex = {regular_expression}
```

This is equivalent to specifying a single regular expression to be matched against the key. This feature is kept for backward compatibility only. It is not encouraged and will vanish in a future release.

Note Cross-references are not considered unless select.crossrefs is set.

A.8.4. Extracting and Cross-references

When extracting entries due to contained sub-strings or regular expression matching cross-references are not considered automatically. This behavior can result in incomplete and thus incorrect BibTeX files.

The automatic selection of cross-referenced entries can be controlled by the resource select.crossrefs. This resource is `off` by default. This means that cross-references are ignored.

The following instruction can be used to turn on the automatic inclusion of cross-referenced entries:

```
select.crossrefs = on
```

The depth of cross-reference chains can be restricted with the resource crossref.limit. This numeric value limits the depth of the crossreferences. If the actual depth is greater

than this value then the cross-referencing is terminated artifically. The default value is 32.

```
crossref.limit = 42
```

Summary

Option	*Resource command*	*Description*
`-x`	*extract.file*{*file*}	Extract the entries from an `aux` file.
	extract.regexp{*expr*}	Discouraged backward compatibility command.
`-X` *regex*	*select*{*spec*}	Select certain entries according to a regular expression.
	select.by.non.string{*spec*}	Select certain entries according to a failing sub-string matching.
	select.by.string{*spec*}	Select certain entries according to a sub-string matching.
	select.by.string.ignored{*chars*}	Define the class of characters to be ignored by the sub-string matching.
	select.case.sensitive=*off*	Turn off the case insensitive comparison.
`-c`	*select.crossrefs*=*on*	Turn on the additional selection of cross-referenced entries.
	select.fields{*fields*}	Determine fields for `-X`.
	select.non{*spec*}	Select certain entries according to a failing regular expression matching.

A.9. Key Generation

The key generation facility provides a mean to uniformly replace the reference keys by generated values. Some algorithms are hardwired, namely the generation of short keys or long keys either unconditionally or only when they are needed. Additionally a free formatting facility is provided. This can be used to specify your own algorithm to generate keys. The generation of new keys can be enabled using the command line option `-f` in the following way:

```
bibtool -f format
```

This command adds format disjunctively to the formatting instructions already given. The same effect can be achieved with the resource key.format.

```
key.format = {format}
```

Some values of *format* have a special meaning. Fixed formatting rules are used when one of them is in effect. The special values are described below. To illustrate their results we consider the following BibTeX database entries:

```
@Unpublished{unpublished-key,
  author    = "First A. U. Thor and Seco N. D. Author and Third A. Uthor
               and others",
  title     = "This is a rather long title of an unpublished entry which
               exceeds one line",
  ...
}
@Article{,
   author = {L[eslie] A. Aamport},
   title = {The Gnats and Gnus Document Preparation System},
   ...
}
@BOOK{whole-collection,
   editor = "David J. Lipcoll and D. H. Lawrie and A. H. Sameh",
   title = "High Speed Computer and Algorithm Organization",
   ...
}
@MISC{misc-minimal,
   key = "Missilany",
   note = "This is a minimal MISC entry"
}
```

short If a field named `key` is present then its value is used. Otherwise if an author or editor field are present, then this field is used. The short version uses last names only. Afterwards a title or booktitle field is appended, after the fmt.name.title separator has been inserted. Finally if all else fails then the default key default.key is used. The result is disambiguated (cf. key.base).

To see the effect we apply BibTool to the example entries given earlier with the command line argument `-- key.format=short`. This results in the following keys (remaining lines skipped):

```
@Unpublished{     thor.author.ea:this,
@Article{         aamport:gnats,
@Book{            lipcoll.lawrie.ea:high,
@Misc{            missilany,
```

long The long version acts like the short version but incorporates initials when formatting names.

If BibTool is applied to the example entries given earlier with the command line argument `-- key.format=long` we get the following keys:

```
@Unpublished{     thor.fau.author.snd.ea:this,
@Article{         aamport.la:gnats,
@Book{            lipcoll.dj.lawrie.dh.ea:high,
@Misc{            missilany,
```

new.short This version formats like short but only if the given key field is empty. This is obsoleted by the resource preserve.keys and will be withdrawn in a future release.

If BibTool is applied to the example entries given earlier with the command line argument `-- key.format=short.need` we get the following keys:

```
@Unpublished{     unpublished-key,
@Article{         aamport:gnats,
@Book{            whole-collection,
@Misc{            misc-minimal,
```

new.long This version formats like long but only if the given key field is empty. This is obsoleted by the resource preserve.keys and will be withdrawn in a future release.

If BibTool is applied to the example entries given earlier with the command line argument `-- key.format=short.need` we get the following keys:

```
@Unpublished{     unpublished-key,
@Article{         aamport.la:gnats,
@Book{            whole-collection,
@Misc{            misc-minimal,
```

empty The empty version clears the key entirely. The result does not conform to the BibTeX syntax rules. This feature can be useful if a resource file must be used which generates only new keys. In this case a first pass can clear the keys and the given resource file can be applied in a second pass to generate all keys.

If BibTool is applied to the example entries given earlier with the command line argument `-- key.format=empty` we get the following keys:

```
@Unpublished{     ,
@Article{         ,
@Book{            ,
@Misc{            ,
```

In contrast to the command line option, the resource instruction only modifies the formatting specification. The key generation has to be activated explicitly. This can be done using the command line option `-F` as in

```
bibtool -F
```

Alternatively the Boolean resource key.generation can be used in a resource file:

```
key.generation = on
```

Usually all keys are regenerated. This can have the unpleasant side-effect to invalidate citations in old documents. For this situation the resource preserve.keys is meant. This resource is usually `off`. If it is turned `on` then only those entries receive new keys if they do not have a key already. This means that the input contains only a sequence of white-space characters (which is not accepted by BibTeX) as in the following example:

```
@Article{,
   author = {L[eslie] A. Aamport},
   title = {The Gnats and Gnus Document Preparation System},
   journal = {\mbox{G-Animal's} Journal},
   year = 1986,
   volume = 41,
   number = 7,
   pages = "73+",
   month = jul,
   note = "This is a full ARTICLE entry",
}
```

Even if preserve.keys is on, BibTool still changes all keys to lower case by default. This can be suppressed by switching preserve.key.case to on (see section A.5).

When the key.format is not empty then the keys are disambiguated by appending letters or numbers. Thus there can not occur a conflict which would arise when two entries have the same key. The disambiguation uses the resource key.number.separator. If a key is found (during the generation) which is already been used then the valid characters from the value of this resource is appended. Additionally a number is added. The appearance of the number can be controlled with the resource key.base. This resource can take the values upper, lower, and digit. The effect can be seen in the following table:

generated key	digit	lower	upper
key	key	key	key
key	key*1	key*a	key*A
key	key*2	key*b	key*B
key	key*3	key*c	key*C
key	key*4	key*d	key*D

As we have seen there are options to adapt the behavior of formatting. Before we explain the free formatting specification in section A.10 we will present the formatting options. Those options can be activated from a resource file or with the corresponding feature to specify resource instructions on the command line.

preserve.keys This Boolean resource determines whether existing keys should be left unchanged when new keys are generated. The default value is `off`.

preserve.key.case This Boolean resource determines whether keys should be recorded and used exactly as read as opposed to normalizing them by translating all uppercase letters to lower case. The default value is `off`.

default.key The value of this resource is used if nothing else fits. The default value is `**key*`.

key.base The value of this resource is used to determine the kind of formatting the disambiguating number. Possible values are upper, lower, and digit. Uppercase letters, lower case letters, or digits are used respectively.

key.number.separator The value of this resource is used to separate the disambiguating number from the rest of the key. The default value is `*`.

key.expand.macros The value of this Boolean resource is used to indicate whether macros should be expanded while generating a key. The default value is `off`.

fmt.name.title The value of this resource is used by the styles short and long to separate names and titles. The default value is `:`.

fmt.title.title The value of this resource is used to separate words inside titles. The default value is `:`.

fmt.name.name The value of this resource is used to separate different names (where the BibTeX file has `and`) when formatting names. The default value is `..`

fmt.inter.name The value of this resource is used to separate parts of multi-word names when formatting names. The default value is `-`.

fmt.name.pre The value of this resource is used to separate names and first names when formatting names. The default value is `..`

fmt.et.al The value of this resource is used to format `and others` parts of a name list. The default value is `.ea`.

fmt.word.separator The value of this resource is used as additional characters not to be considered as word constituents. Word separators are white-space and punctuation characters. Those can not be redefined. The default value is empty.

The key style short can be formulated in terms of the format specification given in section A.10 as follows:

```
{
  { %-2n(author)
  # %-2n(editor)
  }
  { %s($fmt.name.title) %-1T(title)
  # %s($fmt.name.title) %-1T(booktitle)
  #
  }
}
#
{ { %s($fmt.name.title) %-1T(title)
  # %s($fmt.name.title) %-1T(booktitle)
  }
}
# %s($default.key)
```

The syntax and meaning of such format specifications is explained in section A.10.

A.9.1. Aliases for Renamed Entries

BibTool provides a means to automatically generate `@Alias` definitions for those entries whoich have received a new key during the key generation. This works for a sufficiently current BibTeX only.

The aliases can be requested with the boolean resource key.make.alias. This can be set in a resource file file thie:

```
key.make.alias = on
```

The default is `off`. This means that no additional entries are created.

Summary

Option	*Resource command*	*Description*
	preserve.keys=off	Do not generate new keys if one is already present.
	preserve.key.case=on	Do not translate keys to lower case when reading.
	default.key={key}	Key used if nothing else applies.
	fmt.et.al={ea}	String used to abbreviate further names.
	fmt.inter.name={s}	String used between parts of names.
	fmt.name.name={s}	String used between names.
	fmt.name.pre={s}	String separating first and last names.
	fmt.name.title={s}	String used to separate names from titles.
	fmt.title.title={s}	String used to separate words in titles.
	key.base={base}	Kind of numbers or letters for disambiguating keys.
	key.expand.macros=off	Turn off macro expansion for key generation.
-f	*key.format{fmt}*	Set the specification for key generation to *fmt*.
-F	*key.generation=on*	Turn on key generation.
	key.make.alias=on	Turn on creation of `@Alias` entries for entries which have received a new key.
	key.number.separator={s}	String to be used before the disambiguating number.

A.10. Format Specification

A.10.1. Constant Parts

The simplest component of a format is a constant string. Such strings are made up of any character except white-space and the following ten characters

```
"  #  %  '  (  )  ,  =  {  }
```

This choice of special characters is the same as the special characters of BibTeX. Since no means is provided to include a special character into a format string we guarantee that the resulting key string is conform to the BibTeX rules.

For example the following strings are legal constant parts of a format:

```
Key
the_name.of-the-@uthor-is:
```

Now we come to explain the meaning of the special characters. The first case consists of the white-space characters. They are simply ignored. Thus the following format strings are equal:[3]

```
Author Or Editor
AuthorOrEditor
A u t h o r   O r   E d i t o r
```

A.10.2. Formatting Fields

The next component of formats are made up formatting instructions which are starting with a `%` character. The general idea has been inspired by formatting facilities of C. Since there are several different types of information in a BibTeX entry we provide several primitives for formatting. The simplest form is for instance

```
%N(author)
```

The `%` character is followed by a single character—here `N`—which indicates the way of formatting and the name of the field to be formatted enclosed in parenthesis. The example above requests to format the field `author` according to formatting rules for names (`N`).

The general form is

%sign pre.post qualifier letter`(`*field*`)`

In this specification *sign* is `+` or `-`. `+` means that all characters will be translated to upper case. `-` means that all characters will be translated to lower case. If no sign is given, the case of the field is preserved.

pre and *post* are positive integers whose meaning depends on the format letter *letter*. *qualifier letter* is a one letter specification indicating the desired formatting type optionally preceded by the qualifier `#`. Possible values are as described in the following list:

`p` Format names according to the format specifier number *post*. In a list of names at most *pre* names are used. If there are more names they are treated as given as `and others`.

pre defaults to 2 and *post* defaults to 0.

See section A.10.10 for a description of how to specify name formats.

Example

[3] Well, this is not the whole truth. Internally it makes a difference whether there is a space or not. In the presence of spaces more memory is used. But you shouldn't worry too much about this.

```
author = {A. U. Thor and S. O. Meone and others}
```

With the above item we get the following results:

```
%p(author)     Thor.Meone.ea
%1p(author)    Thor.ea
%-2p(author)   thor.meone.ea
%+1p(author)   THOR.EA
```

`n` Format last names only.
In a list of names at most *pre* last names are used. If there are more names they are treated as given as `and others`. If *post* is greater than 0 then at most *post* characters per name are used. Otherwise the whole name is considered.

pre defaults to 2 and *post* defaults to 0.

This is the same as using the `p` format specifier with the post value of 0. The *post* value of the `n` specifier is used as the *len* value of the first item of the name format specifier. (See also section A.10.10)

Example

```
author = {A. U. Thor and S. O. Meone and others}
```

With the above item we get the following results:

```
%n(author)     Thor.Meone.ea
%1n(author)    Thor.ea
%-2n(author)   thor.meone.ea
%+1n(author)   THOR.EA
%.3n(author)   Tho.Meo.ea
```

`N` Format names with last names and initials.
In a list of names at most *pre* last names are used. If there are more names they are treated as given as `and others`. If *post* is greater than 0 then at most *post* characters per name are used. Otherwise the whole name is considered.

pre defaults to 2 and *post* defaults to 0.

This is the same as using the `p` format specifier with the post value of 1. The *post* value of the `n` specifier is used as the *len* value of the first item of the name format specifier. (See also section A.10.10)

Example

```
author = {A. U. Thor and S. O. Meone and others}
```

With the above item we get the following results:

```
%N(author)     Thor.AU.Meone.SO.ea
%1N(author)    Thor.AU.ea
%-2N(author)   thor.au.meone.so.ea
%+1N(author)   THOR.AU.EA
%.3N(author)   Tho.AU.Meo.SO.ea
```

`d` Format a number, e.g. a year.

The $post^{th}$ number in the field is searched. At most *pre* digits—counted from the right—are used. For instance the field `"june 1958"` formatted with `%2d` results in `58`.

pre defaults to a large number except in when the negative sign is present. Then it defaults to 1.

post defaults to 1. Thus if you want to select the second number you can simply use `%.2d` as format specifier.

If no number is contained in the field then this specifier fails. Thus the specifier `%0d` can be used to check for a number.

Positive and negative signs make no sense in specifying translations since numbers have no uppercase or lowercase counterparts. Thus they have a different meaning in this context.

If the positive sign is given then the specifier does not fail at all. Instead of failing a single `0` is used.

If the negative sign is given then the result is padded with `0` if required. In this case the specifier does not fail at all. Even if no number is found then an appropriate number of `0`s is used.

Example

```
pages = {89--123}
```

With the above item we get the following results:

`%d(pages)`	`89`
`%1d(pages)`	`9`
`%4d(pages)`	`89`
`%-4d(pages)`	`0089`
`%-5.2d(pages)`	`00123`
`%.3d(pages)`	*fails*
`%+.3d(pages)`	`0`
`%0d(pages)`	*succeeds with empty result*

`D` Format a number.
This format specifier acts like the `d` specifier except that the number is not truncated. Thus a large number comes out complete and not only the last few digits.

Example

```
pages = {89--123}
```

With the above item we get the following results:

```
%D(pages)       89
%1D(pages)      89
%4D(pages)      89
%-4D(pages)     0089
%-5.2D(pages)   00123
%.3D(pages)     fails
%+.3D(pages)    0
%0D(pages)      89
```

`s` Take a field as is (after translation of special characters).
At most *pre* characters are used.

pre defaults to a large number.

Example

```
author = {A. U. Thor and S. O. Meone and others}
```

With the above item we get the following results:

```
%s(author)     A.-U.-Thor-and-S.-O.-Meone-and-others
%8s(author)    A.-U.-Th
%-8s(author)   a.-u.-th
%+8s(author)   A.-U.-TH
%0s(author)    succeeds with empty result
```

`T` Format sentences. Certain words are ignored.
At most *pre* words are used. The other words are ignored. If *pre* is 0 then no artificial limit is forced. If *post* is positive then at most *post* letters of each word are considered. Otherwise the complete words are used.

New words to be ignored can be added with the resource ignored.word.

pre defaults to 1 and *post* defaults to 0.

Example

```
title = {The Whole Title}
```

With the above item we get the following results:

```
%T(title)      Whole
%2T(title)     Whole-Title
%2.1T(title)   W-T
%-T(title)     whole
%+T(title)     WHOLE
```

The string to be formatted according to this specification is separated into words. To accomplish this white-space characters and punctuation characters are considered to be not part of a word but as separator. To add additional word separators

use the resource fmt.word.separator. In the following example the characters +, -, <, =, >, *, and / are declared as additional word separators.

```
fmt.word.separator =  "+-<=>*/"
```

Note that the effect of fmt.word.separator is accumulating more characters. It is not possible to define a character not to be a word separator once it has this property.

`t` Format sentences. In contrast to the format letter `T` no words are ignored.
At most *pre* words are used. The other words are ignored. If *pre* is 0 then no artificial limit is forced. If *post* is positive then at most *post* letters of each word are considered. Otherwise the complete words are used.

pre defaults to 1 and *post* defaults to 0.

Example

```
title = {The Whole Title}
```

With the above item we get the following results:

```
%t(title)     The
%2t(title)    The-Whole
%2.1t(title)  T-W
%-t(title)    the
%+t(title)    THE
```

`W` Format word lists.
This specifier acts like `T` except that nothing is inserted between words.

Example

```
title = {The Whole Title}
```

With the above item we get the following results:

```
%W(title)     Whole
%2W(title)    WholeTitle
%2.1W(title)  WT
%-W(title)    whole
%+W(title)    WHOLE
```

`w` Format word lists.
This specifier acts like `t` except that nothing is inserted between words.

Example

```
title = {The Whole Title}
```

With the above item we get the following results:

```
%w(title)      The
%2w(title)     TheWhole
%2.1w(title)   TW
%-w(title)     the
%+w(title)     THE
```

`#p` Count the number of names.

If no *sign* is given or the *sign* is `+` then the following rules apply. If the count is less than *pre* or the count is greater than *post* then this specifier fails. Otherwise it succeeds without adding something to the key.

The construction `and others`, which indicates an unspecified number of additional authors, counts as one single author.

If the *sign* is `-` then the specifier succeeds if and only if the specifier without this sign fails. Thus the `-` acts like a negation of the condition.

If post has the value 0 than this is treated like ∞.

If a is the number of names separated by `and` then
`%`*l*`.`*h*`#p` succeeds if and only if $l \leq a \leq h$.
`%-`*l*`.`*h*`#p` succeeds if and only if $l > a$ or $a > h$.

pre and *post* both defaults to 0.

Example

```
author = {A. U. Thor and S. O. Meone and others}
```

With the above item we get the following results:

`%2#p(author)`	*succeeds with empty result*
`%4#p(author)`	*fails*
`%-4#p(author)`	*succeeds with empty result*
`%3.4#p(author)`	*succeeds with empty result*
`%-3.4#p(author)`	*fails*

`#n` Is the same as `#p`.

`#N` Is the same as `#p`.

`#s` Count the number of allowed characters.

If no *sign* is given or the *sign* is `+` then the following rules apply. If the count is less than *pre* or the count is greater than *post* then this specifier fails. Otherwise it succeeds without adding something to the key.

If the *sign* is `-` then the specifier succeeds if and only if the specifier without this sign fails. Thus the `-` acts like a negation of the condition.

If post has the value 0 than this is treated like ∞.

pre and *post* both default to 0.

If a is the number of allowed characters then
`%l.h#s` succeeds if and only if $l \leq a \leq h$.
`%-l.h#s` succeeds if and only if $l > a$ or $a > h$.

Example

```
title = {The Whole Title}
```

With the above item we get the following results:

`%#s(title)`	*succeeds with empty result*
`%13.13#s(title)`	*succeeds with empty result*
`%10.16#s(title)`	*succeeds with empty result*
`%-10.16#s(title)`	*fails*

`#w` Count the number of words. All words are considered as valid. The division into words is performed after deTEXing the field.

If no *sign* is given or the *sign* is `+` then the following rules apply. If the count is less than *pre* or the count is greater than *post* then this specifier fails. Otherwise it succeeds without adding something to the key.

If the *sign* is `-` then the specifier succeeds if and only if the specifier without this sign succeeds. Thus the `-` acts like a negation of the condition.

If post has the value 0 than this is treated like ∞.

pre and *post* both default to 0.

If a is the number of words separated by white-space then
`%l.h#p` succeeds if and only if $l \leq a \leq h$.
`%-l.h#p` succeeds if and only if $l > a$ or $a > h$.

Example

```
title = {The Whole Title}
```

With the above item we get the following results:

`%#w(title)`	*succeeds with empty result*
`%3.3#w(title)`	*succeeds with empty result*
`%1.6#w(title)`	*succeeds with empty result*
`%-1.6#w(title)`	*fails*

`#t` Is the same as `#w`.

`#W` Count the number of words. Certain words are ignored. The ignored words are determined by the resource ignored.word. The division into words is performed after deTEXing the field.

If no *sign* is given or the *sign* is `+` then the following rules apply. If the count is less than *pre* or the count is greater than *post* then this specifier fails. Otherwise it succeeds without adding something to the key.

If the *sign* is `-` then the specifier succeeds if and only if the specifier without this sign fails. Thus the `-` acts like a negation of the condition.

If post has the value 0 than this is treated like ∞.

pre and *post* both default to 0.

If a is the number of words separated by white-space which are not marked to be ignored then
`%`l`.`h`#p` succeeds if and only if $l \leq a \leq h$.
`%-`l`.`h`#p` succeeds if and only if $l > a$ or $a > h$.

Example

```
title = {The Whole Title}
```

With the above item we get the following results:

`%#W(title)`	*succeeds with empty result*
`%2.2#W(title)`	*succeeds with empty result*
`%1.6#W(title)`	*succeeds with empty result*
`%-1.6#W(title)`	*fails*

`#T` Is the same as `#W`.

If some words are enclosed in brace, they are considered as one composed word. For example, with the format `%t(title)`, and this field:

```
title = "{The Whole Title}"
```

In this case we obtain `The-Whole-Title`.

The field specification *(field)* selects the field of the entry to be formatted. As usual in BibTeX the case of the letters is ignored. If the field does not exist then the formatting fails and continues at the next alternative (see below).

But the field is not only sought in the current entry. According to the behavior of BibTeX the special field `crossref` is taken into account. If a field is missing them the entry named in the `crossref` field is also considered. Since this dereferencing contains the potential danger of an infinite loop the number of dereferencing steps is restricted by the numeric resource crossref.limit. The number of uses of the `crossref` field is limited by the value of this resource. The default of this resource is 32.

Usually a value of 1 would be sufficient for BibTeX files conforming to the standard styles. Nevertheless other applications can be imagined where a higher value is desirable.

To turn off the crossref feature complete you can set the value of crossref.limit to 0. In this case only the fields found in the entry itself are considered.

A.10.3. Pseudo Fields

In addition to the ordinary fields of an entry there are several pseudo fields. They are listed below.

`$key` This pseudo field contains the old reference key—before generating a new one. If none has been given then the access fails.

`$sortkey` This pseudo field contains the string according to which the sorting is performed. It defaults to the reference key.

`$default.key` This pseudo field contains the value of the resource default.key similarly the resources fmt.name.title, fmt.title.title, fmt.name.name, fmt.inter.name, fmt.name.pre, and fmt.et.al can be accessed.

`$source` This pseudo field contains the name of the file the entry has been read from. If this file can not be determined, e.g. because the entry has been read from stdin, then this pseudo field is empty.

`$type` This pseudo field contains the type of the entry, i.e. the string following the initial `@` of an BibTeX entry, e.g. `article`. It is always present.

`@type` This pseudo field is matched against the type of the entry. If they are identical (ignoring cases) then the type is returned. Otherwise the access fails.

In an article item the specification `%s(@Article)` succeeds and returns `Article` whereas `%s(@Book)` fails.

`$day` This pseudo field contains the current day as a two digit number or the empty string if this value is not available. The date and time values are determined at the beginning of the BibTool run and does not reflect the execution time used by BibTool.

On some systems the timing function might be missing or returning strange values. In this case the timing fields simply return the empty string.

`$month` This pseudo field contains the current month as a two digit number or the empty string if this value is not available.

`$mon` This pseudo field contains the current month name as a string or the empty string if this value is not available.

`$year` This pseudo field contains the current year as a four digit number or the empty string if this value is not available.

`$hour` This pseudo field contains the current hour as a two digit number or the empty string if this value is not available.

`$minute` This pseudo field contains the current minute as a two digit number or the empty string if this value is not available.

`$second` This pseudo field contains the current second as a two digit number or the empty string if this value is not available.

`$user` This pseudo field contains the contents of the environment variable `$USER` or the empty string if this value is not available. On UN*X systems this variable usually contains the name of the user. This can be used to write logging information into a field.

`$hostname` This pseudo field contains the contents of the environment variable `$HOSTNAME` or the empty string if this value is not available.

A.10.4. Conjunctions

Conjunctions are formatting instructions evaluated in sequence. The conjunctions are simply written by successive formatting instructions. A conjunction succeeds if every part succeeds. The empty conjunction always succeeds.

Suppose an BibTeX entry contains fields for `editor` and `year`. Then the following conjunction succeeds:

```
%-3n(editor) :  %2d(year)
```

If the value of the `editor` field is `"E.D. Itor"` and the `year` field contains `"1992"` then the result is `itor:92`.

A.10.5. If-Then-Else

Depending on the presence of a (pseudo-) field formatting instructions can be issued. This corresponds to an if-then-else statement in a Pascal-like language. The syntax is as follows:

(*field*) {*then-part*} {*else-part*}

If the access to the (pseudo-)field as described in A.10.2 succeeds then the *then-part* is evaluated. Otherwise the *else-part* is evaluated. Both parts may be empty. Nevertheless the braces are required.

Let us look at an example. The following construction can be used to format a field `author` if it is present or print a constant string.

```
(author){%N(author)}{--no-author--}
```

A.10.6. Alternatives

Alternatives (disjunctives) are separated by the hash mark (`#`). The general form is

$alternative_1$ # $alternative_2$ # ... # $alternative_n$

The alternatives are evaluated from left to right. The first one that succeeds terminates the processing of all alternatives with success. If no alternative is successful then the whole construct fails.

An alternative can be empty. The empty alternative succeeds without any other effect.

The example given in subsection A.10.5 can be also written as

```
%N(author) # --no-author--
```

If the author field is accessible the first alternative succeeds and terminates the construct. Otherwise the constant string is used. This constant string always succeeds.

A.10.7. Grouping

Any number of constructs can be enclosed in braces (`{}`) for grouping. Thus the precedence of operators can be bypassed.

Coming back to our example from the previous subsection. To complicate the example we want to append an optional title, or a constant string. This is accomplished as follows.

```
{%N(author) # --no-author-- } {%T(title) # --no-title-- }
```

The grouping allows to restrict the range of the alternative operator `#` in this example.

Another example shows how the alternative together with grouping can be used to share a format specification for certain types of entries:

```
{%0s(@book) # %0s(@proceedings)} --book-or-proc--
```

The `%0s` specifier is used to check for the existence of a certain field without actually adding anything to the output. Other constructs may serve for the same purpose. This construct is applied to the pseudo-fields `@book` and `@proceedings`. The access to the pseudo-field fails if requested in another type of entry. Those two checks are combined to form a disjunction. Thus the following code—the constant in this example—is reached only if we are in a book or in a proceedings entry. It is not reached in an article.

A.10.8. Ignored Words

Certain format specifiers act on lists of words. In this situation it can be desirable to ignore certain words. For instance when a sort key is constructed with the title of books it is common practice to omit certain words like articles. This is accomplished by a list of ignored words. This list is initialized at compile time to contain articles of different languages (If the installer has not modified it).

The resource ignored.word can be used to put additional words onto the list of ignored words. For this purpose the new word is given as argument to the resource. Note that there should be no space between the braces and the word. For example:

```
ignored.word {word}
```

To gain complete control over the list of ignored words you can completely overwrite the compiled in defaults. This can be accomplished by clearing the list of ignored words. Afterwards no word is recognized as ignored word until new words are added to this list. This operation can be performed with the resource clear.ignored.words. In principal this operation does not require any argument. Since this contradicts the syntactic restrictions for resources you have to give an empty argument to this resource:

```
clear.ignored.words {}
```

A.10.9. Expanding TEX/LATEX Macros

When fields are formatted certain LaTeX macros may be replaced by pure text. Each macro not defined is simply ignored. Initially no LaTeX macro is defined. The resource tex.define can be used to define LaTeX macros. The syntax is very close to LaTeX. The simplest form is the following definition.

```
tex.define {macro=replacement text}
```

This resource defines a simple macro which is replaced by the replacement text. This replacement text may in turn contain macros.

In addition to this simple macro also macros involving arguments can be defined. As in LaTeX's `\newcommand` the number of arguments is appended after the macro name.

```
tex.define {macro[arg]=replacement text}
```

The number of arguments may not exceed 9. The actual parameters are addressed by writing `#`n, where n is the number of the argument.

For instance, this feature can be used to ignore certain arguments of macros.

Note that spaces between the macro head and the equality sign (`=`) are ignored. Any unwanted spaces after the equality sign may have strange effects.

Usually the macro name starts with a backslash (\). If the macro name starts with another character then this character is made active (cf. [Knu89]). This feature is especially useful for translating characters with an extended ASCII code ($\geq$ 128) to the appropriate TeX macros.

For instance the following definition forces the expansion of the macro `\TeX` to the string `TeX`.

```
tex.define {\TeX=TeX}
```

Without this definition the title `The \TeX{}book` would result in `book`. With this definition the same title results in `TeXbook`.

Suppose you have an input file containing 8-bit characters (e.g. ISO 8859-1 encoding). The following definition can be used to map this character into a pure ASCII string[4]

```
tex.define {ü=ue}
```

With the following definition the `\protect` macro and the corresponding braces would be ignored when formatting field, otherwise the braces would remain.

```
tex.define {\protect[1]=#1}
```

Some useful definitions can be found in the libraries distributed with BibTool (see also appendix C).

A.10.10. Name Formatting

Names are a complicated thing. BibTool tries to analyze names and "understand" them correctly. According to the BibTeX definition a name consists of four types of components:

- The first names are any names before the last names which start with an upper case letter.
  For instance for the name "Ludwig van Beethoven" the first name is "Ludwig".
- The last name is the last word (or group of words) which does not belong to the junior part.
  For instance for the name "Ludwig van Beethoven" the last name is "Beethoven".
- The von part are the names before the last name which start with lower case letters.
  For instance for the name "Ludwig van Beethoven" the von part consists of the word "van".
- The junior part of a name is an appendix following the last name. BibTool knows only a small number of words that can appear in the junior part: junior, jr., senior, sen., Esq., PhD., and roman numerals up to XXX.

Everything except the last name is optional. Each part can also consist of several words. More on names can be found in [Lam94] and [Pat88b].

BibTool provides a means to specify how the various parts of a name should be used to construct a string. This string can be used as part of a key with the `%p` format specifier (see above).

[4] To add an e is the German convention for umlaut characters.

BibTool uses a small number of name format specifiers.[5] Initially most of them are undefined. The name format specifier 0 is initially set to the value `%*l[`*fmt.inter.name*`]`. The name format specifier 1 is initially set to the value `%*l[`*fmt.inter.name*`]%*1f[`*fmt.inter.name*`]`.

The name format specifiers 0 and 1 are used by the formatting instructions `%N` and `%n`. Thus you should be careful when redefining them. To help you keep an eye on these two name format specifiers BibTool issues a warning when they are modified.

The resource new.format.type can be used to assign values to those name format specifiers:

```
new.format.type {17="%f%v%l"}
```

This instruction sets the name format specifier number 17 to the given value. This value is a string of characters to be used directly. There is only one construct which is not used literally. This construct is started by a % sign optionally followed by a `+` or a `-` and a number. Next comes one of the letters `f`, `v`, `l`, or `j`. Finally there are three optional arguments enclosed in brackets.

Thus the general form looks as follows:

`%`*sign len*`.`*number letter* `[`*pre*`][`*mid*`][`*post*`]`

The letter `f` denotes all first names. The letter `l` denotes all last names. The letter `v` denotes all words in the von part. The letter `j` denotes all words in the junior part.

If *sign* is `+` then the words are translated to upper case. If *sign* is `-` then the words are translated to lower case. If no sign is given then no conversion is performed. If the sign is `*` then the translation is inherited from the calling format.

The number *len* can be used to specify the number of characters to be used. Each word is truncated to at most *len* characters if *len* is greater than 0. Otherwise no truncation is performed. Thus a value of 0 acts like ∞. Note that the length of the name format specifiers 0 and 1 are automatically inherited from the calling format.

The fractional number *number* after the period denotes the number of name parts to be taken into account. This can be used to just show the one first name if more are given.

If `[`*mid*`]` is given then this string is used between several words of the given part. If none is given then the empty string is used.

If `[`*pre*`]` is given then this string is used before the given part, but only if the part is not empty. If none is given then the empty string is used.

If `[`*post*`]` is given then this string is used after the given part, but only if the part is not empty. If none is given then the empty string is used.

Now we can come to an example. Suppose the name field contains the value `Cervantes Saavedra, Miguel de`[6]. This name has two last names, one first name and one word in the von part.

[5] The exact number can be changed in the configuration file before compilation. The default is 128.
[6] This is the author of "Don Quixote"

We want to apply the following name format specifier

```
%1f[.][][.]%1v[.][][.]%3l[-]%1j
```

This means we want to use abbreviation of first name, von and junior part to one letter and of three letters of the last name. Thus we will get the result `M.d.Cer-Saa`.

Note that the name specifier does not take care to include only allowed letters into a key. Thus watch out and avoid special characters as white-space and comma.

A.10.11. Example

To end this section we should have a look at a complete example of key generation specification. For this purpose we define a rule according to which the keys should be generated:

1. If a field named `bibkey` is present then the value of this field should be used.
2. If the type of the entry is a book then the authors/editors are used followed by the year separated by a colon.
3. If the type of the entry is an article in a journal (`article`) then the author, the journal, the number, and the year should be used. Author and journal should be separated by a colon, The journal should be abbreviated with the initials and separated from number and year by a period.
4. If the type of the entry is a volume of conference proceedings (`proceedings`) then the editor, the first 5 initials of the title and the year should be used. The editor should be followed by a colon and the year preceded by a period.
5. If the type of the entry is a contribution in conference proceedings then the author, the initials of the book title and the year should be used.
6. Otherwise the first three letters of the type, the author and the year should be used. If no author is given then the initials of the title should be used instead—but at most 6 characters.

The names should include up to two names abbreviated to four letters and should be translated to lower case. If an information is missing then the respective part together with the following separator should be omitted.

The disambiguation should be done by appending upper case letters without a preceding string. If everything else fails three question marks should be inserted as key.

To implement this scheme we write the following specification into a resource file:

```
key.expand.macros = on
key.base = upper
key.number.separator = {}
key.format =
{
    %s(bibkey)
  #
```

```
    %0w(@book)
    { %-2.4n(author): # %-2.4n(editor): # }
    { %4d(year)       # }
  #
    %0w(@article)
    { %-2.4n(author): # }
    { %-.1W(journal). # }
    { %4d(year)       # }
  #
    %0w(@proceedings)
    { %-2.4n(editor): # }
    { %-.1W(title).   # %-.1W(booktitle). # }
    { %4d(year)       # }
  #
    %0w(@inproceedings)
    { %-2.4n(author):   # }
    { %-.1W(booktitle). # }
    { %4d(year)         # }
  #
    %3s($type)-
    { %-2.4n(author):
    # %-6.1W(title).
    }
    {%4d(year) # }
  #
    %3s($type)-
    %4d(year)
  # ???
}
```

Since each part has been explained before we just need some overall remarks. I prefer to use the backtracking-based disjunctions instead of nested if-then-else constructs because they save some braces. They can be read as a switch statement, or even better as a `cond` statement in Lisp. This means they describe cases. The first successful case terminates the evaluation of the whole cascade.

The constructions like `%0w(@book)` are use to distinguish the different types. This construction does not produce any output. It just succeeds or fails depending on the type of the current entry. The `%0w` could also be replaced by other specifiers which serve the same purpose.

The constructions like `{%4d(year) # }` always succeed. The hash sign (`#`) catches the failure and inserts the second alternative—which happens to be empty—if the requested field does not exist.

Summary

Option	*Resource command*	*Description*
	clear.ignored.words{}	Forget all words from the list of ignored words.
	new.format.type{*n*=*spec*}	Define a new way to format names.
	ignored.word{*s*}	Add a word to the list of ignored words.
	tex.define{*macro*=*text*}	Expand the TEX macro *macro* to *text*.
	tex.define{*macro*[*n*]=*text*}	Expand the TEX macro with arguments.

A.11. Field Manipulation

A.11.1. Adding or Deleting Fields

Certain fields can be added or deleted. This feature can be used to update time stamps. For this purpose it is important to know that deletion is done before addition. It is also important to know that the newly added entries are not rewritten (see next section) even though rewrite rules are applicable.

Two resources are provided to accomplish adding and deleting fields: add.field and delete.field

```
add.field {field=value}
```

This instruction replaces the contents of the field *field* by *value* in each entry. If this field does not exist already then it is added first. The additions are applied in the sequence they are given.

value can contain formatting instructions already introduced in the section A.10.2 about "Formatting Fields" on page 42.

Suppose a time stamp is stored in the field `time`. With these resources the update of a time-stamp can be achieved using the resource instructions

```
add.field {time="June 13, 2000"}
```

If you want to update all time fields to contain the current date the following instruction can be used. It makes use of the pseudo fields (see page 50).

```
add.field {time="%s($mon) %s($day), %s($year)"}
```

If you want to strip the month to three leading letters and the year to two trailing digits this can be achieved with the following instruction:

```
add.field {time="%3s($mon) %s($day), %2d($year)"}
```

The following instruction deletes all fields named *field*:

```
delete.field {field}
```

A.11.2. Field Rewriting

Field modifications can be used to optimize or normalize the appearance of a BibTeX data base. The powerful facility of regular expression matching is used for this purpose as we have already seen in section 1.2.3.

The resource rewrite.rule can be used to specify rewrite rules. The general form is as follows:

```
rewrite.rule {field1 ... fieldn # pattern # replacement_text}
```

field$_1$ ... *field*$_n$ is a list of field names. The rewrite rule is only applied to those fields which have one of those names. If no field name is given then the rewrite rule is applied to all fields.

```
rewrite.rule {pattern # replacement_text}
```

Next there is the separator '#'. This separator is optional. It can also be the equality sign '='.

pattern is a regular expression enclosed in double quotes ("). This pattern is matched against sub-strings of the field value—including the delimiters. If a match is found then the matching sistering is replaced by the replacement text or the field deleted if no replacement text is given.

replacement_text is the string to be inserted for the matching sistering of the field value. The backslash '\' is used as escape character. '\n' is replaced by the n^{th} matching group of *pattern*. n is a single digit (1–9). Otherwise the character following the backslash is inserted.[7] Thus it is possible to have double quotes inside the replacement text.

Other specials are

\$ which is replaced by the key of the current entry.

\@ which is replaced by the type of the current entry.

If no replacement text is given then the whole field is deleted. In fact the instruction delete.field is only an alias for a corresponding rewrite rule with an empty replacement text. This behavior is illustrated in the following abstract examples:

```
rewrite.rule {field # pattern}
rewrite.rule {pattern}
```

More concrete, the rewrite rule

```
rewrite.rule { time # "^{}$" }
```

[7] Future releases may use backslash followed by letters for special purposes. It is not safe to rely on escaping letters.

deletes the time field if the value of the field is empty and enclosed in curly braces. This is checked with the anchored regular expression `^{}$`. The hat `^` matches the beginning of the value and the dollar `$` matches its end. Since nothing is in between—except the field delimiters—the rule is applied only to time fields with empty contents.

This can be generalized to the following rewrite rule which deletes all empty fields using the same mechanism and just omitting the specification of a field name:

```
rewrite.rule { "^{}$" }
```

Note that for a similar kind of rule for double quotes as field delimiters you need to quote these characters with backslashes:

```
rewrite.rule { "^\"\"$" }
```

The replacement text may contain field formatting instructions as described in section A.10.2 on page 42. These field formatting instructions are replaced by their respective values. Thus we could exploit again the time stamp example from above. The following rewrite rule will update an existing time stamp without adding one if none is present:

```
rewrite.rule { time ".*" = "%3s($mon) %s($day), %2d($year)" }
```

The pattern `.*` matches any sequence of arbitrary characters. Thus the old contents of the field is a match. In this example the value is not reused in the replacement text. Thus the old contents is completely replaced by the new one.

Usually the matching is done case insensitive. This means that any upper case letter matches its lower counterpart and vice versa. This behavior is controlled by the Boolean resource rewrite.case.sensitive which is on by default. Changing this variable influences only rewrite rules specified later.

```
rewrite.case.sensitive = off
```

A problem occurs e.g. when a string is replaced by a string containing the original one. To avoid infinite recursion in such cases the numeric resource rewrite.limit controls the number of applications of each rewrite rule. If the number given in rewrite.limit is not negative and this limit is exceeded then a warning is printed and further applications of this rule are stopped. A negative value of the resource rewrite.limit indicates that no limitation should be used.

Next we will investigate some concrete examples. Note that in these examples the character '␣' denotes a single space. It is used to highlight places where spaces have to be used which would be hard to recognize otherwise.

- Empty entries are composed of delimiters — either double quotes or curly braces which enclose an arbitrary number of spaces. If we want to delete empty entries we can use the following two rules.

```
rewrite.rule { "^\"␣*\"$" }
rewrite.rule { "^{␣*}$" }
```

  The caret '^' denotes the beginning of the whole string and the dollar is its end. The star is an operator which says that an arbitrary number of the preceding regular expression — i.e. the space — can occur at this point.

- Ranges of pages should usually be composed of numbers separated by an n-dash (`--`). The next example shows how the pages field can be normalized. Spaces are deleted and a single minus sign is replaced by a double minus.

```
rewrite.rule { pages # "\([0-9]+\)␣*-␣*\([0-9]+\)" = "\1--\2" }
```

- Field rewriting may be used to remove LaTeX commands. This example shows how to remove from titles a `\protect` macro together with the braces, in case the delimiter is a double quote.

```
rewrite.rule { title # "^\"␣*\\␣*protect␣*{\(.*\)}\"$" = "\"\1\"" }
```

A.11.3. Field Ordering

Fields can be reordered within an entry. This feature is controlled by the presence of a specification for the order to use. The order is specified with the resource sort.order. The general form is as follows:

```
sort.order { entry = field1 # field2 # ... }
```

entry is the name of an entry like `book`. The *field*s are an arbitrary number of field names like `author`. This specification says that *field1* should precede *field2* etc. Fields which are not in this list are arranged after the specified ones. The are left in the same order as they appear in the entry.

Another possibility is to specify the entry `*`. Such a sorting order is applicable to any kind of entry. If no specific sort order is found then this general order is used if one has been specified.

Any sorting order is added to a list of sorting orders if it has not been defined before. If a sorting order is specified again, the old one is simply overwritten.

Consider the following part of a resource file:

```
sort.order {* = author # title}
sort.order {misc = author # title # howpublished # year # month # note}
```

This means that the author field goes before the title field in any entry type. For the misc entries additional specifications are made.

The library sort_fld.rsc contains a sample sorting order for the standard entry types.

Summary

Option	*Resource command*	*Description*
	add.field{*field=value*}	Add a new field to each entry.
	delete.field{*field*}	Delete the named field from all entries.
	rewrite.case.sensitive=off	Turn off the case comparison during field rewriting.
	rewrite.rule{*fields#pattern#text*}	Replace in all given fields the pattern by the replacement text.
	sort.order={*entry=f#...#f*}	Specify a preference order for fields in a given entry.

A.12. Semantic Checks

Semantic checks can be enabled in addition to the syntactic checks performed during parsing.

A.12.1. Finding Double Entries

When merging several bibliographic data bases a common problem is the occurrence of doubled entries in the resulting data base. When searching for double entries several problems arise. Which entries should be considered equal and what should happen to double entries.

The first question is answered as follows. Two entries are considered equal if their sort key is identical and they are adjacent in the final output. The first condition of identical sort keys allows the user to specify which criteria should be used when comparing entries. This can be achieved with the resource sort.format (see section A.6). The second condition can easily be achieved by also sorting when requesting checking of doubles.

It remains the question what to do with the doubles. Usually it is not desirable to keep double entries in one data base, so only one entry found is kept. The others are printed as comments, i.e. the initial "@" is replaced by "###". Thus all information is still present but inactive in the BibTeX file. However, further processing with BibTool will remove these entries if pass.comments is off, which is the default.

Sometimes it is not desirable to include deleted entries in the output – not even as comments. In this case the default behavior can be changed with the help of the Boolean resource print.deleted.entries. If this resource is off then deleted entries are suppressed completely.

The prefix for deleted entries is stored in the resource print.deleted.prefix which defaults to "###". Thus it can be redefined. However note that you should avoid using a string ending in an at sign @ since this would undo the effect of deleting an entry.

The Boolean resource check.double.delete can be used to delete double entries completely. For this purpose it has to be turned off as in:

```
check.double.delete = on
```

The resource check.double can be used to turn on the checking of doubles. This feature is turned off initially.

```
check.double = on
```

Checking of doubles can also be turned on with the command line option -d:

```
bibtool -d
```

A.12.2. Regular Expression Checks

The regular expressions (see section A.7) which are used to rewrite fields (see section A.11.2) can also be used to perform semantic checks on fields. For this purpose the resource check.rule is provided. The syntax of check.rule is the same as for rewrite.rule.

```
check.rule { field # pattern # message }
```

Again *field* and *message* is optional. The separator # can also be written as equality sign (=) or omitted.

Each field is processed as follows. Each check.rule is tried in turn until one rule is found where *field* (if given) is identical to the field name and *pattern* matches a sub-string of the field value. If such a rule is found then the *message* is written to the error stream. If no message is given then nothing is printed and processing of the current field is ended.

message is treated like the replacement text in rewrite.rule, Thus the special character combinations described in section A.11.2 are expanded.

Usually the matching is not done case sensitive. This means that any upper case letter matches its lower counterpart and vice versa. This behavior is controlled by the Boolean resource check.case.sensitive which is ON by default. Changing this variable influences only rewrite rules as described in section A.11.2.

```
check.case.sensitive = off
```

Consider the following example. We want to check that the year field contains only years from 1800 to 1999. Additionally we want to allow two digit abbreviations.

```
check.rule { year "^[\"{]1[89][0-9][0-9][\"}]$" }
check.rule { year "^[\"{][0-9][0-9][\"}]$" }
check.rule { year "" "\@ \$:  Year has to be a suitable number" }
```

The first rule matches any number starting with 1 followed by 8 or 9 and finally two digits. The whole number may be enclosed in double quotes or curly braces.[8] The hat at the beginning and the dollar at the end force that the pattern matches against the whole field value only.

The next rule covers years consisting of two digits. The first two rules produce no error message but end the search for further matches. Thus is something suitable is found then one of the first two rules finds it.

Otherwise we have to produce an error message. This is done with the third rule. The empty pattern matches against any value of the year field. This rule is only applied if the preceding rules do not match. In this case we print an error message. `\@` is replaced by the current type and `\$` by the current key.

Summary

Option	*Resource command*	*Description*
	check.case.sensitive=off	Perform semantic checks case sensitive.
`-d`	*check.double=on*	Find and mark or delete entries with identical sort keys.
	check.double.delete=on	Delete double entries instead of deactivating them.
	check.rule{field#pattern#msg}	If the value of field matches pattern then print the given message.

A.13. Strings — also called Macros

Strings in BibTeX files play an important role when managing large bibliographic data bases. Thus the deserve special treatment. If the resource macro.file is defined then the macros are written to this file. The argument is a file name as in

```
macro.file {macro/file/name}
```

[8] In fact the regular expression allows also strings starting with a quote and ending in a curly brace. But this syntactical nonsense is ruled out by the parser already.

Note that the reverse operation to string export namely the import of strings does not deserve special treatment. You can simply give the macro file as one of the input files—preferably before any input file that makes use of one of the macros contained therein.

The Boolean resource print.all.strings indicates if all macros defined in the BibTeX file should be printed or only those macros actually used.

```
print.all.strings = on
```

The appearance of string names is controlled by the resource symbol.type (see 27).

Strings can be expanded when printing entries. This feature of BibTool is controlled by the resource expand.macros as in

```
expand.macros = on
```

The effect is that all known strings in normal entries are replaced by their values. If the values are not defined at the time of expansion then the macro name remains untouched. As a side effect strings concatenations are simplified. Imagine the following BibTeX file.

```
@string{ WGA = " World Gnus Almanac" }

@Book{ almanac-66,
        title =  1967 # WGA,
        month = "1~" # jan
}
```

If BibTool is applied with expand.macros turned on this results in the following output — if the default settings are used for every other resource.

```
@STRING{wga   = " World Gnus Almanac" }

@Book{          almanac-66,
  title       = {1967 World Gnus Almanac},
  month       = {1~} # jan
}
```

The macro WGA has been expanded and merged with 1967. Note that the string jan has not been expanded since the value should be defined in a BibTeX style file (.bst).

When macros are expanded the delimiters of entries are normalized, i.e. only one style is used. In this example braces have been used. The alternative would be to use double quotes. This behavior is controlled by the resource print.braces. If this resource is on then braces are used otherwise double quotes are taken. It can be changed like in

```
print.braces = off
```

The delimiters of the whole entry are recommended to be braces. For compatibility with Scribe it is also allowed that parentheses are used for those delimiters. This behavior

can be achieved with the Boolean resource print.parentheses. Initially this resource is off. It can be set like in the following instruction:

```
print.parentheses = on
```

Summary

Option	*Resource command*	*Description*
-m file	*macro.file={file}*	Write the macro definitions to the file *file*.
	print.all.strings=off	Print only those macro definitions which are used instead of all.
	expand.macros=on	Turn on macro (string) expansion in fields.
	print.braces=off	Switch to the use of quotes for expanded macros instead of braces.
	print.parentheses=on	Enclose the whole entry in parentheses instead of braces.

A.14. Statistics

Some information can be obtained at the end of a BibTool run. The number of BibTeX items read and written is printed. To enable this feature the resources count.all and count.used are provided.

```
count.all = on
```

count.all indicates that all known types of BibTeX items should be listed.

```
count.used = on
```

count.used forces only those types of BibTeX items to be listed which have been found in the input files.

Summary

Option	*Resource command*	*Description*
-#	*count.all=on*	Print statistics about all known entry types.
-@	*count.used=on*	Print statistics about the used entry types only.

A.15. BibTeX1.0 Support

BibTool supports already some of the feature proposed for BibTeX1.0.

A.15.1. Including Bibliographies

The bibliography file may contain an instruction of the following form:

```
@include{abc.bib}
```

Such an entry is stored in the database and printed when requested. Nevertheless the resource apply.include can be used to control this behavior.

A.15.2. Aliases

The bibliography file may contain an instruction of the following form:

```
@alias{abc=def}
```

This means that the key `abc` is treated as an alias for the key `def`. Usually this alias is stored as alias in the database. For old BibTeX files it may be desirable to eliminate aliases and introduce copies of records instead. Nevertheless the resource apply.alias can be used to control this behavior.

A.15.3. Modifications

The bibliography file may contain an instruction of the following form:

```
@modify{key,
  abc = {def}
}
```

This modification is stored in the database without being applied. Nevertheless the resource apply.modify can be used to control this behavior.

Summary

Option	*Resource command*	*Description*
	apply.alias=on	Expand the aliased entries in the database.
	apply.include=on	Include the entries contained in the bibliography file given in `@include`.
	apply.modify=on	apply the modifies in the database.

B. Limitations

B.1. Limits of BibTool

BibTool has been written with dynamic memory management wherever possible. Thus BibTool should be limited by the memory available only. Especially the limitation on the field length which is present in BibTeX 0.99 is not present in BibTool.

BibTeX needs a special order when cross-referenced entries are used. This limitation has also been released in BibTool.

B.2. Bugs and Problems

Problems currently known are the following ones. They are not considered to be bugs.

- The referencing feature of BibTeX is not supported. `\cite` macros can be contained in fields (e.g. notes). Such things can be confused.
- The memory management uses dynamic memory. This memory is reused but not returned to the operating system. Thus BibTool may run out of memory even if a more elaborated memory management may find free memory. This is a design decision and I don't think that I will change it.
- The TeX reading apparatus is only imitated to a certain limit. But this should be enough for most applications to produce satisfactory results.
- In several modules ASCII encoding is assumed. I do not know to which extend this influences the functionality since I don't have access to non-ASCII machines.
- Macro expansion uses a dynamic array which can turn out to be too short. This will be corrected as soon as I have an example where this bug shows up.

The distribution of BibTool also contains a file named ToDo. If you are interested in more detailed descriptions of possible problems, limitations, and ideas for improvements in further releases then you can have a look at the contents of this file.

C. Sample Resource Files

Sample resource files are included in the distribution of BibTool in the directory Lib. Only some of them are reproduced in this section.

C.1. The Default Settings

The following list shows the defaults for all resource instructions.

```
apply.alias               = off
apply.include             = off
apply.modify              = off
bibtex.env.name           = "BIBINPUTS"
check.case.sensitive      = on
check.double              = off
check.double.delete       = off
count.all                 = off
count.used                = off
crossref.limit            = 32
default.key               = "**key*"
dir.file.separator        = "/"
env.separator             = ":"
expand.macros             = on
fmt.et.al                 = ".ea"
fmt.inter.name            = "-"
fmt.name.name             = "."
fmt.name.pre              = "."
fmt.name.title            = ":"
fmt.title.title           = "-"
ignored.word              = "{a}"
ignored.word              = "{a}n"
ignored.word              = "the"
ignored.word              = "le"
ignored.word              = "les"
ignored.word              = "la"
ignored.word              = "{}un"
ignored.word              = "{}une"
ignored.word              = "{}el"
ignored.word              = "{}il"
ignored.word              = "der"
ignored.word              = "die"
ignored.word              = "das"
ignored.word              = "{}ein"
ignored.word              = "{}eine"
key.base                  = lower
key.expand.macros         = on
key.format                = short
key.generation            = off
key.make.alias            = off
key.number.separator      = "*"
```

```
new.entry.type            = "{}Article"
new.entry.type            = "Book"
new.entry.type            = "Booklet"
new.entry.type            = "Conference"
new.entry.type            = "{}InBook"
new.entry.type            = "{}InCollection"
new.entry.type            = "{}InProceedings"
new.entry.type            = "Manual"
new.entry.type            = "MastersThesis"
new.entry.type            = "Misc"
new.entry.type            = "PhDThesis"
new.entry.type            = "Proceedings"
new.entry.type            = "TechReport"
new.entry.type            = "{}Unpublished"
preserve.keys             = off
preserve.key.case         = off
print.align               = 18
print.align.string        = 18
print.align.preamble      = 11
print.align.comment       = 10
print.align.key           = 18
print.braces              = on
print.comma.at.end        = on
print.all.strings         = on
print.deleted.prefix      = "\#\#\#"
print.deleted.entries     = on
print.entry.types         = "pisnmac"
print.equal.right         = on
print.indent              = 2
print.line.length         = 77
print.newline             = 1
print.parentheses         = off
print.terminal.comma      = off
print.use.tab             = on
print.wide.equal          = off
rewrite.case.sensitive    = on
rewrite.limit             = 512
quiet                     = off
select.case.sensitive     = off
select.crossrefs          = off
select.field              = "\$key"
sort                      = off
sort.cased                = off
sort.format               = "\%s(\$key)"\index{s@\%s}
sort.macros               = on
sort.reverse              = off
suppress.initial.newline  = off
symbol.type               = lower
verbose                   = off
```

C.2. Useful Translations

The resource file `tex_def` translates international characters into plain text representations. Especially the German umlaut sequences are translated. For instance the letter Ä which is written as `{\"A}` in a BibTeX file is translated to `Ae`.[1]

[1] Note that the short notation of `german.sty` or `babel` is not understood by BibTeX nor by BibTool.

Additionally some logos are defined.

```
tex.define {\"[1]=#1e}
tex.define {\ss=ss}
tex.define {\AE=AE}
tex.define {\OE=OE}
tex.define {\aa=aa}
tex.define {\AA=AA}
tex.define {\o=o}
tex.define {\O=O}
tex.define {\l=l}
tex.define {\L=L}
tex.define {\TeX=TeX}
tex.define {\LaTeX=LaTeX}
tex.define {\LaTeXe=LaTeX2e}
tex.define {\BibTeX=BibTeX}
tex.define {\AMSTeX=AMSTeX}
```

C.3. Other Resource Files

The distribution contains additional resource files. Some of them are sketched here. Others may be contained in the distribution as well. Look in the appropriate directory.

iso2tex
: define rewrite rules to translate ISO 8859-1 characters into BibTeX compatible sequences.

iso_def
: define macro equivalents for ISO 8859-1 characters into TeX compatible sequences.

sort_fld
: defines a sort order for the common BibTeX entry types.

check_y
: contains a sample for semantic checks. The year field is checked to be a suitable number.

month
: tries to introduce BibTeX strings for month names. Provisions are made to preserve other information contained in the month field.

opt
: copes with `OPT` prefixes as introduced e.g. by bibtex-mode.

braces
: tries to replace double quotes as field delimiters by braces.

Bibliography

[GMS94] Michel Goosens, Frank Mittelbach, and Alexander Samarin. *The LaTeX companion*. Addison-Wesley Publishing Company, 1994.

[Knu89] Donald E. Knuth. *The TeXbook*. Addison-Wesley Publishing Company, 15th edition, 1989.

[Lam94] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley Publishing Company, 2nd edition, 1994.

[Pat88a] Oren Patashnik. BibTeX*ing*, 1988.

[Pat88b] Oren Patashnik. *Designing* BibTeX *Styles*, 1988.

Index