BIBTOOL

A Tool to Manipulate BibTeX Files

Tcl Programmers Manual

Gerd Neugebauer

Abstract

BibTool provides a library of useful C functions to manipulate BibTeX files. This library has been used to implement new primitives for Tcl which utilize these functions. Thus it is easy to write tools to manipulate BibTeX files or graphical interfaces to them.

— This documentation is still in a rudimentary form and needs additional efforts. —

This file is part of BibTool Version 2.41

Gerd Neugebauer
Mainzer Str. 8
56321 Rhens (Germany)

WWW: `http://www.uni-koblenz.de/~gerd`

Net: `gerd@informatik.uni-koblenz.de`

`gerd@imn.th-leipzig.de`

`gerd@intellektik.informatik.th-darmstadt.de`

Contents

1

The Tcl Module

1.1 Introduction

The following description assumes that you are familiar with Tcl [Ous94, Wel95]. Thus it does not repeat any introductory material on this language. You are referred to one of the books or the material to be found on the Web. If you are not familiar enough with Tcl you are strongly encouraged to read an introductory text since there are some subtle points in Tcl which have to be understood. Otherwise the language is not usable.

The BibTool library is made available for the Tcl programming language in the form of a dynamic loadable library. To use this library it is usually sufficient to source the file bibtool.tcl which is created during the installation of the library (see section 2) and which can be found in the same directory where the machine dependent library resides—even though this file is not architecture dependent. Thus the first instruction to use the library in a Tcl program is

```
source bibtool.tcl
```

The file bibtool.tcl contains the information where the libraries can be found. It tries to find and load the appropriate dynamic library for the operating system and version it is running on. If you encounter problems like missing libraries then just reinstall the BibTool library on this architecture into the same directory. This will create a new subdirectory containing the missing library.

1.2 Functional vs. Object-Oriented Reading

During the development of the BibTool library an interesting question occurred. In principle there are two ways of thinking present even in plain Tcl/Tk, namely a functional/procedural and an object oriented point of view. The object oriented point of view can be seen

mainly in Tk where the widgets act like objects. In Tcl most things have only a procedural flavor.

The question arose which paradigm should be supported. The first consideration was to use the procedural paradigm since this would require only a single new Tcl command to be implemented. This means the simplicity argument won.

When lots of subcommands of this single command `bibtool` had already been implemented the issue was reconsidered. It turned out that only one single wrapper function had to be written in C to provide the object oriented point of view. Thus for many subcommands of `bibtool`—namely those dealing with databases or records—there are two alternative notations available.

Consider for example the following situation. the Tcl variables `rec1` and `rec2` contain references to single records (we will see later how we can come to this situation). Then the equality of the records (in the sense of pointers to the same internal data structure) can be checked with

```
bibtool equal $rec1 $rec2
```

This returns `1` upon success and `0` upon failure. In the object oriented paradigm we have to provide a method of a record which decides whether or not it is identical to another record. This can be written as

```
$rec1 equal $rec2
```

Another example is concerned with the access to a record in a database. Suppose we have a reference to a database stored in the Tcl variable `db` and we have already loaded some BibTeX files into this database. Then we can get a reference to the first record—storing it in the Tcl variable `rec`—with the following command:

```
set rec [bibtool first $db]
```

If we consider `$db` as an object then we can use the method `first` to get the first record:

```
set rec [$db first]
```

Thus we have available both paradigms in this library. This effect is achieved by using handles to objects. These handles have the prefix `=BibTcl=`. For each object created a command is defined which is an alias to the appropriate `bibtool` subcommand. As long as the user does not try to define commands with this prefix everything works fine.

1.3 Databases

`bibtool` is designed to handle several databases simultaneously. To distinguish the different databases they get assigned a handle—which is a unique identifier—when they are created. This handle has to be used to refer to a database later on. Thus it is preferable to store it in Tcl variable.

The handle can also be seen as an object in an object oriented context. The database class provides a fixed set of methods for this object. Instead of using an object oriented

notation we have decided to use a more procedural approach. But you can have your own model in mind when you read the code.

One consequence of the use of handles is that BibTool can determine which kind of object is addressed. In case of an invalid object an appropriate error is raised. This can occur for instance if a database handle is used after the database has been deleted, or if an arbitrary string is given as handle which does not correspond to an object in BibTool.

The first step to work with a database is the creation. This can be done with the command `bibtool new`. This command returns a handle for a new database which should be stored in a Tcl variable for later user:

```
set db [bibtool new]
```

This database is empty. If you want to load some files into this database you can give the filenames as arguments to `bibtool new`. Any number of BibTeX files can be given this way:

```
set db [bibtool new a.bib b.bib c.bib]
```

Since BibTool is rather noisy by default it might be preferable to reduce the verbosity before loading a database. This can be done by utilizing BibTool resources as in the following command:

```
bibtool quiet=on verbose=off
```

Details on the use of BibTool resources can be found in section 1.15 on page 29.

But the files can not be loaded at creation time only but also later on. This is done with the command `bibtool read` as shown in the following example:

```
bibtool read $db c.bib d.bib
```

`bibtool read` needs the next argument to be a handle for a database. Any following arguments are taken to be file names which are loaded into the given database. Upon success the databse handle is returned. If one of the files does not exist, can not be read, or another error occurrs then this command issues a Tcl error. You can use `catch` to avoid a termination of Tcl and implement an error recovery scheme:

```
foreach file {c.bib d.bib} {
  if {[catch {bibtool read $db $file} message]} {
    puts stderr "*** $message"
  }
}
```

The next important operation on a database is the writing operation. The command `bibtool write` writes the named database into the given file. Again an error is raised if this writing can not be performed. This is most probably caused by the inability to open the file.

```
set file abc.bib
if {[catch {bibtool write $db $file} message]} {
  puts stderr "*** $message"
}
```

Usually the database is written to the file deleting its previous contents. Sometimes it can be desirable to append the database to an existing file. For this purpose the flag `-append` can be used:

```
if {[catch {bibtool write $db $file -append} message]} {
  puts stderr "*** $message"
}
```

There is one special case of a file name. If the file name is the empty string then the output is written to the standard output stream.

The final operation on a database is its deletion. This can be done with the command `bibtool delete`. This command takes as argument a single BibTool object and performs the delete operation for it. Now we are interested in a database object only. Thus the object is a database handle.

```
bibtool delete $db
```

This command deletes the database `$db`. Any reference to it or its contents might lead to a Tcl error. This deletion does not remove the file on the disk but only frees the handle.

With the means given in this section we can already write a useful tool. If we want to write a Tcl program which normalizes a BibTeX file then this can be done with the following piece of code:

```
bibtool quiet=on verbose=off
bibtool write [bibtool new [lindex $argv 0]] [lindex $argv 1]
```

This Tcl program reads the BibTeX file given as command line argument one and writes it to the file given as the second argument. No error detection is performed. To implement error recovery and the generalization to several input files is left to the reader.

In fact this is a special case of the main loop of BibTool. The same effect could be achieved with the following command line invocation of BibTool:

```
bibtool -- quiet=on -- verbose=off infile -o outfile
```

where `infile` and `outfile` are the names of the input and output files respectively.

Next we want to have access to the records in a database. For this purpose two methods are provided which return a handle for the first or last record in a given database respectively. These handles should be stored in Tcl variables for later use:

```
set rec1 [$db first]
set rec2 [$db last]
```

If the database is empty then no handle is created and the empty string is returned.

Next we can try to find a certain record in the database by speifying its key. This can be accomplished with the `find` method:

```
set rec [$db find $key]
```

DB **delete**

bibtool delete *DB*
: delete a database such that no further operations are possible on them. Cf. `bibtool delete` for records.

DB **find** *KEY*

bibtool find *DB KEY*
: Return a new record handle which corresponds to the first normal record in the database *DB* which has the key *KEY*. If none is found then the empty string is returned.

DB **first**

bibtool first *DB*
: Return a new record handle which corresponds to the first normal record in the database *DB*. If the database is empty then the empty string is returned.

DB **last**

bibtool last *DB*
: Return a new record handle which corresponds to the last normal record in the database *DB*. If the database is empty then the empty string is returned.

Table 1.1: Summary of database operations

If no appropriate record is found then the empty string is returned. If you are finished with the database you should release it. This is done with the `delete` method:

```
$db delete
```

This command tries to free the memory occupied by the database[1]. This operation invalidates the handle the the database and any handles to records therein. This means that any access with such handles leads to a Tcl error.

One method is present which can be used to check a handle:

```
$db valid
```

This command returns `1` if the database is a valid database or record handle. Here is one place where it is preferable to use the procedural writing:

```
bibtool valid $db
```

This works for any string argument `$db` not only those which have been valid BibTool handles previously.

A summary of the commands discussed in this section can be found in Table 1.1.

[1] Currently a part of the memory is not really freed but kept internally to be reused by the next database.

bibtool new
create a new database and return a handle to it.

DB **preamble**

bibtool preamble *DB*
Return the preamble of the database *DB*.

DB **read** *?FILE ... ?*

bibtool read *DB ?FILE ... ?*
read the contents of the BibTeX file *FILE* into the database *DB*.

DB **record** *TYPE*

bibtool record *DB TYPE*
Create a new record in the database *DB* with the type *TYPE*.

DB **sort** *?-reverse?*

bibtool sort *DB ?-reverse?*
sort the database *DB* according to the sort keys. The sort order is ascending or descending depending on the resource sort.reverse. If the argument *-reverse* is given then this order is reversed again.

bibtool sort.format =*FORMAT*

DB **valid**

bibtool valid *DB*
check whether the database handle *DB* is still valid. A database handle is invalidated by the `delete` operation.

DB **write** *FILE ?-append?*

bibtool write *DB FILE ?-append?*
write the contents of the database *DB* into the file *FILE*. If the option *-append* is given then the contents is appended to the file. If the file is the empty string then the output is written to stdout.

Table 1.1: Summary of database operations (continued)

1.4 Records and Fields

Any BibTool database consists of a set of records. Currently there are three types of records which are treated differently. The normal record consists of a set of fields which have names and values. The following example illustrates such a normal record:

```
@Manual{        neugebauer:bibtool,
  author  = {Gerd Neugebauer},
  title   = {{BibTool} -- A Tool to Manipulate {\BibTeX} Files},
  edition = {2.41},
  year    = {1997}
}
```

Currently there are two kinds of special records, namely `@preamble` and `@string` records. These types of records will be discussed later and we will focus for the moment on normal records.

To navigate through the database the methods `forward` and `backward` are provided. These commands take a record handle as argument and modify it such that it references the next or previous record respectively. Upon success the record handle is returned. If there is no next or previous record then the handle is released and the empty string is returned.

```
if {[$rec forward]==""} {puts {At end.}}
if {[$rec backward]==""} {puts {At beginning.}}
```

With these methods we are able to write down a loop which visits all normal records. We show the loop starting from the first element and approaching to the last one. As an application we count the number of records in the database:

```
set count 0
for { set rec [$db first] } \
    { $rec != "" } \
    { set rec [$rec forward] }\
    {
  incr count 1
}
```

Like for database handles we might be interested to get rid of a record handle. For this purpose the method `delete` is also provided for records. The deletion of a record handle invalidates this handle. It does not alter the database in any way. Thus after the following piece of code the database has not been changed.

```
set rec [$db first]
$rec delete
```

If you want to delete a database record you can use the method `bibtool remove`. This takes as argument a single record and removes it from the database. Additionally the handle is invalidated. Thus it can not be used any more.

Consider the problem of looping through all records and deleting some of them. For this purpose we want to assume that we have a boolean function `has_to_be_deleted` which decides whether the record given as argument has to be deleted. Then the loop can be implemented as follows:

```
for {set rec [$db first]} \
    {[bibtool $rec valid]} \
    {set rec $next} {
  set next [$rec dup]
  $next forward
  if {[has_to_be_deleted $rec]} {$rec remove}
}
```

In this example we see several new things. First of all we see the method `bibtool remove` in action. Next we see the method `bibtool valid` which can not only be used to check valid database handles but also valid record handles. It works absolutely analogously and is no surprise.

The new method which might be surprising is the `bibtool dup`. This method creates a new record handle which points to the same record as the handle given as argument. This is necessary since manipulations on the first handle can now be performed safely because they do not effect the new handle—except when the record is deleted.

Let us have a look at `bibtool dup` with another example. Let us assume that the Tcl variable `rec` contains a valid record handle for the database `db`. The following piece of code creates a Tcl variable `new` which contains the same record handle as `rec`:

```
set new $rec
$rec forward
```

The `bibtool forward` operation on `rec` modifies `new` as well. Thus `new` either points to the next record of the initial one or it is invalidated if no such record exists.

In contrast the following piece of code creates a new handle and stores it in the Tcl variable `new`.

```
set new [$rec dup]
$rec forward
```

In this example `new` is not effected by the `bibtool forward` operation on `rec`. Even if there is no next record and `rec` is invalidated then `new` still references the initial record.

We have to come back to `bibtool valid` to complete the picture. Often it is interesting to know in advance whether the next or previous record exists without duplicating a record handle and moving just to avoid to invalidate a record handle. For this purpose an additional argument can be used. If this optional argument to `bibtool valid` after the record handle is `-next` or `-previous` then the existence of the next resp. previous record is checked in addition to the validity of the record handle itself.

Thus the following fragment checks whether there are at least two records in the database `$db`. This is done by positioning a record at the beginning of the database and checking the validity of this record and the next record:

```
set rec [$db first]
if { [bibtool valid $rec] &&
     [bibtool valid $rec -previous] } {
  puts {At least two records.}
}
```

Since we have the possibility to clone a record handle with `bibtool dup` or allocate new ones with `bibtool first` or `bibtool last` we need method to compare two record handles to see whether they point to the same record. This function is called `bibtool equal`. This function takes two arguments and returns `1` if they are valid and point to the same record. If they are valid and point to different records then `0` is returned. If either one is invalid then an error is raised.

Suppose we have a valid record handle stored in the Tcl variable `rec` for which a next record exists. We execute the following piece of code:

```
set new $rec
$rec forward
if {[$rec equal $new]} {print yes}
```

The last line will always produce the answer `yes`. This is due to the fact that the Tcl variables `rec` and `new` contain in fact the same record handle. Whereas the following piece of code will never produce `yes` since here two independent record handles are involved which represent successive records in the database.

```
set new [$rec dup]
$rec forward
if {[$rec equal $new]} {print yes}
```

As an convenient alias the method `bibtool ==` is provided as an alias for the method `bibtool equal`. Thus you can write

```
if {[$rec == $new]} {print yes}
```

This line will have the same effect as the last line in the example above. But beware not to forget the outer brackets. This subtle point is illustrated in the following example:

```
set new [$rec dup]
if {[$rec == $new]} {print yes} else {print no}
if { $rec == $new } {print yes} else {print no}
```

According to the explanations given earlier it is not surprising that the first conditional prints `yes` since both record handles point to the same record in the database. But they are in fact two different handles since `new` is derived from `rec` with the `bibtool dup` method. Thus the textual representation of the handles is different. These textual representations are compared in the second conditional and turn out not to be equal. As a consequence `no` is printed.

This slight distinction might lead to confusion. If you have problems of this kind try to use `equal` instead of `==`. On the other hand the `==` notation is very intuitive and you just have to take care of the context in which it is evaluated by Tcl.

RECORD **backward**

bibtool backward *RECORD*

Makes *RECORD* reference to the predecessor of its current value. Return *RECORD* upon success. If none is present then delete the record handle and return the empty string.

RECORD **delete**

bibtool delete *RECORD*

Deletes the record handle *RECORD*. This does not mean that the record itself is modified. Only the reference to it can not be used any more.

RECORD **dup**

bibtool dup *RECORD*

Create a new record handle pointing to the same record as the record handle *RECORD*.

$RECORD_1$ **equal** $RECORD_2$

bibtool equal $RECORD_1$ $RECORD_2$

compares the two records. Returns 1 if they point to the same physical record.

$RECORD_1$ == $RECORD_2$

bibtool == $RECORD_1$ $RECORD_2$

the same as `equal`.

RECORD **fields**

RECORD **remove**

bibtool remove *RECORD*

Remove the record from its database. The handle is invalidated. The record ceases to exist.

RECORD **valid** *?-next|-previous?*

bibtool valid *RECORD ?-next|-previous?*

check whether the record handle *RECORD* is still valid. If the optional argument *-previous* or *-next* is given then it is also checked whether the previous or next record exists. `1` is returned when all tests are successful. Otherwise `0` is returned.

Table 1.2: Summary of record operations

1.5 Record Fields

As a database consists of a set of records, any record consists of a set of fields. The fields are determined by their name. Each name is unique within a record. In addition to the name any field has a value. BibTeX itself does not impose much restrictions on the allowed fields—except that they have some minor syntactic restrictions. Especially there is only one field which is treated special. This is the `crossref` field which is used for inheritance. Before we come to this point we want to present a method how to check for existing and not existing fields.

The method `bibtool fields` of a record returns the list of all fields defined in a record. For instance we can reconsider the BibTeX record on page 11. Suppose the Tcl variable `rec` contains a handle pointing to this record then

```
fields $rec
```

would return the following list of fields:

```
author title edition year
```

In addition to the normal fields which are stored directly in int record some information can be acessed as if it was stored in a field. Those pseudo-fields will be discussed later.

To check for the existence or non-existence of a field one would could extract the list of fields and use the Tcl command `lsearch`. To avoid the overhead of constructing the intermediate list the method `bibtool missing` is provided.

```
$rec missing publisher
```

This invocation returns `1` if there is no field with the name —publisher— in the record `rec`. Otherwise `0` is returned. As always an error is raised if `rec` does not contain a valid record handle.

Now we come to the access methods for field values. We need a way to retrieve the value and a method to modify the value. The first approach to retrieve a value is using the method `bibtool get`. It takes a record handle and a field name and returns the contents of the field as a string. Let us consider the following example:

```
@String{ BibTool = {{BibTool}} }
@Manual{ neugebauer:bibtcl,
  title =    BibTool # { -- Tcl Programmers Manual},
  crossref = {neugebauer:bibtool},
  remark =   {Distributed with BibTool}
}
```

If we want to get the value of the `remark` field we can use the following method:

```
$rec get remark
```

This command returns the contents of the remark field as string. Thus the result is the Tcl string

```
Distributed with BibTool
```

The example has been chosen to illustrate some other points as well. In this example a BibTeX macro `BibTool` is defined which contains the BibTool logo protected from case changes in BibTeX. This macro is used in the title field with the concatenation operator `#`. Now we want to get the value of the title field:

```
$rec get title
```

This command yields as a result the string representation of the title field. For this purpose all macros are replaced by their values and the resultings strings are concatenated. Thus the result is

```
{BibTool} -- Tcl Programmers Manual
```

Note that the outer double quotes or braces are not contained in the result but inner braces or doulbe quotes are. This is due to the fact that the result is a Tcl string which does not need additional delimiters.

What would have happend if the macro `BibTool` would not have been defined? In this case the result of the macro expansion is the empty string. This empty string would have been concatenated with the rest yielding the result

```
-- Tcl Programmers Manual
```

Sometimes it is undesirable to get the expanded version of the value. For instance if you want to take advantage of the BibTeX macro feature and manipulate things yourself. For this purpose the optional flag `-noexpand` can be used. If this flag is given the result is a Tcl list consisting of strings which contain the components of the value. In this case the delimiters (`{}` or `""`) are part of the elements to distinguish the string constants from macros. Thus the command

```
$rec get title -noexpand
```

leads to a Tcl list with the following two elements:

```
BibTool
{ -- Tcl Programmers Manual}
```

Next we have to consider the case that we are asking for a field which does not exist in the record considered. For instance we might ask for the author field:

```
$rec get author
```

Since the record does not contain an `author` field we get the empty string as the result.

BibTeX has the feature to use the `crossref` field for inheritance. If a field is missing in a record but it has a `crossref` field then the field is sought in the record whose key is the value of the `crossref` field as well. This behaviour can be triggered with the optional flag `-all` of the method `bibtool get`:

```
$rec get author -all
```

In our example the record `neugebauer:bibtool` would be considered in this case. If we assume that this record—as given on page 11—is also present in the database then the result is the string

```
Gerd Neugebauer
```

In addition to the normal fields some pseudo-fields can be queried. These pseudo-fields give access to information not really stored as a field but present for a record, a database, or as a global value in other form. One such a pseudo-field is the citation key. This citation key can be referenced as `$key`. Thus the citation key can be retrieved with the following Tcl construct:

```
set key [$rec get {$key}]
```

Since the `$` is a special character for Tcl it has to be protected to be not evaluated by Tcl. For our example the record as given on page 11 this command will set the Tcl variable `key` to the value

```
neugebauer:bibtcl
```

Table 1.3 contains a list of pseudo fields. There are more pseudo fields which can be found in the BibTool documentation. But the additional pseudo fields can not be considered really important in the context of Tcl.

Now we come to the opposite operation. We want to set the value of a field to a given string. For this purpose the method `bibtool set` is provided. This method takes a field name and a string or a list of components and sets the value accordingly. Since Tcl does not distinguish between a list and a string the second case has to be marked with the flag `-concat`. The effect of this operation is that the field has the given value afterwards. If the field did not exist already then this field is added to the record.

Let us consider an example. The simplest desire is to set the value of a field to new string. Suppose we want to change the value of the remark field then this can be done as follows:

```
$rec set remark {Distributed as part of BibTool}
```

Suppose we want to change the value of the field edition from `2.41` to `2.42`. This field is not present in the record but inherited via a `crossref` field. Nevertheless this is not honored by `bibtool set`. A new field is added to this record containing the new value. The other record is left unchanged.

Suppose we want to add a field `month` to the record. This is one place where BibTeX macros are indispensible. Instead of using the constant `"June"` we should always use the macro `jun` which is defined by most BibTeX styles. Thus the BibTeX style designer can decide to use the full month name, an abbreviation, or even switch to a different language.

If we write

```
$rec set month jun
```

pseudo field name	*meaning*
`$key`	The citation key.
`$sortkey`	The sort key, i.e. the string used for sorting records.
`$source`	The file the record is read from or the empty string.
`$type`	The type of the record.
`$default.key`	The value of the resource `default.key`. This string is used as a key if the key specification failes completely.
`$fmt.et.al`	The value of the resource `fmt.et.al`. This string is used to abbreviate unnamed additional authors.
`$fmt.name.pre`	The value of the resource `fmt.name.pre`. This string is inserted between the first name and the last name.
`$fmt.inter.name`	The value of the resource `fmt.inter.name`. This string is inserted between several last names of one person.
`$fmt.name.name`	The value of the resource `fmt.name.name`. This string is inserted between two names.
`$fmt.name.title`	The value of the resource `fmt.name.title` This string is inserted between name and title.
`$fmt.title.title`	The value of the resource `fmt.title.title`. This string is inserted between different words of the title.
`$fmt.key.number`	The value of the resource `fmt.key.number`. This string is inserted between the generated key and the disambiguating number.

Table 1.3: Some pseudo fields

then the result would be the string `jun` and not the BibTeX macro. Thus we have to use the flag `-concat`.

```
$rec set month jun -concat
```

This works fine since Tcl does not distinguish the one string `jun` from the list containing `jun` as a single element. Nevertheless the correct way would be to use a construction like the following one:

```
$rec set month [list jun] -concat
```

If we want to add a specific day as well then this can be done like in the following example[2]

```
$rec set month [list jun {{~13}}] -concat
```

Beware, BibTool does not check the structure or the contents of field values. Thus you have to be careful to use only valid components in order for BibTeX to work properly.

[2]Note that this does not allow switching the languages any more so easy. But this is not our topic here.

bibtool fields *RECORD*
: This command returns a list of field names in *RECORD*.

RECORD **forward**

bibtool forward *RECORD*
: Makes *RECORD* reference to the successor of its current value. Return *RECORD* upon success. If none is present then delete the record handle and return the empty string.

RECORD **get** *FIELD ?-noexpand|-all?*

bibtool get *RECORD FIELD ?-noexpand? ?-all?*
: This command returns the value of the field *FIELD* in the record *RECORD*. The value returned has all strings expanded. If the option *-noexpand* is given then the unexpanded form as it is contained in the database is returned. If the option *-all* is given then the inheritance via crossref is honored. I.e. each missing field is extracted from the crossrefed record (etc). If the field does not exist then the empty string is returned.

RECORD **missing** *FIELD*

bibtool missing *RECORD FIELD*
: This command returns `1` if the field *FIELD* is missing in the record *RECORD*. Otherwise it returns `0`.

RECORD **remove** *FIELD*

bibtool remove *RECORD FIELD*
: Delete the field *FIELD* from the record *RECORD*. If it does not exist then nothing is changed.

RECORD **set** *FIELD VALUE*

bibtool set *RECORD FIELD VALUE*
: This command changes the value of the field or pseudo field *FIELD* to *VALUE*. The field is added if it has not been present already.

Table 1.4: Summary of field operations

special format	*meaning*
`empty`	Use the default key.
`short`	Use the last names of authors or editors and the first relevant word from the title.
`long`	Use the names of authors or editors with initials and the first relevant word from the title.
`new.short`	Like `short` but only applied to records without a key.
`new.long`	Like `long` but only applied to records without a key.

Table 1.5: Special format specifiers

To complete the operations on record fields we need a method to get rid of a certain field. For this purpose the method `bibtool remove` can be used. We have seen this method already on page 11. If a field is given together with the record then this field is removed from the record. Thus

```
$rec remove remark
```

Deletes the `remark` field from the given record. If the field is not present in the record given then nothing is done. This can be especially confusing if a field is inherited via a `crossref` field. The value can be retrieved with the `bibtool get` method even when the field has been removed:

```
$rec remove author
$rec get author -all
```

The result of the second command is `Gerd Neugebauer` even so the field has been deleted before.

1.6 Key Generation

The generation of new reference keys has been one of the first functionalities present in BibTool. The keys are generated according to a specification described in detail in the BibTool documentation. Thus it is not repeated here.

The format specification can be specified with a resource command (see section 1.15. For convenience the command `bibtool key.format` is provided. It can be used to set the key format. It is important to construct the argument such that the string which arrives in the resource command is constructed properly. This means that you have to be careful which characters might need quoting. The easiest way is to enclose the complete format with braces. Thus nearly everything works as expected.

Usually the function `bibtool key.format` adds the given format specification as a further alternative after the specification already present. Thus it is possible to iteratively construct the specification by giving cases in decreasing order. This exception to this rule

bibtool key *RECORD*
: This command creates a new reference key for *RECORD* according to the specification in effect and stores it in *RECORD*. This string is returned.

bibtool key *DB*
: This command creates a new reference key for each record in *DB* according to the specification in effect. The empty string is returned.

bibtool key.format =*FORMAT*

bibtool ignored.word =*WORD*

Table 1.6: Summary of key generation operations

are the special specifications shown in Table 1.5. Those special specifiers are not added but they entirely replace the old values. Thus the following instruction can be used to clear the old value before adding new alternatives:

```
bibtool key.format {empty}
```

Two practical schemes are provided as convenient abbreviations, namely `short` and `long`. They use the names of authors or editors of a record together with the first relevant word of the title.

```
bibtool key.format {short}
```

The relevant word is determined by skipping over all words which are added to the list of ignored words with `bibtool ignored.word`. This list usually contains articles from different languages. and is initialized at compile time.

Now we are ready to generate a new key. This cam be done either globally by applying the function `bibtool key` to a database. In this case the new key is generated for each record in this database.

```
bibtool key $db
```

The alternative is to apply they key generation algorithm to a single record. In this case only the key of this record is generated. As a side effect the new key of this record is returned.

```
set key [bibtool key $rec]
puts $key
```

To be completed.

1.7 BibTeX Macros

In section 1.5 we have already seen BibTeX macros and their expansion when the value of a field is retrieved. Since it is possible to get the value of a field in unexpanded form we need a way to get our hands on the value of a macro. For this purpose the database method `bibtool string_get` is provided.

```
$db string_get BibTool
```

returns the value of the string `BibTool` as Tcl string. If this macro is not defined then the empty string is returned. The same remarks as for the values of fields hold for BibTeX macros as well. They can be defined in terms of other macros with the concatenation operation (`#`). If you want to get the unexpanded definition the you have to use the flag `-noexpand`. In this case the value returned is a list of components to be concatenated. In this case the string delimters are included to allow you to distinguish strings—which are enclosed in delimiters—from macros.

The reverse operation to accessing a macro value is the definition of one. Analogeously to field values the method `bibtool string_set` can be used. This method takes a macro name and a string and arranges that the macro expands to the string given:

```
$db string_set BibTool {{\sffamily BibTool}}
```

To be completed.

```
$db string_set BibTool [list Bib Tool] -concat
```

If the value of a macro is the empty string we can not distinguish the case that the macro is not defined at all from the case that the macro is defined and has the empty string as its value. To allow this differtiation the method `bibtool string missing` can be used:

```
$db string_missing BibTool
```

This method returns `1` if the macro `BibTool` is not defined and `0` otherwise. Thus we would get `0` in our example.

Since we are now able to distinguish a not existing macro from an empty macro we need a method to get rid of a macro completely. This can be accomplished with the method `bibtool string_remove`:

```
$db string_remove BibTool
```

After this operation the macro `BibTool` is missing. If the given macro has not been defined in the database then nothing will be done. Otherwise the macro definition is removed from the database.

Finally we might be interested to get a complete list of all macros defined for a database. For this purpose the method `bibtool strings` can be used which returns the names of all macros as a Tcl list:

```
$db strings
```

DB **string_get** *MACRO ?-global?*

bibtool string_get *DB MACRO ?-global?*

This command retieves the value of the macro *MACRO* from the database *DB* or the global set of macros. If the flag *-global* is given then only the global maros are considered.

DB **string_missing** *MACRO ?-global?*

bibtool string_missing *DB MACRO ?-global?*

This tests whether the macro *MACRO* is defined in the database *DB* or the global macros. If the flag *-global* is given then only the global macros are considered.

DB **string_remove** *MACRO ?-global?*

bibtool string_remove *DB MACRO ?-global?*

Remove the definition of the macro *MACRO* from the database *DB*. If the flag *-global* is given then the global macros are considered only.

DB **string_set** *MACRO VALUE ?-global?*

bibtool string_set *DB MACRO VALUE ?-global?*

This command assigns the new value *VALUE* to the macro *MACRO* in the database *DB*. If the flag *-global* is given then the change is not made local to the database but in the global set of macros.

Table 1.7: Summary of macro operations

1.8 Embedding (LA)TEX Macros

```
$db preamble
```

To be completed.

1.9 DeTEXing

When you are dealing with (LA)TEX commands it is pretty easy to get rid of macros. This cam be done with a `regsub` command wich matches the macro names and replaces them by the empty string. Suppose the Tcl variable `str` contains a string which should be deTEXed.

```
regsub -all {\\([^a-zA-Z]|[a-zA-Z]+)} $str {} str
```

After the invocation of this command the Tcl variable `str` has all macro names stripped. For instance the string

```
M{\"u}ller GmbH{\&}Co.
```

is translated to

```
Muller GmbH{}Co.
```

Even if the first substitution might be acceptable for non-German speaking people then the second replancement is by far too aggressive. On the other side we could just replace the backslashes. This can be done with a Tcl command like

```
regsub -all {\\} $str {} str
```

results in

```
M"uller GmbH{&}Co.
```

This leaves the `&` but also the crippled `"`. Th eother problem are the arguments. They require matching braces. this can not be expressed with regular expressions. Thus BibTool provides a command to define TEX macros and expand in a string them.

The command `bibtool tex define` can be used to define TEX macros which should be used for expansion lateron. For our previous example we would have needed the following definitions:

```
bibtool tex define {\&=&}
bibtool tex define {\"[1]=#1e}
```

The argument of `bibtool tex define` is one string containing a control sequence on the left side optionally followed by the number of arguments in brackets. It is completed by a equality sign and the replacement text. If no arguments are given then the macro is assumed to have no arguments.

In the example above the first definition makes a macro `\&` without arguments and the second definition makes `\"` a macro with one argment. Like in LATEX the sequence `#`n represents the n^{th} argument.[3]

[3]Note that in German ae is the representation of ä without the umlaut accent.

bibtool tex define *mac?[n]?=repl*
: Define the TeX macro or active character *mac*. If *[n]* is given then *n* must be a single digit which is interpreted as the muber of arguments of the macro. The replacement text is *repl*. In ttthe replacement text `#`*n* represents the n^{th} macro argument.

bibtool tex expand *STRING*
: This command returns the string which results from *STRING* by replacing each defined macro or active character and the optional arguments by it's replacement text.

bibtool tex reset
: This command deletes all macros and active characters previously defined.

Table 1.8: Summary of operations on TeX strings

There is one additional case which has not been described already. If you are a little bit familiar with TeX you might know that you can make a character active and assign a macro to this character. Thus you can omit the leading backslash.

This case is coverd in `bibtool tex define` as well. If the macro name consists of a single character only which is not the backslash then this character is made active then the remainder is used to assign a macro to it.

As a side note I want to note that the reading and macro expansion apparatus of TeX is internally imitated to a certain degree. This included catagory codes. But those catagory codes can (currently) not be modified except by making a character active as described above.

Initially no macros are defined in BibTool. This can also be reached with the following command:

```
bibtool tex reset
```

After this command has been executed no macros are defined an no characters are active.

But now we want to come to the point where we make use of thsoe definitions. This is achieved with the caommand `bibtool tex expand`. All defined macros are replaced by their replacement text. Undefined macros are left unchanged. Thus we could use the following invocation:

```
set str [bibtool tex expand $str]
```

With the definitions given above and the value of the Tcl variable `str` given in the example above we get as a result the following new value in the Tcl variable `str`:

```
M{ue}ller GmbH{&}Co.
```

Now we could savely strip the braces (and possibly remaining TeX macros with the method described above.

1.10 Name Formatting

`bibtool name list`

To be completed.

`bibtool name count`

To be completed.

`bibtool name format $format $name`

To be completed.

1.11 Formatting

In principal Tcl provides everything neccessary to extract the contents of records and create a formatted string which is stored in a field. Since BibTool started as a tool to autmatically create reference keys for BibTeX databases it has a powerful (and fast) mechanism to extract information from a record and format it according to a given specification. For a full description we refer to [Neu97].

The Tcl interface provides not the full functionality as a separate procedure. Instead the basic operations are provided only. The procedure `bibtool format` takes a record and a format specification and usually returnes a string representation of the format where certain constructions are expanded.

To start with the easy case we can state, that any character except the percent sign `%` is passed unchanged from the fromat to the result. The percent sign acts as an escape character which starts an extended command. To produce a single percent sign in the output you have to give two percent signs in the format specifier:

```
$rec format {123%%abc}
```

This instruction returns `123%abc`.

The percent sign starts a format specifier which is similar to a format specifier of the Tcl `format` instruction. But instead of giving arbitrary arguments to be formatted in addition to the format specifier, the format contains the names of fields in parentheses. The values of those fields – or pseudo-fields – are formatted according to the format specifier. The general form is as follows:

`%` *sign pre.post spec* (*field*)

We will explain the meaning of the different parts from right to left. *field* is the name of a field or pseudo-field. The value of this field is formatted according to the remaining parts of the format specifier.

spec is a single letter which determines the main functionality. Details can be found in Table 1.9. For instance the letter `n` denotes name formatting. The other parts of the format specifier are optional and influence the exact operation of the formatting. In any case characters that are not allowed are silently ignored and TeX macros are expanded.

In general *spe* can be preceeded by the qualifier `#`. But this is not meaningful in the context of Tcl since it always returns the empty string.

Let us finally reconsider our example record from page 11. For this record `$rec` we get the following results:

```
$rec format "%n(author)"       ↦  Neugebauer
$rec format "%+.3n(author)"    ↦  NEU
$rec format "%-.2n(author)"    ↦  ne
$rec format "%d(edition)"      ↦  2
$rec format "%d(edition)"      ↦  2
$rec format "%+3.2d(edition)"  ↦  041
$rec format "%2d(year)"        ↦  97
$rec format "%-w(title)"       ↦  bibtool
```

1.12 Analyzing a LaTeX aux File

BibTeX uses inormation from the auxiliary files created during a LaTeX run. If you want to write a program to extract the records used in a document or if you want to write a replacement for BibTeX you need a method to get the information from the aux file. Since the information might be distributed in several aux files if the LaTeX document uses `\include` to combine several subdocuments.

Thus it is rather handsome to have a method which collects th required information. This is done by the functions `bibtool aux` and `bibtool aux -db`.

The function `bibtool aux` returns a list containing all citations mentioned in the aux file. One key is traeted special. If the key `*` is used then all records in the database should be used. In LaTeX this is accomplished by the macro invocation

```
\nocite{*}
```

Thus if a `*` is recognized then the list contains only one element namely this star. Otherwise you get the complete list with

```
bibtool aux $auxfile
```

If the document contains references to `neugebauer:bibtool` and `neugebauer:bibtcl` then the following Tcl list is returned:

```
neugebauer:bibtool neugebauer:bibtcl
```

The other information which can be extracted from an aux file is the list of databases to be used. This is done with the function `bibtool aux databases`. It takes the name of an aux file and returns the list of databases mentioned there:

```
bibtool aux $auxfile -db
```

More than one databases can be used in one document by separating them by a comma (`,`). In a LaTeX document this might look at follows:

```
\bibliography{db1,db2,db3}
```

In this case the function `bibtool aux -db` returnes the following list

```
db1 db2 db3
```

If the aux file given to either function does not exist or can not be read then a Tcl error is raised. Thus you should always use `catch` and implement your own error recovery routine.

1.13 Recognized Entry Types

In BibTeX any record has a certain type – the entry type. BibTool maintains a list of known entry types and complains if a record is read which has an unknown entry type. Such records are not stored in the database. Usually the entry types of the standard BibTeX styles are already known to BibTool. But if you are using some other kind of entry type then this has to be declared in BibTool. This can be done with the resource command `new.entry.type`.

```
bibtool new.entry.type={Law}
```

Note that no spaces are allowed around the equality sign and that the braces are neccesary to preserve the case of the letters. This is useful since BibTeX and BibTool normally ignore the case of letters. But the string given as entry type is not only used to determine whether or not a record can be stored in the database but also as the printing representation of the entry type. Thus in our example it would be possible to have records of the following kind

```
@law{ ... }
@Law{ ... }
@LAW{ ... }
```

All of those would be accepted and be assigned the same entry type. All of them would be printed like the second example.

As a sidemark you should note that `new.entry.type` can also be used to redefine the printing representation of existing entry types. Whenever an existing entry type is encountered as argument then just this modification is performed.

For a Tcl program it might be necessary to know which entry types are defined in Bib Tool. Thus it can present a menu to the user to select the appropriate type. For this purpose the instruction `bibtool types` is provided. This can be seen in the following fragment of a program where `$menu` is assumed to contain the path to a previously created `menu`.

```
foreach type [bibtool types] {
  $menu add checkbutton       \
      -variable entry_type    \
      -value $type            \
      -label $type
}
```

If you run this example you will see that the menu contains normal entry types only. Currently the following entry types are considered special:

COMMENT	ALIAS
PREAMBLE	INCLUDE
STRING	MODIFY

The three sepcial entry types to the left are well known. The entry types to the right are likely to be introduced in BibTeX 1.0.

If you want to get a complete list of all entry types you can give the optional argument `-all` to the `bibtool types` command. Then the special entry types are returned in addition to the normal entry types.

1.14 The Version Number

Finally, BibTcl provides a means to get hold of the version number. This can be useful to give this information to the user or to see whether a new enough version is used when older versions might miss some features. I would like to provide means to test for features directly. But until I have descided how this will be implemented the version is the only indicator which can be used.

The version number is returned by the function `bibtool version`. The return value consists of a number *major.minor* where *mayor* is the major version number and *minor* is the minor version number. Additionally some additions may be returned. Thus

```
bibtool version
```

may return `2.41` but a value of `2.41-a` is also a legal value—even so it is not very likely that you get your hands on such a release.

1.15 Using BibTool Resources

Nearly all resource commands of BibTool are accessible through the `bibtool` command. If nothing else fits then the argument of the `bibtool` command is passed on to the resource evaluator. Since this is done for each single argument this means that special characters have to be protected in a way to ensure that they arrive safely there.

For instance the resource evaluator treats the equality sign as optional. Separating spaces are enough. Thus the following two commands have the same effect:

```
bibtool quiet=on
bibtool {quiet on}
```

Note however that the second form requires the braces to protect the embedded space.

BibTool resources can also be queried. Since not each resource command corresponds to a single variable not all resource commands can be used to query values. At least each resource command which corresponds to a string, a boolean, or a numeric value can be queried.

Consider the resource instruction from above `quiet=on`. This instructions sets the boolean resource `quiet`. This variable can be queried with

```
bibtool quiet
```

This Tcl command returns the value of the resource variable. Booleans are returned as `0` or `1` to conform to the Tcl conventions for booleans.

To be completed.

format	*meaning*
%*sign pre.post* d(*field*)	The $post^{th}$ number is extracted from the field – *post* defaults to 1. *pre* digits starting from the right are returned. If *sign* is + then missing digit are replaced by 0. If *sign* is - and no number is found then 0 is returned instead of the empty string.
%*sign pre.post* D(*field*)	Acts like d but does not truncate longer numbers to *pre* digits.
%*sign pre.post* n(*field*)	The field is treated as a list of names. Last names are extracted. At most *pre* names are used and remaining names are indicated. At most *post* letters from each name are shown. If *sign* is + then all letters are translated to upper case. If *sign* is - then all letters are translated to lower case.
%*sign pre.post* N(*field*)	Acts like n but appends the initials as well.
%*sign pre.post* p(*field*)	Use the name format *pre* to format at most *post* names accordingly. Translate the case according to the *sign*.
%*sign pre* s(*field*)	Use at most *pre* characters. Translate the case according to the *sign*.
%*sign pre.post* t(*field*)	Use a list of words. The words in the ignored.word list are not considered. At most *pre* words are shown. At most *post* letters from each word are used. Translate the case according to the *sign*. fmt.titlt.title is inserted between words.
%*sign pre.post* T(*field*)	Acts like t but does not ignore words.
%*sign pre.post* w(*field*)	Acts like t but inserts nothing between words.
%*sign pre.post* W(*field*)	Acts like T but inserts nothing between words.

Table 1.9: Format specifiers

bibtool aux *FILE*

Reads the given LaTeX aux file and collects a list of all citations. If one citation is * then this * is returned. Otherwise a list of all citations found is returned. If the LaTeX document has used \include then the respective aux files are also considered.

bibtool aux *FILE -db*

Reads the given LaTeX aux file and collects a list of all databases requested. If the LaTeX document has used \include then the respective aux files are also considered.

Table 1.10: Summary of operations on aux files

bibtool types *?-all?*
: Return the list of defined entry types. Normally all normal entry types are returned. If the option `-all` is given then the special entry types are delivered as well. The entry types are returned in some internal order. They are not sorted.

bibtool version
: Return the version number of BibTool.

Table 1.11: Summary of misc operations

bibtool *RSC_VARIABLE*
: Returns the contents of the resource variable *RSC_VARIABLE* if this variable is available. Strings, booleans, and numerics are available.

bibtool *RSC ...*
: Passes the resource command *RSC* to the resource evaluator of BibTool. Each single argument is treated as a separate resource command.

Table 1.12: Summary of resource operations

2

Installation

This section is the last one since it is required only once. It describes the installation of the dynamic loadable library.

2.1 UNIX

- I suppose that you have unpacked BibTool. You should be in the main directory containing the directory BibTcl.
- Configure and compile BibTool—if not already done. See the files README and INSTALL there. The object files must be present in this directory. (In a next release a single library will be assembled and the C API might be documented.)
- Find the file tclConfig.sh. This file is created during the configuration of Tcl and installed with it. One place to search for it is in the directory /usr/local/lib or wherever Tcl has installed its libraries. Copy this file into the directory BibTcl.
- Go to the directory BibTcl and run

  ```
  ./configure
  ```

  This will have a look at your system and find out which properties it has. From this information a Makefile is created.
- Run

  ```
  make LIBDIR=/usr/local/lib/BibTool
  ```

  where the path on the right side of the = should point to the installed BibTool.

  This step should produce the shared library and a small Tcl loader bibtool.tcl which is independent of the architecture. In fact it figures out the architecture and uses the appropriate installed version.

- Run

  ```
  make install LIBDIR=/usr/local/lib/BibTool
  ```

  This creates a subdirectory of LIBDIR according to the OS used. This directory contains the shared library as well as the loader script. The loader script can be freely copied to any other place. The shared library must stay in this directory since the location is compiled into the loader script.

Index

Bibliography

[Neu97] Gerd Neugebauer. *BibTool – A Tool to Manipulate* BibTeX *Files*, 2.41 edition, 1997.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley Publishing Company, Reading, Mass., 1994.

[Wel95] Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall PTR, Upper Saddle River, New Jersey, 1995.