btOOL: An object-oriented library for processing BibTeX-style text databases

Gregory P. Ward (`greg@bic.mni.mcgill.ca`)
School of Computer Science
McGill University
3480 University Street
Montreal, Quebec, Canada H3A 2A7

March 17, 1998

Contents

List of Figures

Abstract

btOOL is an object-oriented programming library for accessing BIBTEX databases in a general way, unhindered by any assumptions about the nature of the data or range of operations one might wish to perform on it. btOOL addresses the lack of support for writing general-purpose utilities to process this flexible and extensible database format; most existing tools (including BIBTEX itself) either make unwarranted assumptions about the data and its intended purpose, depend for extensibility on a special-purpose programming language that unnecessarily limit the programmer's freedom, or are not extensible at all. Because btOOL provides an object-oriented interface to a widely-used high-level programming language (Perl), these problems disappear. In addition, btOOL provides a framework for writing classes specific to a programmer-defined set of database conventions, which opens the door to an object-oriented analogue to BIBTEX's style files.

1 Introduction and Motivation

TEX and LATEX are widely used for typesetting and document-processing, particularly of academic and scholarly publications. One of the most essential and common tools used in conjunction with LATEX is BIBTEX, a system for preparing bibliographies for typesetting by LATEX. Unfortunately, while BIBTEX's database format is very flexible and general, support for arbitrary database operations is hampered by its use of an arcane, special-purpose programming language. At least two auxiliary tools, BibTool and **bibclean**, have been developed to address this weakness, but both have in the past suffered from lack of a full-featured programming interface, leading to limited extensibility and/or customizability.

To addresses these weaknesses, a full-featured programming library, btOOL (BIBTEX Object-Oriented Library), has been developed to allow programmers full and unhindered access to BIBTEX databases. btOOL consists of a tight, efficient foundation written in C, and a convenient, high-level, object-oriented interface written in Perl. This gives programmers access to BIBTEX data from both a low- and high-level language for simple database accesses and transformations. Using general-purpose languages means that a bit more code may be required for certain tasks than if a special-purpose language were available, but that no arbitrary restrictions are placed on what data may be processed or what may be done to it. Additionally, btOOL takes full advantage of Perl's object-oriented features to provide a framework for developing new bibliography formatting modules completely independent of BIBTEX's dedicated formatting language—or, in fact, modules specific to any kind of text-oriented database that can be adapted to BIBTEX's syntax.

1.1 About BIBTEX

BIBTEX generates reference lists for LATEX documents by selecting a subset of a (possibly large) database of bibliographic entries. This subset is document-specific, determined by the citations that appear in the text of the document. The source database is stored in a structured but highly flexible format. The generation of marked-up text from this database format is driven by a BIBTEX *style file*, which is just a program written in a custom, special-purpose language geared towards the production of formatted bibliographies. The formatted bibliography is then included and typeset with the text of the LATEX document.

A sample entry from a BIBTEX database is shown in Figure 1. A database entry consists of an *entry type*, a *key*, and a list of *fields* and *values*. In this example, the entry type is `article`, and the key (which must be unique across all entries in the same database) is `patashnik:tugboat:1994`. In general, a field value is the concatenation of one more simple values, each of which may be a string, a number, or an *abbreviation*. In this example, every field has a single simple value, so concatenation is not used. The `author`, `title`, and `pages` fields illustrate the use of literal strings; `year`, `volume`, and `number` demonstrate numeric field values; and `journal` and `month` make use of abbreviations (also known as *macros* in the btOOL context). Macros have a multitude of purposes: they save repeated typing of long strings that occur often in a database; they ensure consistency and help avoid typographical errors; and they can be used easily to substitute context-dependent variations on a string. For example, BIBTEX's standard style files provide pre-defined macros for the twelve month names; depending on the style, these may be either full (`sep` = "September") or abbreviated ("Sept.").

```
@article{patashnik:tugboat:1994,
  author  = "Oren Patashnik",
  title   = "{{\BibTeX}} 1.0",
  journal = TUGboat,
  year    = 1994,
  volume  = 15,
  number  = 3,
  pages   = "269--273",
  month   = sep
}
```

Figure 1: An example BIBTEX entry. This entry assumes the definition of two macros, `sep` and `TUGboat`.

The advantages of BIBTEX's data language are many, although not all are immediately obvious from this example. For starters, the syntax is quite flexible and free-form; fields may come in any order, arbitrary whitespace may be used, and strings may be of unlimited length[1]. These syntactic features improve readability and make simple maintenance a matter merely of editing the database file. The data language is also extensible; applications may accept (almost) any entry types that they like, and may recognize and/or require any desired field names for a particular entry type. Automated database maintenance, however, is hampered by the need to write a full-blown parser for the data language—a task comparable to writing the front end to a compiler for a simple programming language such as Pascal.

The flexible and extensible syntax is important, but the fundamental feature of BIBTEX is the separation of content from form through an explicitly structured representation. For instance, the above reference would conventionally be rendered as (and in fact the ultimate output of BIBTEX's standard style files after processing that entry would be something like):

> Oren Patashnik. BIBTEX 1.0. *TUGboat*, 15(3):269–273, September 1994.

Here, the structure is implied by the tightly interwoven form and content: only through familiarity with the conventional form do we understand the nature of the publication and how to access it. Automatic interpretation of this implicitly structured form would be quite difficult when one considers the multitude of variations: what is one to make of journals with numbers but no months (or vice-versa), or multi-sentence titles? Expanding one's view to handle the multitude of publications commonly referenced—books, book chapters,

[1] At least in theory, strings may be of unlimited length. BIBTEX 0.99 suffers from hard-coded constraints, but btOOL has no such restrictions.

pamphlets, master's and doctoral theses, and so forth—makes the job well-nigh impossible. In Figure 1, the journal and issue being referenced are immediately obvious. The explicit, unambiguous syntax makes the creation of tools for automatic bibliography processing, such as BIBTEX itself, feasible. The separation of content from form—content explicitly labelled in the structured database, form programmed into the style file—means that the same data can be processed in different ways to achieve different forms. Finally, the extensibility of the data language makes it applicable to many applications other than bibliographic databases.

However, BIBTEX does not—and was never intended to [10]—cover the whole range of possible bibliography processing tasks. Rather, it is specifically geared towards formatting sets of bibliographic entries into a form suitable for inclusion in a LATEX document. It is unsuitable for programming "database-to-database" transformations where one wishes to preserve the overall structure of a database while manipulating the data in it in some way. It is also difficult to generate something other than the reference list for a LATEX document; for instance, one often wishes to generate a document consisting of nothing but bibliography entries, or to format all entries in a bibliographic database. These are both possible, but awkward, with BIBTEX in its present form. Finally, generating output in a markup language other than LATEX (say HTML) would be quite difficult (but not impossible—see [7]), given the assumptions inherent in BIBTEX's handling of the data. These are not problems that arise in everyday use for typical LATEX documents, but they often arise when dealing with large databases or generating certain specialized types of documents.

1.2 Where BIBTEX falls down

The project that led to btOOL was just such an example of dealing with large databases and generating specialized documents—in particular, documents consisting entirely of bibliographic data. As happens periodically to many large research groups, the Brain Imaging Centre at the Montreal Neurological Institute was faced in 1996 with the daunting task of preparing a group grant proposal involving nearly fifty principal investigators and roughly fifteen separate projects. To assemble bibliographies for all investigators, we started by collecting all of their individual *curricula vitæ*, which were then manually entered into BIBTEX databases. The ultimate goal was to merge all the individual CVs into a large master database (roughly 2,000 entries) with all names/titles (authors, journals, and conferences) standardized and duplicate entries removed. From this master database would then be extracted new, canonical per-investigator databases (which invariably contained more publications than the investigator's original CV). A LATEX document containing all of an investigator's publications, sorted chronologically, was then generated. This document was then divided into three parts, one for each of the classes of publications deemed relevant by the granting agency (journal articles, book chapters, and conference abstracts). The inability of any existing software to handle elegantly all of these tasks led directly to btOOL, a pair of libraries that let programmers easily write their own database processing tools.

Transforming the manually-entered BIBTEX databases into a form suitable for merging proved to be the major task. For starters, the data entry had been done by several users, each possessed of a slightly different misunderstanding of BIBTEX's syntax and database structure. The first step was to correct the resulting errors—first, using a stand-alone parser as a syntax-checker, and then automatically correcting systematic structural errors. Author names were standardized by automatically clustering all of the various misspellings of each name, manually correcting this clustering, and assigning a canonical form for each name. (This meant breaking the unwritten rule that bibliographies should always record the author's name as it appears in the publication in question. Following this rule, however, would have made extracting per-investigator databases after the merge more difficult.) Standardizing journal titles and conference names was simpler, but only because there were so many fewer of them—it was feasible to do most of the work manually, rather than having to attempt automatic fuzzy matching of the various representations of the same name. With these transformations made, merging the databases was feasible: a "signature" string was generated for each entry by combining the authors, title, and publication information (e.g., for a journal article, journal name, year, volume, number, and pages) into a single string and removing extraneous words. Entries with identical signature strings were deemed duplicates; after several passes, we brought the databases into a form where each entry had sufficient information (correct volume, number, and page range, for journal articles) to allow its automatic detection and removal as a duplicate if appropriate.

Once the corrections to and merging of the individual databases were complete, there remained the matter of extracting three new per-investigator databases (one for each class of publications) from the master list, and sorting them chronologically. Thanks to the standardization of author names, selecting all of a particular investigator's publications was trivial. And thanks to BIBTEX's extensible syntax, we were able to insert an extra field earlier in the processing to allow easy differentiation of publication types (BIBTEX's standard types divide publications up in a slightly different way which was not easily applicable to our needs). The entries were sorted by year, authors, and title (as opposed to the sorting available with BIBTEX's standard styles, which follows authors/title/year order). Finally, a single LATEX document containing three bibliographies was created using some LATEX trickery (the bibunits package [5]), the bibliographies were formatted using one of BIBTEX 0.99's standard style files, and typeset with LATEX.

From the beginning, it was obvious that BIBTEX itself was inappropriate for most of the transformations described above. For starters, the "style file" programming language used to write new formatting programs is a dedicated, special-purpose programming language. Learning it would have taken precious days or weeks away from the time needed to program all of the needed database operations. Even after that time was spent, the programming would have been much more difficult: being a stack postfix language, BIBTEX's formatting language is inherently confusing and difficult for programmers used to conventional infix notation. The special-purpose nature of the language limits its use and the likelihood of third parties adding value to the language—meaning it lacks such vital elements of development support as ample documentation, a debugger, or access to the operating system or file-system. Furthermore, the effort invested in learning a single-task language has no future payoff; a language solely dedicated to writing BIBTEX formatting programs has little applicability elsewhere. Much of this applies, of course, to any dedicated language, and should not be construed as an attack specifically against BIBTEX. In fact, when BIBTEX was created in the mid-1980s, portable, widespread, well-designed, general-purpose high-level procedural programming languages were not yet a reality—so designers of extensible software had little choice but to create a custom extension language.

One half-solution would have been to harness BIBTEX's dedicated language by using it to write small, special-purpose programs wrapped by higher-level programs written in a general-purpose language. This would have side-stepped the lack of external interface support, but the high cost of learning and using the dedicated formatting language dissuaded us from trying this approach.

Thus, we turned elsewhere for support. Two existing packages—Gerd Neugebauer's BibTool [8] and Nelson Beebe's **`bibclean`** family of tools [1]—would appear applicable. They are both powerful, impressively general, and applicable to a wide range of bibliography-processing tasks. Unfortunately, they both suffer from similar assumptions and the resulting restrictions meant (or, in the case of **`bibclean`**, if we had found the software sooner, would have meant) that they were just slightly inadequate to the tasks required for the 1996 grant proposal. It was this inadequacy that led to the creation of btOOL, a programmer's library rather than a user tool: by giving complete access to BIBTEX data in a high-level, general-purpose language, the programmer is limited only by his imagination and ability.

1.3 Existing tools and where they fall down

We discovered BibTool via a search of CTAN (the Comprehensive TEX Archive Network [4]) early in the preparation of bibliographic data for the grant application (March, six months before the September deadline). In a nutshell, BibTool is a program that copies an input BIBTEX database to a new form (also in BIBTEX syntax), with a large number of possible checks and transformations performed along the way. These include generating new entry keys, sorting entries, checking that a given field matches a given regular expression, pattern-based field rewriting, and "pretty-printing" entries to enforce a uniform look. All of these transformations are extensively customizable using a "resource file"—in effect, yet another "toy" special-purpose programming language. The generation of entry keys and sort keys in particular is extremely customizable; in fact, there is a whole formatting mini-mini-language (distantly related to C's `printf()`, but vastly more sophisticated) embedded in the resource file mini-language expressly for generating such key strings.

In short, BibTool is extremely flexible, and applicable to a wide range of common transformations to BIBTEX databases. Like BIBTEX itself, though, BibTool was at the time hobbled by the use of a dedicated mini-language.[2] However simple its syntax (and, for the most part, it is simple—until one wants to generate custom keys) the "one more language to learn" syndrome constitutes a barrier to user acceptance—especially non-programming users. (By contrast, btOOL is by its very nature useless to non-programmers. However, the use of a well-known and widespread programming language such as Perl means that is immediately accessible to a large number of programmers.) More fundamental, though, is that providing access to the data through a "closed box" means users were limited to the operations conceived and implemented by BibTool's author. This is fine for most users, as BibTool's built-in operations are both extensive and highly customizable. When we evaluated it, though, they were not extensible—the user could not step outside the boundaries of the problem as imposed by BibTool. For instance, BibTool could add, delete, or rewrite certain fields—but it could not do so conditionally upon the value of other fields. The generation of entry keys was highly customizable, including the ability to append an optional "disambiguating letter" to distinguish otherwise-identical keys: `smith:1990` and `smith:1990b`. However, defensively appending a letter `a` to all keys to allow for future expansion was simply impossible. (This, incidentally, was the proverbial straw that broke the camel's back—the first version of what eventually became btOOL was finished within a day of running up against this particular wall.)

Unfortunately, we discovered Nelson Beebe's **`bibclean`** family of tools [1] too late for use on this particular project. An after-the-fact evaluation of this set of programs indicates that, in their original form, they would not have been sufficient for the job. Being more extensible than BibTool, though, they would have provided a more convenient jumping-off point on the road that led to btOOL. **`bibclean`** is just one of a handful of utilities all based on Beebe's `lex`/`yacc`-based BIBTEX parser, including **`bibcheck`**, **`bibextract`**, and **`bibsort`**. For convenience, I will refer to the whole set as the **`bibclean`** collection. This collection follows the "toolkit" convention set by Unix: a set of independent programs, each specializing in one particular task, that together form more than the sum of their parts. With BibTool, the set of allowed operations can only be extended by modifying BibTool itself. Using the **`bibclean`** model, one simply has to follow the example of an existing program and write a new tool to extend the toolkit. This is made even easier by **`biblex`**, which does the hard work of lexically scanning a BIBTEX file, and prints the results in an easy-to-parse form. The relatively simple parsing and subsequent processing can then be done in a high-level text-oriented language

[2] It should be noted that BibTool has evolved considerably since mid-1996, when these limitations led us to discard it as a possible basis for all the bibliography transformations needed in preparing the grant application. Most importantly, it is moving away from the monolithic "one-size-fits-all" approach towards providing a general interface for a high-level, general-purpose language (in this case, Tcl). The converging aims of BibTool and btOOL serve to underscore the validity of the approach.

such as awk or Perl, by reading the output of **biblex** and writing a new, transformed BIBTeX database. This model foreshadows btOOL's mode of operation: with **biblex**, a lexical scanner written in C communicates via a pipe with the parser and some task-specific code; with btOOL, the scanner and parser are encapsulated in a C library that is dynamically loaded by a Perl program, which carries out arbitrary database operations via some C/Perl "glue" code.

Perhaps more important than the particular features provided by **bibclean** are the recommendations in the article describing it [1]. In particular, Beebe calls for a rigorous definition of BIBTeX's data language, and recommends separate phases for lexical and syntactic analysis, as is common in modern compilers. This in fact is exactly the approach taken in btOOL, as described in section 3; in fact, the lexical and syntactic grammars used (section B.2) draw on Beebe's recommendations.

1.4 First step: a library for simple database processing

Ultimately, the needs of the Brain Imaging Centre's grant application were filled by using BibTool's pretty-printer to transform BIBTeX files into an easy-to-parse format, and wrapping this in a Perl library that provided object-oriented access to the entries. Using Beebe's **biblex** program might have been a bit more appropriate, but the end effect would have been the same: harnessing an existing tool to do the hard job of lexical scanning, and performing arbitrary transformations of the data by allowing access to it from a high-level, general-purpose language. This is the library that ultimately evolved into btOOL's Perl library (section 4); the role of "parsing aid" has been taken over by a custom C library (section 3) that includes not only a scanner and parser for BIBTeX files, but several common string operations needed for a faithful emulation of BIBTeX. These two libraries together comprise btOOL; all the features of the C library are available to Perl programmers, and the utility of this approach is demonstrated by the example btOOL programs described in section 5.

The key feature of this library is that it provides complete access to the data, making no assumptions about its structure (allowed entry types or required fields). Perl's general-purpose nature makes it possible to write programs with unrestricted flow-of-control and complete access to the operating system, the filesystem, and the user; its high-level nature and text-processing features make most such programs easy. By contrast, writing unrestricted programs is impossible with BIBTeX's style language, and the programs that are possible are quite difficult due to the nature of the language. With BibTool, the tasks allowed by the mini-language are quite easy to specify and customize, but the range of available operations is even more restricted. And with the **bibclean**/**biblex** toolkit, writing arbitrary programs is an option, but the toolkit provides no help beyond lexically scanning the data.

1.5 Beyond simple database transformations

The initial motivation for btOOL was to provide an environment in which programmers could easily write simple database transformations (add/delete certain fields, generate new entry keys, and so forth) using a familiar, high-level language. However, the use of a modern, modular, object-oriented language—Perl, in this case—provides an opportunity to apply object-oriented programming principles to the problem of writing customizable and extensible bibliography-formatting programs. (In fact, one could use the btOOL framework for a wide range of text-oriented databases; currently, though, only the possibilities for emulating and improving upon BIBTeX have been considered.)

Traditionally, BIBTeX style files have been customized or extended by the copy-and-modify model, potentially leading to a confusing divergence from a common source base. There is some customizability inherent in the "standard style files" of BIBTeX 0.99; these four programs were actually derived from a single source file through application of the C pre-processor. This base source file has six boolean options controlling its behaviour, meaning that 64 "standard style files" are, in principle, possible. Generating any of the 60 other available style programs is easy, but only if the user is aware of the possibility, has access to a C pre-processor and knows how to use it.

A far more ambitious and easy-to-use, but essentially similar, approach is embodied by the custom-bib package (also known as **makebst**) [6]. Rather than relying on the C pre-processor, this is a TeX program, guaranteeing that all LaTeX users will be able to run it. **makebst** queries the user on a wide range of options to customize bibliography formatting, and then generates a new BibTeX style file to implement the user-specified format. **makebst** allows customization of typesetting details, punctuation and layout, sorting, internationalization, and several other parameters, leading to an enormous number of theoretically possible bibliography formatting programs.

With the advent of high-level object-oriented languages such as Perl, though, an alternative to either the "copy-and-modify" or "custom code generation" models suggests itself. In particular, one can implement formatting classes (the analogue of BibTeX "style files") which, using the language's facilities to access the rest of the world (including the user's formatting wishes), are customizable at run-time. More fundamentally, though, one can take advantage of inheritance to extend the database: new databases can be described in terms of old ones, and new formatting classes can add functionality on top of old ones. It turns out that writing extensible bibliography formatters via inheritance is a natural fit, and btOOL provides a framework for doing just that.

2 Overview of btOOL

The guiding principle behind btOOL is that BibTeX's data language is sufficiently general for a wide range of textual databases whose uniquely keyed records (entries) fall into a small number of "entry types," where the type of an entry determines the structure to which it must conform. The *structure* dictates which fields are required or expected (or even allowed) to be present in an entry. For instance, in the bibliographic database structure defined by BibTeX 0.99, some of the allowed entry types are `article`, `inproceedings`, and `book`, representing respectively journal articles, conference articles, and whole books. Almost every entry type recognizes such fields as `author`, `title`, and `year`, and various other fields are recognized as appropriate for the type of publication—e.g., `article` entries include a `journal` field. The key point is that every entry has a *type*, a unique *key*, a list of *fields*, and a *value* for each field. Thus, any abstraction must provide access to these four elements; any less precludes the ability to fully represent BibTeX data, and any more is either icing on the cake (such as macros for string abbreviation) or an unnecessary restriction (such as accepting only the entry types defined by BibTeX itself).

From this goal of supporting a "general-purpose data language," it follows that one ought to have a general interface to a general-purpose programming language: if the data language imposes few limits on the data, then the interface should impose few or no limits on what one may do with the data. Special-purpose languages may be interesting to design and implement, and allow one to write programs that do a great deal with very little code; but they have consistently shown themselves to be too restrictive in the real world: creative programmers will always want to step outside the bounds of the language with arbitrary string-processing, system access, or order of execution—any or all of which may be restricted. The tight bounds of special-purpose languages consistently frustrate this creative impulse. Thus, in the long run it pays off for an application to provide an interface to a well-known and widely-used general-purpose programming language, rather than providing a limited-use "toy" language. (It was this special-purpose language problem with BibTeX and earlier versions of BibTool that led to the development of btOOL in the first place.)

In choosing a target language, one must balance popularity, efficiency, elegance, portability, power, and ease-of-use. For popularity and efficiency, it is hard to beat C—and for that reason, the low-level code underlying btOOL's object-oriented interface is written in C. However, writing serious string-processing code in C is tedious and prone to hard-to-find bugs; furthermore, C provides little in the way of support for modern, modular programming, and—in the words of Larry Wall—it is not so much portable as it is ubiquitous. For those reasons, providing access to a popular, portable, and powerful high-level language is important.

Perl provides excellent string-processing capabilities, a thorough interface to both the file system and the operating system, and a simple, elegant mechanism for object-oriented programming. In addition, it is very popular; portable to almost every known Unix variant as well as VMS, MacOS, and Windows 95 and NT; and thoroughly documented both on-line and in print [12, 14].

The high-level architecture of btOOL (illustrated in Figure 2) thus immediately falls into two portions: the underlying C library and the Perl class hierarchy. (The "glue code" connecting the C and Perl libraries falls in between; since it is essential for use of the Perl library, but irrelevant for C programmers, it is grouped with the Perl code, even though it is written in a specially-formatted, pre-processed form of C.) The Perl library further divides in two: the library includes a core class hierarchy that is not specific to any particular database structure, and a sub-hierarchy that provides an example implementation of one particular database structure—that of bibliographies as defined by the standard styles of BibTeX 0.99. Because of their specificity to a particular structure, these classes fall outside the core library, although they inherit from classes within it.

The lexical analyzer and parser were written using PCCTS (the Purdue Compiler Construction Tool

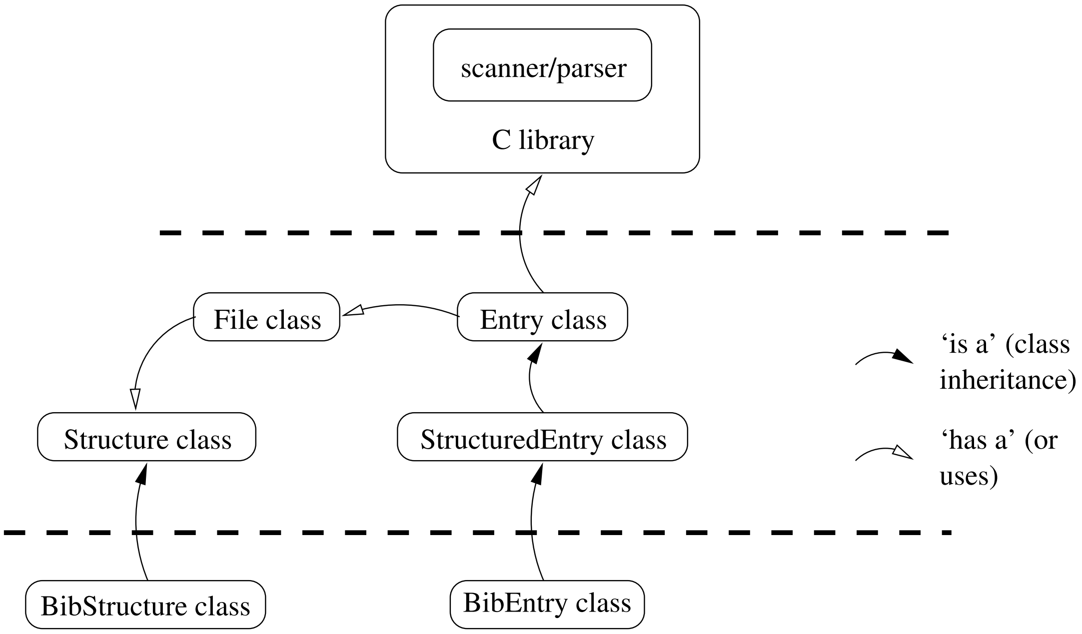


Figure 2: Simplified overview of btOOL: C library, core Perl class hierarchy, and classes responsible for one particular database structure.

Set [9]); the central parts of the C library consist of an interface to the parser and functions for traversing and querying the abstract syntax tree (AST) that it generates (see section 3). This interface is used by the XSUB (Perl e**x**ternal **sub**routine) "glue" code which is called by the Entry class, which in turn gives complete access to a single entry from a BIBTEX database (or any other textual database using the BIBTEX data language). At this level, no restrictions on aspects of database structure such as the allowed entry types, or lists of fields recognized for a particular type, are imposed. Imposing the conventions of a particular database structure falls on the third level of code, which consists of two classes that can be replaced at the user's desire. The entire Perl library (including the class hierarchy) is discussed in section 4.

Finally, several "proof of principle" example programs are provided to demonstrate the usefulness of a library for a general-purpose language as well as the benefits of an object-oriented approach. These are **btcheck**, **btsort**, and **btformat** (which eventually will be able to function as a drop-in replacement for BIBTEX's core program); they are discussed in section 5.

3 Lexical Analyzer, Parser, and C Library

The C component of btOOL, the **btparse** library, includes a lexical analyzer, a recursive-descent parser, a hash table for string abbreviations (macros), some basic string processing, query/traversal functions for the abstract syntax tree (AST) that results from parsing, and some more advanced string-processing that is specific to BibTeX-style bibliographic data.[3] The reference documentation for **btparse** is included as Appendix B.1; the following sections are ordered similarly to the reference manual.

3.1 Lexical and syntactic analysis

The lexical analyzer and parser were generated by DLG and ANTLR, two components of PCCTS (the Purdue Compiler Construction Tool Set) [9]. DLG and ANTLR are similar in concept to the venerable Unix utilities `lex` and `yacc`: DLG converts a list of (regular expression, token) pairs to a deterministic finite automaton (DFA) which reads a stream of characters and outputs a stream of tokens. The stream of tokens is then processed by a parser generated by ANTLR. The major difference between PCCTS and `lex`/`yacc` is that ANTLR generates a recursive-descent rather than a shift-reduce parser, meaning it handles the theoretically smaller LL class of languages, rather than the LALR class handled by shift-reduce parser generators. However, ANTLR provides a number of useful features that make it both easier to use and more powerful than `yacc`. Among those relevant to **btparse** are improved communication between grammar rules, automatic generation of abstract syntax trees, and "semantic predicates"—the ability for something outside the token stream (such as a bit of state set by the lexer) to affect parsing decisions.

One BibTeX problem not mentioned in section 1.1 is the lack of a formal definition of the language. This does not affect users the way it would for a full-blown programming language, but it means that implementors of new BibTeX parsers can never be entirely sure that they have covered everything. For instance, BibTeX has least one non-obvious and undocumented grammatical feature: the final field of an entry may be followed by a dangling comma. Until I encountered data files that used this feature—causing syntax errors with the then-current versions of my grammar—I was unaware of it, and had to adjust the grammar to fit the existing (and, according to BibTeX, perfectly valid) data. It is conceivable that there are other obscure and undocumented features of the original grammar which are not supported by **btparse**'s parser.

In the interests of future compatibility, the lexical and syntactic grammars used by **btparse** are described in section B.2. Nelson Beebe, the author of **`bibclean`**, has also published the grammars used by his lexer and parser [1]. Some superficial differences between the two pairs of grammars are due to the requirements of `lex`/`yacc` (which **`bibclean`** uses) vs. PCCTS; a more important difference is that Beebe's grammar recognizes whitespace as a syntactic element, rather than discarding it in the lexical analyzer. This allows the **`bibclean`** parser to be used for tools that wish to preserve the original form of the input, at the cost of complicating the grammar. This "input preservation" feature is a long-term goal for btOOL, but it is not currently supported.

btparse currently recognizes almost exactly the data language recognized by BibTeX 0.99. Some known and deliberate exceptions (any unknown exceptions are not deliberate) are:

- The same lexical element, a "name," is used as a catch-all for entry types, citation keys, and field and macro names, with slight variations to maximize compatibility with BibTeX. The rules for "names"

[3] Although this bends the "general-purpose library for a general-purpose data language" rule, implementing some of these string processing steps would have been very inefficient in Perl—thus, they were implemented in C, but in such a way that they do not interfere with the generality of the rest of the library.

are based on BibTeX's rules for citation keys, but with two differences. Most importantly, BibTeX allows all characters *except* a certain set; **btparse** reverses this by explicitly listing the allowed set of characters. Also, some characters that are accepted by BibTeX, but cause it to generate bad TeX code, are disallowed.

- Braces in strings must balance, regardless of the string delimiter (braces or double quotes). With BibTeX, an unbalanced string can result in the inclusion of incorrect TeX code in formatted bibliography entries.
- In-line comments (from a `%` to the next newline) are allowed, and may be either between entries or inside of them.
- `comment` entries may contain `@` characters. (`@` characters may also be present without effect between entries, as long as they are in an in-line comment.)
- "Junk text"—any non-comment text in between entries—is explicitly recognized by the lexer and generates a warning. BibTeX considers such text comments, and discards it silently; this can cause problems when earlier syntax errors cause actual data to be considered "junk."
- Many other warnings have been added for common errors, including a heuristic detection of runaway strings that helps pinpoint the source of the error much better. Some problematic data files that BibTeX accepts without complaint will generate warnings, and other files that trigger confusing syntax error messages with BibTeX will generate clearer warning or error messages.

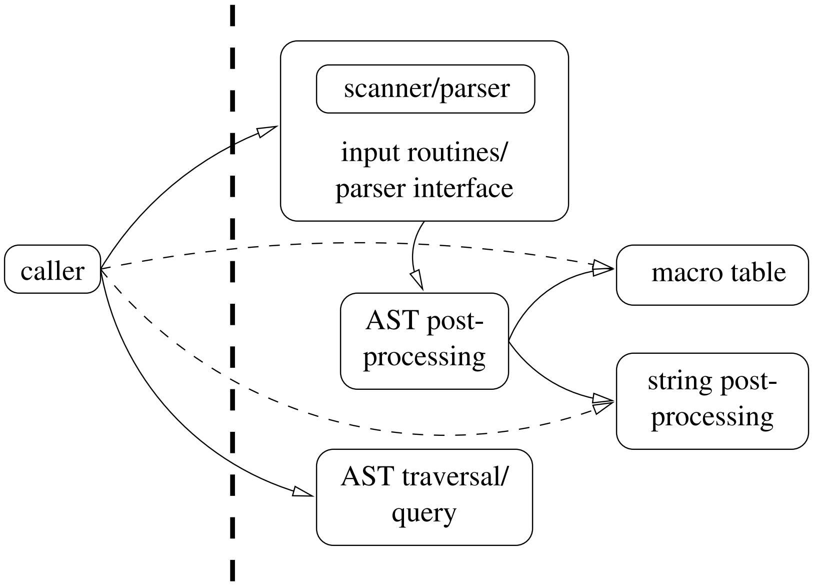
Like BibTeX, the **btparse** parser recognizes three entry types specially: `string` (for defining macros, otherwise known as string abbreviations), `preamble` (for supplying a bit of text, presumably LaTeX code, that must precede the BibTeX-generated bibliography), and `comment`. An entry with any other type is considered a "regular entry," a concept which is formalized by the introduction of the "entry metatype," which is one of "regular", "macro definition," "preamble," or "comment." The long-anticipated BibTeX 1.0 [11] will add a few new special entry types; support for them is written but currently unavailable pending the release of BibTeX 1.0, so I can determine the exact syntax of the new entry types and maintain compatibility with BibTeX.

3.2 Parser, AST, and macro table interface

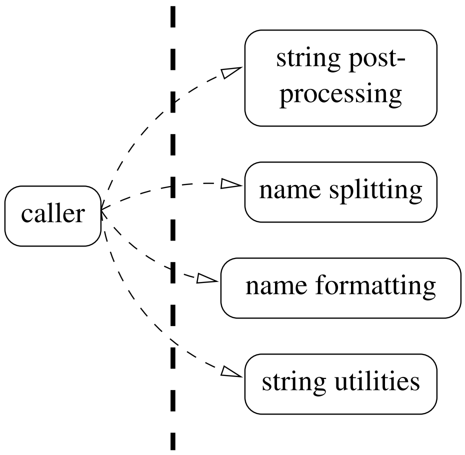
The rest of **btparse** falls into two broad camps, depicted in Figure 3: an interface to the parser and its output (the abstract syntax tree, or AST); and various string-processing functions that follow the conventions used by BibTeX, but don't assume that the strings being operated on are embedded in a **btparse** AST. The interface to the parser is described in full detail in section B.3; it consists of functions for reading and parsing either a single entry (from an open file or a string) or an entire file of entries.

Thanks to the automatic generation of abstract syntax trees provided by ANTLR, it is quite easy for the **btparse** parser to return such an AST to its caller. To make the AST useful, **btparse** provides a number of functions (documented in section B.4) for traversing and querying it. *Traversal* consists of returning the next entry in a list of entries, the next field in a list of fields, the compound value (list of simple values) associated with a field, and the next simple value in a compound value. *Query* consists of fetching the entry type, metatype, key, or the full text of a field, macro, `comment`, or `preamble` entry.

Application programs may also directly manipulate and query the **btparse** macro table. Normally, macro definitions are added when `string` entries are parsed, and fetched from the table when other entries use



(a) Parser, AST, and macro table interface



(b) String processing

Figure 3: Interface to the C library, **btparse**. Solid arrows denote typical, almost inevitable uses of the library; dashed arrows are for special-purpose, less frequent uses.

those macros. Using the functions described in section B.6, the caller may circumvent the normal process; this could be useful for pre-defined macros such as the month names provided by BIBTEX.

3.3 String-processing utilities

Every field in a BIBTEX entry is associated with a *compound value*, which is just a list of *simple values* separated by the string concatenation operator (`#`). A simple value is just a quoted string, a number, or a macro invocation. For example, the field "assignment"

```
date = 15 # "/" # jun
```

has three simple values: the number `15`, the string `/`, and the macro `jun`. Assuming the macro `jun` expands to the string `June`, then the actual value of the `date` field in this entry would be `15/June`.

By default, **btparse** performs "full processing" of strings: macros are expanded, numbers treated as strings, strings concatenated, and whitespace collapsed. The rules followed by both BIBTEX and **btparse** for collapsing whitespace are: 1) all whitespace characters in strings are converted to spaces, 2) spaces at the beginning and end of strings are deleted, and 3) strings of multiple spaces are collapsed to a single space.

However, it is possible to disable these features selectively before parsing an entry, so that the AST returned by the parser interface may have unexpanded macros, unconcatenated strings, and so on. In addition, an application may have strings gathered from other sources (such as user input) that it wishes to process in "the BIBTEX way." In either case, a stand-alone interface to the string post-processing code is needed. This interface is provided by **btparse**, and documented in section B.5.

In addition, **btparse** provides functions for processing author names from a bibliographic database. While this is inconsistent with the "general-purpose library for a general-purpose data language" philosophy, it is a justifiable exception on the grounds that the vast majority of existing BIBTEX-like databases are in fact bibliographic databases that expect BIBTEX's conventions for author names to be followed. Any program that processes such databases will need to be able to split up names in "the BIBTEX way," and some programs will need to be able to format them. Thus, **btparse** provides the name-splitting and -formatting functions described in sections B.7 and B.8.

As well as author names, BIBTEX has conventions for dealing with accented and non-English letters using a subset of TEX's syntax. A collection of miscellaneous string utilities that respects these conventions (and works identically to similar functions provided by BIBTEX's formatting engine), described in section B.9, is also included in **btparse**. Both the name-handling code and miscellaneous string utilities are entirely independent of **btparse**'s AST; they simply process C strings. Thus, **btparse**'s generality could be maintained by simply moving this functionality to a separate library, with a small inconvenience cost to the C programmer interested in both libraries.

4 Perl Library

Perl's traditional reputation is as a high-level scripting language geared towards managing files, text, and processes. It has always had a strong interface to the Unix services needed for most administrative tasks, excellent string-processing centred around a powerful regular expression engine, a simple yet flexible type system, and a rich syntax optimized for making common tasks easily and succinctly expressible. With the release of Perl 5 in late 1994, the language was greatly enhanced. The most important changes were the introduction of references to allow easy construction of complicated data structures, a simple, elegant mechanism for object-oriented programming, a standard interface for extending Perl with C code, and a new system for using library modules.

The fundamental unit of Perl's type system is the *scalar*, which, before Perl 5, was usually either an integer, a string, or a floating-point number. Like `awk`, these three kinds of scalars are readily inter-operable, and any scalar variable can have any kind of scalar value. Perl also has always had two compound types, *arrays* and *hashes* (formerly known as associative arrays). Arrays are ordered and keyed with integers, while hashes are unordered and keyed with arbitrary strings. With Perl 5, the scalar type was extended to allow a *reference* to any other value: scalar, array, hash, code (a subroutine reference), "glob" (a symbol table entry), and "blessed object." Using references, arbitrarily complex nested data structures can be built by embedding references to arrays or hashes within other arrays or hashes.

Blessed objects are the key to Perl's object-oriented mechanism. Typically, an ordinary-looking data structure is constructed to contain all the information an "object" (class instantiation) will need, and the value representing the whole data structure is "blessed" into the class which governs it. The blessed object is then returned to the class user, which then uses it as the key to accessing the class methods that operate on the new object. Two new subroutine call mechanisms allow calling the two types of methods: class methods (such as constructors), which don't operate on any particular object; and object methods, which do. Inheritance is implemented by looking up methods in each class' list of base classes: thus multiple inheritance is allowed, and inheritance may even be adjusted at run-time.

Since the btOOL class hierarchy is based on a C library, the mechanism for extending Perl with C code is also used. This system is centred around XSUB, a language that consists of chunks of C code which define Perl subroutines and are glued together by a preprocessor to create a C source file [13]. This file is then compiled and linked to create a shared object, which is dynamically loaded by Perl at run-time to give Perl programs access to a C library such as **btparse**.

A *module* in Perl 5 is simply a source file whose name and location are tied to the package that it provides. (A package in Perl is just a namespace.) For instance, the Text::BibTeX module supplies functions in the `Text::BibTeX` package; the actual source file is Text/BibTeX.pm relative to some directory in Perl's search path. Modules typically provide either a group of related functions, or a class with a constructor and object methods that operate on the object returned by the constructor. Most of the modules in btOOL's Perl library provide one or more classes; only Text::BibTeX is a "plain" module with no methods.

One innovation that has come along in the wake of Perl 5 affects not the language, but the library: the Comprehensive Perl Archive Network (CPAN) and the closely associated Perl Module List provide guidelines for creating, documenting, and naming modules; anyone wishing to share their modules with the rest of the world can follow these guidelines and make use of the module list and CPAN for easy dissemination [3, 2]. To cooperate with the rest of the world, the class names in btOOL's Perl library all start with Text::BibTeX; thus, references to the Entry and File classes in the following sections should be interpreted as Text::BibTeX::Entry and Text::BibTeX::File. (The shorter form will be used in the text in the interests of brevity; all example code will use the full class names.)

The full structure of the btOOL Perl library, along with its interface to the C library, **btparse**, is shown in Figure 4. The Text::BibTeX module serves two purposes: it dynamically loads all the compiled C code (**btparse** itself, as well as the layer of code connecting it to Perl), and it provides some miscellaneous utility functions that don't fit in the class hierarchy. Having an "umbrella" module for the whole library is also convenient for documentation—the manual page for Text::BibTeX (section C.1) gives an overview of the whole library that is not available in the manual pages for the individual classes.

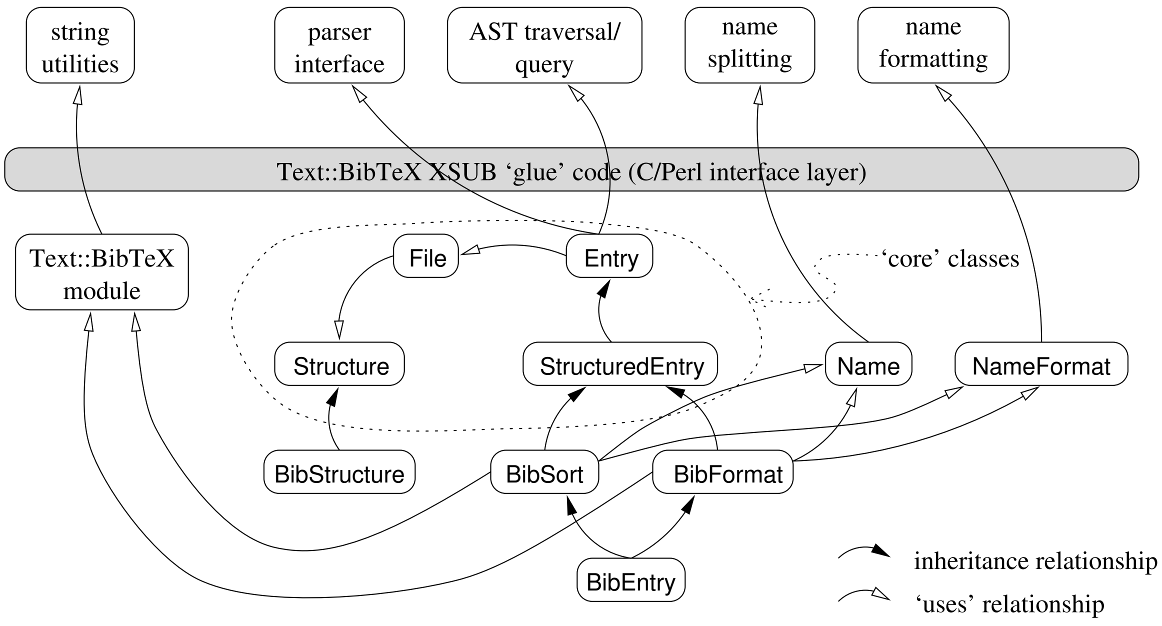


Figure 4: The modules in the btOOL Perl library, including all class relationships and dependencies between the Perl modules and the C library, **btparse**.

4.1 The core of the library: the Entry and File classes

The heart of the class hierarchy is the Entry class, the central clearinghouse of information coming from and going to BIBTEX databases. Each entry from a database (whether a `string`, `comment`, `preamble`, or regular entry) is represented by one Entry object. An Entry object is initialized by parsing data from a file-handle, a string, or a File object. The user may query or change the object's type, metatype, key (for regular entries), string value (comment and preamble entries), and field/macro values (regular and macro definition entries). Finally, the entry may be written to a file or string in BIBTEX syntax.

The Entry class is also an important connecting point in the web of class relationships shown in Figure 4. Every entry that is parsed from a File object keeps a reference to that File object; it is through this reference that the Entry has access to the powerful mechanisms for enforcing and using the database structure (see section 4.2). Also, Entry serves as the ultimate base class for all "structured entry" classes, which form an important part of the mechanism for imposing database structures.

The File class provides an abstraction for entire BIBTEX databases (where a database is a single file). This mainly provides a central location for attributes that properly belong to an entire database rather than a particular entry. Mainly, this means the database structure, which determines the *structure module* used to process entries. In addition, the File class provides an entry point for the client application to pass *structure options* to the actual structure module, which may be used to customize the behaviour of the module.

Two other classes, Structure and StructuredEntry, are considered part of the "core" hierarchy, since they are not specific to any particular database structure. However, they serve only as base classes for the user-supplied classes which define particular structures, so are described more fully in the next section.

4.2 Defining a database structure

Treating BibTeX databases as abstract entities, with no requirements on the types of entries or the fields in them, is useful to a certain extent. This in fact is how the core engine of BibTeX works, and it is how the portions of btOOL discussed so far work. Eventually, though, the programmer wishes to impose some restrictions on the structure of the data. The structure to which a database conforms consists of the set of allowed entry types and, for each type, sets of required, optional, and constrained fields. The three sets of fields together make up the set of all fields that are actually used by the processing code; fields outside this set may be silently ignored or may generate a warning or error, depending on the application. Required and optional fields are independent of any other field; required fields for entry type `foo` must be present in all entries of type `foo`, and optional fields will be used if they are present, but have no other effect. A set of constrained fields represents a middle ground: the presence of some members of the set determines whether other members may be present. This formalizes such relationships as "either `x` or `y` may be present, but not both" or "one or both of `x` or `y` must be present." Constraints are based solely on counting how many fields from the set are present—it is not possible to express relationships like "if `x` is present, then `y` must be present."

For example, the database structure defined by BibTeX 0.99 includes entry types such as `article`, `book`, and `inproceedings` (along with 10 others). One of these types, `book`, has required fields `title`, `publisher`, and `year`; its optional fields include `address` and `edition`. Furthermore, the `book` type has two pairs of constrained fields: `author` and `editor`, and `volume` and `number`. The nature of the constraint is different for each pair: any given `book` entry must have either (but not both) of the `author` or `editor` field present, but may optionally have one or the other, or neither—but not both—of `volume` and `number`. In the BibTeX style file that defines this structure, only the list of all fields recognized across all types is explicitly given. The list of known entry types is implicit in the set of "type functions" (each of which handles the formatting for one type of entry), and all the "required field" and "constrained fields" relationships are buried in the actual formatting code. Thus, the description of a database structure in BibTeX is coupled fairly tightly with the formatting decisions for that database.

In btOOL, this coupling is reduced—not eliminated—by making the definition of the database structure more explicit. The coupling cannot be eliminated, since at some point it is still necessary to have code to handle each type of entry and the fields in each entry. This "reduce but don't eliminate" problem is solved by defining the database structure in one class and the processing (sorting, formatting, ...) code in another class, but making a single module that provides both classes. Recall that a *module* in Perl is simply a source file whose name and location are derived from the module name. By convention, such a module provides subroutines in the package (namespace) with the same name as the module: the Text::BibTeX module provides a subroutine `Text::BibTeX::purify_string`. Often, a module will provide a class (which is just a package that happens to have methods rather than regular subroutines) with the same name as the module: the Text::BibTeX::Entry module provides the class Text::BibTeX::Entry, which in turn provides such methods as `parse` and `set_type` (fully documented in section C.3). However, there is no hard-and-fast requirement about the package or class that a module must provide, so btOOL steps outside of convention by requiring that a *structure module* provide two classes: the *structure class* and the *structured entry* class. Every database structure must have one of each, although the programmer defining a structure may simply derive his class from an already-defined class, thus inheriting certain properties of an existing database structure.

4.2.1 Structure classes

The structure class is responsible for defining the database structure and for other properties that apply to a whole database, such as handling user-supplied *structure options* that drive the processing (e.g., sorting or formatting options). As shown in Figure 4, structure classes (such as BibStructure) are normally derived from the Structure class in the core hierarchy; this class provides methods for defining the entry types and sets of fields, querying the database structure, and dealing with structure options. Thus, a user-defined structure class only has to define the structure and supply a set of default options, leaving the rest up to the methods inherited from Structure. In fact, a typical structure class will not provide any methods that are called from outside of the btOOL class hierarchy—if a user application does need to know anything about the database structure, it can find it out using inherited methods provided by Structure.

4.2.2 Structured entry classes

The structured entry class provides methods that operate on single entries taken from a database which (presumably) conforms to the database structure. Again consulting Figure 4, we see that structured entry classes such as BibEntry inherit from the StructuredEntry class, which in turn inherits from the Entry class. StructuredEntry adds methods that are not particular to any database structure, but assume that the entry does have a structure which is defined by a structure class. These include methods for checking that an entry conforms to a structure, and for coercing it into conformance by deleting some fields (those that exceed field constraints) and adding empty versions of others (missing required fields, or those that should be present to meet field constraints).

The actual user-supplied structured entry class doesn't have any particular requirements regarding which methods must be supplied or how they should behave. This is to allow btOOL to be used for as wide a range of applications as possible; for instance, any database structure that is intended to describe publications with the aim of creating bibliographies will presumably follow certain conventions (such as supplying methods for generating sort keys and for formatting entries), but one would not make the same requirements of a completely unrelated database. The features required of a structured entry class will be specified by the application that uses those features, rather than by a library speculating about what future applications might do.

4.2.3 An example database structure: the Bib module

btOOL provides a sample module implementing the `Bib` database structure, both as an example to other programmers wishing to implement new structures, and to duplicate the functionality of BibTeX 0.99 in a completely different setting. The naming conventions for structure modules dictate that the Bib module provide the classes BibStructure and BibEntry, as shown in Figure 4. (It is possible to use different class names, but using the defaults minimizes bureaucratic overhead.) Due to the complexity of sorting and formatting bibliographic entries, most of BibEntry's work is actually done by two intermediate classes, BibSort and BibFormat; the fact that this abstraction is invisible to the user of the `Bib` structure is an obvious benefit of the object-oriented paradigm.

Currently, the `Bib` structure only handles sorting and formatting of bibliographic entries; alphabetic labels in the style of BIBTEX 0.99 are not yet supported. Since a structured entry class only deals with one entry at a time, the act of actually sorting entries is left to external applications; the BibEntry class (via methods inherited from BibSort) just generates sort keys that reduce the comparison of complicated bibliographic entries to a simple string comparison. The formatting of entries likewise is not completely handled by the methods of the BibFormat class; the calling application is expected to supply markup text that is interpolated around the database-derived strings to turn them into text appropriate for inclusion in a document written in the user's choice of markup language (e.g., LATEX or HTML). If no markup text is supplied, then plain text will be the output format.

4.3 Miscellaneous classes

Finally, to facilitate the emulation of BIBTEX 0.99, two classes peculiar to processing bibliographic data are provided. These are Name and NameFormat; as shown in Figure 4, these two classes provide an intermediary between the name-splitting and -formatting code in **btparse** and the bibliography sorting and formatting methods in BibSort and BibEntry.

The Name class provides an abstraction for a single BIBTEX-style author name, which includes four "parts" designed to handle most names in the Western European languages: 'first', 'von', 'last', and 'jr'. (BIBTEX's name convention is actually flexible enough to handle most non-Western names, at least in their conventional anglicized forms.) The syntax for individual names is explained in section B.7, from the **btparse** documentation, and the Perl interface is described in section C.6.

The NameFormat class abstracts the task of formatting author names, which in a sense is the opposite of splitting them up into parts. The reason for the split-and-reformat approach is to ensure that names are consistently printed: in the database, a variety of forms are allowed, but in the final output, all names should be printed the same way (e.g., "last name first" or "first name first," initials or full name, abbreviations punctuated or not, and so forth). Name formatting is described in sections B.8 and C.7.

5 Example Applications

Three example programs—**btcheck**, **btsort**, and **btformat**—have been written to demonstrate the capabilities and utility of btOOL, particularly of the Perl class library. These are general-purpose tools for processing bibliographic databases—or rather, for processing databases that conform to a structure whose structured entry class provides certain methods that are expected of bibliographic database structures. Using the Bib structure described in section 4.2.3, they can impose the conventions and operations of BibTeX 0.99's standard styles. Given another structure module that supplies the required methods, these programs could just as easily operate on any database structure, bibliography-related or not. **btcheck**, in particular, makes no assumptions about the structured entry class being used, and should work with any structure module regardless of the entry methods it provides.

5.1 Syntax- and structure-checker: btcheck

Two of the general-purpose utilities are quite short—they are essentially wrappers for functionality provided by the btOOL Perl library. **btcheck** simply parses an entire BibTeX file, checking the syntax and structure of each entry; the complete source code is shown in Figure 5.

```
1    use Text::BibTeX (':metatypes');
2
3    my ($filename, $structure, $bibfile, $entry, %seen_key);
4    die "usage: btcheck file [structure]\n" unless @ARGV == 1 || @ARGV == 2;
5    ($filename, $structure) = @ARGV;
6    $structure ||= 'Bib';
7
8    $bibfile = new Text::BibTeX::File $filename or die "$filename: $!\n";
9    $bibfile->set_structure ($structure);
10
11   while ($entry = new Text::BibTeX::Entry $bibfile)
12   {
13      next unless $entry->parse_ok and $entry->metatype == BTE_REGULAR;
14      my $key = $entry->key;
15      $entry->warn ("repeated entry key \"$key\"") if $seen_key{$key};
16      $seen_key{$key} = 1;
17      $entry->check;
18   }
```

Figure 5: Complete source code for **btcheck**, a btOOL-based syntax- and structure-checker for BibTeX databases. This program simply reads, parses, and checks the structure of every entry in a database file.

Line 1 tells Perl to load the Text::BibTeX module, and to import the set of symbols tagged by `metatypes` (this lets us use the constant `BTE_REGULAR` rather than `Text::BibTeX::BTE_REGULAR` later on). In addition to providing many constants (such as `BTE_REGULAR`) that correspond to enumerated type values in the C library, **btparse**, the Text::BibTeX module also dynamically loads **btparse** and the compiled XSUB "glue" code that binds it to Perl. And, since much of that code is useless without the Text::BibTeX::Entry and Text::BibTeX::File classes, they are loaded as well.

Line 3 declares all the variables that will be used in the program. Lines 4–6 handle the command line arguments: there must be one or two of them, of which the first (required) is the database file to parse, and the second (optional) is the database structure to use. If the second argument, `$structure`, is not given, then it defaults to `'Bib'`.

Line 8 creates a new Text::BibTeX::File object, opening the file named on the command line in preparation for reading BibTeX entries from it. If the File constructor is unable to open the file, it returns a false value, which is caught by the `or` clause—if this happens, the program crashes with a brief error message.

Line 9 is crucial—without it, we would only be able to write a syntax-checker, leaving any structural errors in the database undetected. As implied in Figure 4, a Structure object may be associated with each File object; in turn, each Entry object may refer to a File object. To have access to the actual database structure and the structure-checking methods provided by the StructuredEntry class, both of these conditions must be met. Line 9 takes care of the first one: associating a Structure with the File.

The loop control statement on line 11 takes care of the second condition necessary for accessing the database structure: it associates each Entry object with a File method. More accurately, it creates a new object in the Entry class, reads and parses an entry from the file associated with the File object `$bibfile`, and "re-blesses" the new object into the structured entry class that the Bib structure provides. This "re-blessing" is a key step; it allows us to use a structured entry class without knowing its name—we simply pass the name of a structure to the File method `set_structure`, and everything else is taken care of for us: the structure module is loaded, and all future Entry objects parsed from this file are actually made objects of the structured entry class. This is a slightly unconventional way for an object constructor to behave, but the freedom from having to know the name of the structured entry class is enormously convenient.

A side-effect of parsing the entry into the `$entry` object is that any syntax error or warning messages are printed, the entry type and metatype are stored in the object, and a "parse status" flag in the object is set. Once inside the loop, at line 13, we immediately jump to the next entry if there were any parse errors or the entry is anything other than a "regular" database entry. (Macro definitions, comments, and preambles should all be syntax-checked—but it makes no sense to check their structure, since they lie outside the actual database.)

Lines 14–16 take care of checking that entry keys are really unique. First, the key for the current entry is fetched using the `key` method. Then, we print a warning (using the `warn` method, which takes care of printing the filename and line number(s) that are the source of the warning) if this key was seen before according to the `%seen_key` hash (associative array), which is updated for each entry on line 16. Note that this simple check is *not* done in the library on the principle that entries are independent—it is the application program that loops over a set of entries, so it is the application's responsibility to handle any possible interactions between entries.

Finally, line 17 performs the actual structure-checking. The `check` method is provided by the StructuredEntry class; recall that `$entry` is blessed into the BibEntry class (or some other structured entry class, depending on `$structure`), which derives from StructuredEntry. `check` first ensures that the entry's type is valid, then makes sure that all required fields are present, and finally checks field constraints. For example, if `$entry` were a `book` entry from a Bib database, then it would check that each of the `title`, `publisher`, and `year` fields (the required fields) were present, that exactly one (not both) of `author` and `editor` were present (constrained fields), and that either zero or one (not both) of `volume` and `number` were present. (See section C.8 for complete documentation of the Bib database structure.)

btcheck is an example of how much can be accomplished in a very short amount of code, given a sufficiently high-level language and powerful library. The code shown in Figure 5 is almost exactly the "production" version that is distributed with btOOL; the only change is to make the structure a true command-line option, and to allow multiple input files to be specified.

5.2 Database sorter: **btsort**

The next example, **btsort**, loops over all entries in a file just like **btcheck** does, but additionally stores the Entry objects in a list for later sorting. From the source code in Figure 6, we see that its initialization section in lines 1–9 is essentially the same as **btcheck**'s. The one difference is that **btsort** allows the user to supply a set of structure options to control the sorting. Options are specified as *key-value* pairs; for instance, the user would tell **btsort** to sort on year (rather than on author name, the default) by specifying `year` for the `sortby` option:

```
btsort test.bib Bib sortby year
```

Currently, `sortby` is the only option that supported by the Bib module that affects sorting. If the user supplies other recognized options, they are ignored; unknown options cause an error.

```
1   use Text::BibTeX (':metatypes');
2
3   my ($filename, $structure, @options, $bibfile, $entry, %sortkey, @entries);
4   die "usage: btcheck file [structure [options]]\n" unless @ARGV >= 1;
5   ($filename, $structure, @options) = @ARGV;
6   $structure ||= 'Bib';
7
8   $bibfile = new Text::BibTeX::File $filename or die "$filename: $!\n";
9   $bibfile->set_structure ('Bib', @options);
10
11  while ($entry = new Text::BibTeX::Entry $bibfile)
12  {
13     next unless $entry->parse_ok && $entry->metatype == BTE_REGULAR;
14     $entry->check;
15     $sortkey{$entry} = $entry->sort_key;
16     push (@entries, $entry);
17  }
18  $bibfile->close;
19
20  @entries = sort { $sortkey{$a} cmp $sortkey{$b} } @entries;
21  $entry->print while $entry = shift @entries;
```

Figure 6: The second example program, **btsort**. This reads and stores all entries from a single database file, computes sort keys while reading the entries, and sorts and prints out the entry list. This version is somewhat simplified—macro definition, comment, and preamble entries are not handled at all.

(Structure options can in fact customize any aspect of a database structure, so in principle **btcheck** ought to support them. However, they are currently unnecessary in that context, as the Bib structure module has no options that affect the database structure or how it is checked. See section C.8 for a description of all options supported by the Bib module.)

The loop body, in lines 13–16, starts out in much the same way as that of **btcheck**. Note that we immediately reject any "non-regular" entries (macro definitions, comments, and preambles); in a real database sorter, this would be unacceptable, as the user expects a sorted database to contain everything the input database contained. The "production" version of **btsort** handles this by assigning artificial sort keys to these entry types that place them either at the top of the file (for `string` and `preamble` entries) or preserve their location relative to the "regular" entries (for `comment` entries); this detail is omitted in Figure 6 to save space.

The key to the sorting is line 15, where we compute the sort key (using the `sort_key` method, supplied by the BibSort class). The sort key is computed by reducing the essentials of the entry—which vary according to entry type and which fields are present—to a single string; see section C.10 for details on sort keys for the Bib structure. The sort key is stored in a hash (associative array), `%sortkey`, for later reference, and the actual Entry object is pushed onto a list of all entries in the file.

Once out of the loop, we close the file—this is not really necessary when we just process a single file and quit, but it is good practice. Line 20 performs the actual sort, using Perl's `sort` operator. This operator takes a list of values to sort (`@entries`) and an optional block of code that returns the result of comparing two values from the list. In this case, we fetch the pre-computed sort keys for the two entries (`$a` and `$b` within this little chunk of code) from the `%sortkey` hash, and compare them as strings using the `cmp` operator. Once the sort is complete, line 21 loops over all entries in the now-sorted list, using the `print` method to print each one to standard output.

A subtle point that might be missed in a casual examination of Figure 6 is the nature of the strings used as keys to the `%sortkey` hash. One might expect that, since BibTeX entries themselves have (supposedly) unique keys, those keys would be eminently suitable to use for storing and fetching each entry's sort key. This is not done in **btsort** for two reasons: first, it would require checking that the entry keys really are unique; and second, it avoids the overhead of repeatedly calling the `key` method on the `$entry` object. Instead, **btsort** uses the object reference itself as the hash key. This may seem odd: hash keys must be strings, and references in Perl are a third kind of scalar—neither strings nor numbers. However, one of Perl's fundamental concepts is that all scalar values may be interpreted as strings, and references are no exception. For example, a reference to an array might be "stringified" as `ARRAY(0x81ea53c)`, while a Text::BibTeX::Entry object such as `$entry` in **btsort** might become `Text::BibTeX::BibEntry=HASH(0x83f6850)`. The key point here is that these string representations of reference values include the unique address of the object in memory, making them immediately usable as hash keys with no need for a uniqueness check. Using this trick is not without its risks: if we later attempt to treat the hash key as a reference to a Entry object, it will fail because the hash key is just a string, not a reference; and, if we attempt to make the "entry→sort key" relationship persistent by writing it to disk and later reading it back, the "stringified" object references will no longer correspond to the same object in memory—in fact, they probably will not even refer to a valid address. In the limited scope of **btsort**, though, the convenience and efficiency of using object references as hash keys outweigh these factors.

5.3 Database formatter: **btformat**

The third general-purpose utility, **btformat**, is considerably more complex than **btcheck** or **btsort**. In fact, it is planned that **btformat** will grow into a complete replacement for the BibTeX formatting "engine," the core program which is driven by a style file. (This engine is the non-customizable portion of BibTeX, so any limitations in it are much harder to deal with than limitations in a particular style file.) Rather than being driven by a style file, though, **btformat** is driven by a structure module. In particular, the module must provide a structured entry class with a `format` method that conforms to a particular convention (documented in section C.9). For the Bib structure, this method is actually provided by BibFormat, which (as shown in Figure 4) is a base class for the structured entry class BibEntry.

In addition, **btformat**'s user interface and output style are much more flexible than BibTeX's. In the standard BibTeX styles, the underlying assumption is that a bibliography is being generated for an English-language LaTeX document. The lack of support for human languages other than English is not a fundamental limitation; it has been corrected in many non-standard style files, most notably the `custom-bib` "meta-style file" package [6]. Markup languages other than LaTeX could also be supported in the style file, as in the `bib2html` package [7].

A more serious limitation of BIBTeX is its mechanism for selecting entries, which—being part of the user interface rather than a formatting detail—is hard-coded into the formatting "engine," rather than being part of a customizable style file. BIBTeX works closely with LaTeX in this matter: on the first pass through a document, LaTeX records every citation (among other things) in its "auxiliary" file. After this pass, the user runs BIBTeX's formatting engine, which scans the auxiliary file and uses the citations recorded there to select entries from the bibliographic database for formatting. (The user also specifies the database(s) and style file in the LaTeX document, which are duly recorded in the auxiliary file for BIBTeX's use.) Further LaTeX passes incorporate the BIBTeX-generated reference list to make a document that includes only cited publications in its reference list. This is an excellent scheme for the "citation-reference list" model which is the typical style of document for which LaTeX and BIBTeX are used.

However, there are many circumstances in which this document model is not used, and BIBTeX's file-driven entry selection becomes unnecessarily cumbersome. For instance, when generating comprehensive publication lists (such as for an academic *curriculum vitæ* or a grant application), one typically wants simply to format all entries in a database. (One might also wish to select all entries with a particular author, or with particular keywords, or from some range of years. **`btformat`** deliberately does not handle these options—general querying of a BIBTeX database was deemed a sufficiently complicated task to justify the eventual creation of a fourth general-purpose utility, tentatively named **`btselect`**.) Thus, **`btformat`** takes a more explicit approach to user interaction: the database file(s) to process are specified on the command line, and the various command-line options control all aspects of the formatting. These include the structure module (defaults to `Bib`), the entries to format (defaults to the entire file), and the markup language to use (LaTeX 2.09, LaTeX 2_ε, and HTML are currently supported). The selection of markup language is another important feature but, as mentioned above, BIBTeX is not intrinsically limited to generating LaTeX markup—allowing customizable markup is a feature of the Bib module, and any BIBTeX style file (or style file generator, like `custom-bib`) could offer the same feature. However, the support for particular markup languages is a feature of **`btformat`**, and making it generate anything other than LaTeX or HTML would involve overriding every individual "markup option"—a tedious, but not impossible, task.

In the interests of backwards compatibility, a future version of **`btformat`** will also offer a "BIBTeX emulation" mode, in which it will read all required information (which entries to format, database file(s) to read, and bibliography style) from a LaTeX auxiliary file. Translating the BIBTeX style to a structure module (and some set of options) will be non-trivial; at the very least, **`btformat`** will have to recognize the standard BIBTeX style names (`unsrt`, `plain`, `alpha`, `abbrv`) and convert each to a particular configuration of options for the Bib module. Unfortunately, other BIBTeX styles will remain unsupported, unless structure modules are written to emulate them.

For the time being, **`btformat`** is deliberately limited to its "format all entries" mode. While it is intended for **`btformat`** eventually to be usable as a drop-in replacement for BIBTeX's core program, doing so before an assurance of backwards compatibility exists between all relevant levels of btOOL and BIBTeX 0.99 would run the risk of creating annoying incompatibilities. This could in turn lead to a divergence of database formats, some geared for use with BIBTeX and some with btOOL. This would defeat the whole purpose of providing backwards compatibility, so until such compatibility can be guaranteed by a rigorous regression test suite, **`btformat`** will remain an example application for experimental use.

6 Conclusions

btOOL includes three significant components. First is a new lexer/parser written in C, with a complete interface to parsed entries for C programmers. Included with this C interface are a number of functions for performing various string-processing operations in "the BibTeX way," which are necessary for a faithful emulation of BibTeX.

For a truly useful interface, though, C is too low-level and clumsy for convenient processing of BibTeX databases. Thus, the second component: a simple, object-oriented Perl interface to BibTeX databases and entries. With this interface, arbitrary database operations are easily implemented, with the full facilities of a modern, high-level, text-oriented language at the programmer's service. One could combine this mid-level interface with the string-processing utilities provided by the C library, **btparse**, and rewrite BibTeX 0.99's formatting engine and style files.

However, the use of an object-oriented language suggests a more elegant approach that allows us to create extensible database structures and "style file" analogues through inheritance. This, then, is the third component: the intertwined pairs of "structure classes" and "structured entry classes," all provided by structure modules. More important than the actual classes that emulate the database structure and bibliography formatting of the BibTeX 0.99 standard styles is the framework and set of conventions for building future structure modules.

One might question the utility and necessity of a complete set of programming libraries for what are generally simple database operations, with the occasional tricky string operation—especially given that BibTeX, BibTool, and the **`bibclean`** suite already address most of the common problems. After all, the majority of BibTeX users get what they need out of BibTeX itself, its standard styles, and possibly some third-party styles released independently of BibTeX—showing that it is possible to extend BibTeX. For operations other than bibliography formatting, **`bibclean`** provides a very thorough heuristic check of a bibliographic database, and BibTool allows for highly customizable database checking, as well as generation of new fields and entry keys. What more could one need?

One reason for attempting to supplant BibTeX is that the days of special-purpose programming languages are drawing inexorably to a close, and replacing BibTeX's current arcane programming interface with one accessible to more programmers will ultimately provide programmers and users alike with greater choice and flexibility. Providing a new interface is not the insurmountable task it might have once seemed, given the profusion of well-designed, high-level, modular, portable languages the last few years have seen.

However, there is more to btOOL than an exercise in object-oriented Perl programming with an interface to a C library. BibTeX's data language is itself extensible and applicable to a wide range of database formats where easy manual editing and a flexible structure are more important than raw speed. With btOOL, one has access to the data at a point before any assumptions are made about its structure (allowed entry types and expected fields), and one can easily write new structure modules to impose new conventions for new applications. This would be possible with BibTeX's "style file" language, but needlessly difficult because of the use of a stack-based programming language with little documentation and no development support; furthermore, it would have been pointless, because BibTeX controls access to the data in a way geared exclusively towards producing LaTeX documents. As for BibTool (at least, the version evaluated for the 1996 grant proposal described in section 1.2) and **`bibclean`**, they are very powerful and general tools for a wide range of bibliography-processing tasks. However, BibTool lacked extensibility, and **`bibclean`**'s companion programs lack customizability—and they both lack a mechanism for describing wholly new database structures (or even structures derived from the "standard" bibliography structure, but with new entry types and fields added). The fact that BibTool is moving towards a programmer's interface to a high-level language—Tcl—shows that the closed-box, dedicated mini-language solution it formerly used is a dead end.

There are two distinct dangers to the btOOL approach. The first is in backwards compatibility with BIBTeX 0.99—any time one programmer sets out to emulate the work of another, it is inevitable that there will be differences, be they deliberate or accidental. There are already some deliberate differences in the language accepted by btOOL's parser, and there are differences in the low-level string-processing that have to be rooted out and corrected (or made optional; some of btOOL's behaviour can be justified as "better" than that of BIBTeX in the same situation, but it is crucial to provide a "warts and all" compatibility mode). This is one reason that **`btformat`** is not yet fully fleshed out: providing a drop-in replacement for BIBTeX without ensuring that it is a completely faithful emulation opens the door to a dangerous diversion of paths, with some databases being tweaked so that BIBTeX will do "the right thing" while others are made to work better with btOOL. Currently, **`btformat`** is better suited to experimental work by others interested in taking the BIBTeX data language in new directions, rather than day-to-day use by LaTeX users. It will remain that way until a rigorous suite of regression tests can ensure the utmost compatibility between the behaviour of btOOL's Bib module (possibly in a "complete emulation" mode) and BIBTeX 0.99's standard style files.

The second potential pitfall is for future divergence: ironically, if it becomes *too* easy to invent new database structures and to write formatting code for them, then there exists the danger of mutually incompatible databases circulating. That danger is certainly there with BIBTeX 0.99—btOOL is, in principle, no more or less extensible than it—but the difficulty of writing new formatting programs has apparently prevented this divergence of databases. This danger can be minimized if structure writers conscientiously support all existing entry types and fields, only adding to rather than subtracting from the structure definition.

And, it should be noted that with btOOL, non-programmers are still unable to extend database structures and formatting programs. The canonical Bib structure module is highly customizable by means of run-time options (and will become more so in the future), meaning that changing most superficial sorting/formatting options is trivial; if future structure modules follow this example, then the problem of customizability is addressed to at least as satisfactory degree (and with less overhead for the user) as by the `custom-bib` package. The lack of extensibility for non-programmers, though, is a problem shared with BIBTeX (including `custom-bib`), and given the complexity of bibliography formatting—not to mention the range of other operations one might wish to perform on a BIBTeX-like database—real programming is probably inevitable. At least with btOOL, a widely-used, general-purpose language is available for this programming.

There remains important work to be done on btOOL. Many more test programs need to be written to ensure that all aspects of the system are subjected to full regression testing. More example programs need to be written, both to find holes or outright bugs in the interface, and to help new programmers learn how to program with btOOL. Finally, it would be nice to emulate **`bibclean`**'s ability to preserve comments and whitespace via additions to the grammar, although this might unacceptably complicate the parser, the syntax tree, and the interface to it.

To exploit the full potential of the Bib module and its position as a starting point for other, more complete database structures, it is important to verify rigorously its compatibility with BIBTeX 0.99's standard styles. Doing so will open the way to completing **`btformat`**, by adding all the planned features described in section 5.3. In conjunction with this will be the fleshing out of the framework for new structure modules—many BIBTeX styles apart from the four standard styles supplied with BIBTeX 0.99 are in use, and emulating some of the more popular ones would be a good demonstration of the real-world applicability of both btOOL and Perl to bibliography formatting. Along the other dimension of structure modules, new style files exist for databases that describe a wider range of publications—software packages, Internet resources, USENET posts—than BIBTeX's standard style files. These can easily be implemented for btOOL as structure classes. Ultimately, the existence of a new programming interface for a modern, modular programming language to BIBTeX's extensible and flexible data language may yet prove that it was well ahead of its time.

Appendix A Availability, Further Information, and Licensing

The two major components of btOOLare available separately. Currently, they are both modified and released in tandem, but it is intended that the C library, **btparse**, will stabilize while the Perl library, Text::BibTeX, continues to undergo development. Both libraries are freely available and redistributable, and covered by the GNU Library General Public License as published by the Free Software Foundation. (Text::BibTeX may, at the user's discretion, instead be covered by the Artistic License included with Perl.) They are *not* in the public domain; the software and all documentation are copyrighted by the author.

This report describes version 0.3 of both libraries, which may be downloaded from

```
ftp://ftp.bic.mni.mcgill.ca/pub/btOOL/
```

as the files btparse-0.3.tar.gz and Text-BibTeX-0.3.tar.gz (although a newer version may be available by the time you read this).

Also, up-to-date information on btOOL may be found at

```
http://www.bic.mni.mcgill.ca/~greg/btOOL/
```

Appendix B C Library Documentation

The C portion of btOOL, **btparse**, is documented in a series of manual pages. There is one such page for each major component of the library, plus an introductory overview and a semi-formal description of the language recognized by **btparse**'s lexical scanner and parser. This documentation is available with the library, and is also presented here. Like the software, this documentation is copyrighted by the author; it is freely available and redistributable, with full source available as part of the **btparse** package.

B.1 Introduction to and Overview of btparse

B.1.1 Synopsis

```
#include <btparse.h>

/* Basic library initialization / cleanup */
void bt_initialize (void);
void bt_free_ast (AST *ast);
void bt_cleanup (void);

/* Input / interface to parser */
void   bt_set_stringopts (bt_metatype_t metatype, ushort options);
AST * bt_parse_entry_s (char *    entry_text,
                        char *    filename,
                        int       line,
                        ushort    options,
                        boolean * status);
AST * bt_parse_entry   (FILE *    infile,
                        char *    filename,
                        ushort    options,
                        boolean * status);
AST * bt_parse_file    (char *    filename,
                        ushort    options,
                        boolean * overall_status);

/* AST traversal/query */
AST * bt_next_entry (AST * entry_list,
                     AST * prev_entry)
AST * bt_next_field (AST *entry, AST *prev, char **name);
AST * bt_next_value (AST *head,
                     AST *prev,
                     bt_nodetype_t *nodetype,
                     char **text);

bt_metatype_t bt_entry_metatype (AST *entry);
char *bt_entry_type (AST *entry);
char *bt_entry_key (AST *entry);
char *bt_get_text (AST *node);

/* Splitting names and lists of names */
bt_stringlist * bt_split_list (char *   string,
                               char *   delim,
```

```
                                  char *   filename,
                                  int      line,
                                  char *   description);
void bt_free_list (bt_stringlist *list);
bt_name * bt_split_name (char *  name,
                        char *  filename,
                        int     line,
                        int     name_num);
void bt_free_name (bt_name * name);

/* Formatting names */
bt_name_format * bt_create_name_format (char * parts, boolean abbrev_first);
void bt_free_name_format (bt_name_format * format);
void bt_set_format_text (bt_name_format * format,
                        bt_namepart part,
                        char * pre_part,
                        char * post_part,
                        char * pre_token,
                        char * post_token);
void bt_set_format_options (bt_name_format * format,
                           bt_namepart part,
                           boolean abbrev,
                           bt_joinmethod join_tokens,
                           bt_joinmethod join_part);
char * bt_format_name (bt_name * name, bt_name_format * format);

/* Construct tree from TeX groups */
bt_tex_tree * bt_build_tex_tree (char * string);
void          bt_free_tex_tree (bt_tex_tree **top);
void          bt_dump_tex_tree (bt_tex_tree *node, int depth, FILE *stream);
char *        bt_flatten_tex_tree (bt_tex_tree *top);

/* Miscellaneous string utilities */
void bt_purify_string (char * string, ushort options);
void bt_change_case (char transform, char * string, ushort options);
```

B.1.2 Description

btparse is a C library for parsing and processing BibTeX files. It provides a lexical scanner and LR parser (constructed by PCCTS), both of which are efficient and offer good error detection and recovery; a set of functions for traversing the AST (abstract syntax tree) generated by the parser; and utility functions for manipulating strings according to BibTeX conventions. (Note that nothing in the library assumes that you're using BibTeX files for their original purpose of bibliographic data for scholarly publications; you could use the file format for any conceivable purpose that fits it. However, there is some code in the library that is really only appropriate for use with strings meant to be processed in the same way that BibTeX itself does. This is all entirely optional, though.)

Note that the interface provided by **btparse**, while complete, is fairly low-level. If you have more sophisticated needs, you might be interested my Text::BibTeX module for Perl 5 (available on CPAN).

B.1.3 Concepts and Terminology

To understand this document and use **btparse**, you should already be familiar with the BibTeX language—more specifically, the BibTeX data description language. (BibTeX being the complex beast that it is, one can conceive of the term applying to the program, the data language, the particular database structure described in the original BibTeX documentation, the ".bst" formatting language, and the set of conventions embodied in the standard styles included with the BibTeX distribution. In this document, I'll stick to the first two meanings—the data language because that's what **btparse** deals with, and the program because it's occasionally necessary to explain differences between my parser and BibTeX's.)

In particular, you should have a good idea what's going on in the following:

```
@string{and = { and },
        joe = "Blow, Joe",
        john = "John Smith"}

@book(ourbook,
      author = joe # and # john,
      title = {Our Little Book})
```

If this looks like something you want to parse, but don't want to have to write your own parser for, you've come to the right place.

Before going much further, though, you're going to have to learn some of the terminology I use for describing BibTeX data. Most of it's the same as you'll find in any BibTeX documentation, but it's important to be sure that we're talking about the same things here. So, some definitions:

top-level All text in a BibTeX file from the start of the file to the start of the first entry, and between entries thereafter.

name A string of letters, digits, and the following characters:

```
! $ & * + - . / : ; < > ? [ ] ^ _ ` |
```

A "name" is a catch-all used for entry types, entry keys, and field and macro names. For BibTeX compatibility, there are slightly different rules for these four entities; currently, the only such rule actually implemented is that field and macro names may not begin with a digit. Some names in the above example: `string`, `and`.

entry A chunk of text starting with an "at" sign (`@`) at top-level, followed by a name (the *entry type*), an *entry delimiter* (`{` or `(`), and proceeding to the matching closing delimiter. Also, the data structure that results from parsing this chunk of text. There are two entries in the above example.

entry type The name that comes right after an `@` at top-level. Examples from above: `string`, `book`.

entry metatype A classification of entry types that allows us to group one or more entry types under the same heading. With the standard BibTeX database structure, `article`, `book`, `inbook`, etc. all fall under the "regular entry" metatype. Other metatypes are "macro definition" (for `string` entries), "preamble" (for `preamble`) entries, and "comment" (`comment` entries). In fact, any entry whose type is not one of `string`, `preamble`, or `comment` is called a "regular" entry.

entry delimiters `{` and `}`, or `(` and `)`: the pair of characters that (almost) mark the boundaries of an entry. "Almost" because the start of an entry is marked by an `@`, not by the "entry open" delimiter.

entry key (Or just *key* when it's clear what we're speaking of.) The name immediately following the entry open delimiter in a regular entry, which uniquely identifies the entry. Example from above: `ourbook`. Only regular entries have keys.

field A name to the left of an equals sign in a regular or macro-definition entry. In the latter context, might also be called a macro name. Examples from above: `joe`, `author`.

field list In a regular entry, everything between the entry delimiters except for the entry key. In a macro definition entry, everything between the entry delimiters (possibly also called a macro list).

compound value (Usually just "value".) The text that follows an equals sign (=) in a regular or macro definition entry, up to a comma or the entry close delimiter; a list of one or more simple values joined by hash signs (`#`).

simple value A string, macro, or number.

string (Or, sometimes, "quoted string.") A chunk of text between quotes (`"`) or braces (`{` and `}`). Braces must balance: `{this is a {string}` is not a BibTeX string, but `{this is a {string}}` is. (`"this is a {string"` is also illegal, mainly to avoid the possibility of generating bogus TeX code–which BibTeX will do in certain cases.)

macro A name that appears on the right-hand side of an equals sign (i.e. as one simple value in a compound value). Implies that this name was defined as a macro in an earlier macro definition entry, but this is only checked if **btparse** is being asked to expand macros to their full definitions.

number An unquoted string of digits.

Working with **btparse** generally consists of passing the library some BibTeX data (or a source for some BibTeX data, such as a filename or a file pointer), which it then lexically scans, parses, and constructs an abstract syntax tree (AST) from. It returns this AST to you, and you call other **btparse** functions to traverse and query the tree.

The contents of AST nodes are the private domain of the library, and you shouldn't go poking into them. This being C, though, there's nothing to prevent you from doing so except good manners and the possibility that I might change the AST structure in future releases, breaking any badly-behaved code. Also, it's not necessary to know the structural relationships between nodes in the AST—that's taken care of by the query/traversal functions.

However, it's useful to know some of the things that **btparse** deposits in the AST and returns to you through those query/traversal functions. First off, each node has a "node type," which records the syntactic element corresponding to each node. For instance, the entry

```
@book{mybook, author = "Joe Blow", title = "My Little Book"}
```

is rooted by an "entry" node; under this would be found a "key" node (for the entry key), two "field" nodes (for the "author" and "title" fields); and associated with each field node would be a "string" node. The only time this concerns you is when you ask the library for a simple value; just looking at the text is not enough to distinguish quoted strings, numbers, and macro names, so **btparse** returns the nodetype as well.

In addition to the nodetype, **btparse** records the metatype of each "entry" node. This allows you (and the library) to distinguish, say, regular entries from comment entries. Not only do they have very different structures and must therefore be traversed differently by the library, but certain traversal functions make no sense on certain entry metatypes—thus it's necessary for you to be able to make the distinction as well.

That said, everything you need to know to work with the AST is explained in section B.4.

B.1.4 Data Types and Macros

btparse defines several types required for the external interface. First, it trivially defines a `boolean` type (along with `TRUE` and `FALSE` macros). This might affect you when including the btparse.h header in your own code—since it's not possible for the code to detect if there is already a `boolean` type defined, you might have to define the `HAVE_BOOLEAN` pre-processor token to deactivate btparse.h's `typedef` of `boolean`.

Next, two enumeration types are defined: `bt_metatype` and `bt_nodetype`. Both of these are used extensively in the library itself, and are made available to users of the library because they can be found in nodes of the `btparse` AST (abstract syntax tree). (I.e., querying the AST can give you `bt_metatype` and `bt_nodetype` values, so the `typedef`s must be available to your code.)

B.1.4.1 Entry metatype enum

`bt_metatype_t` has the following values:

- `BTE_UNKNOWN`
- `BTE_REGULAR`
- `BTE_COMMENT`
- `BTE_PREAMBLE`
- `BTE_MACRODEF`

which are determined by the "entry type" token. (`@string` entries have the `BTE_MACRODEF` metatype; `@comment` and `@preamble` correspond to `BTE_COMMENT` and `BTE_PREAMBLE`; and any other entry type has the `BTE_REGULAR` metatype.)

B.1.4.2 AST nodetype enum

`bt_nodetype` has the following values:

- `BTAST_UNKNOWN`
- `BTAST_ENTRY`
- `BTAST_KEY`
- `BTAST_FIELD`
- `BTAST_STRING`
- `BTAST_NUMBER`
- `BTAST_MACRO`

Of these, you'll only ever deal with the last three. They are returned when you query the AST for a simple value—just seeing the text isn't enough to distinguish between a quoted string, a number, and a macro, so the AST nodetype is supplied along with the text.

B.1.4.3 String processing option macros Since BibTeX is essentially a system for glueing strings together in a wide variety of ways, the processing done to its strings is fairly important. Most of the string transformations are done outside of the lexer/parser; this reduces their complexity, and makes it easier to switch different transformations on and off. This switching is done with an "options" bitmap which can be specified on a per-entry-metatype basis. (That is, you can have one set of transformations done to the strings in all regular entries, another set done to the strings in all macro definition entries, and so on.) If you need finer control than that, it's currently unavailable outside of the library (but it's just a matter of making a couple functions available and documenting them—so bug me if you need this feature).

There are three basic macros for constructing this bitmap:

BTO_CONVERT Convert "number" values to strings. (The conversion is trivial, involving changing the type of the AST node representing the number from `BTAST_NUMBER` to `BTAST_STRING`. "Number" values are stored as strings of digits, just as they are in the input data.)

BTO_EXPAND Expand macro invocations to the full macro text.

BTO_PASTE Paste simple values together.

BTO_COLLAPSE Collapse whitespace according to the BibTeX rules.

For instance, supplying `BTO_CONVERT | BTO_EXPAND` as the string options bitmap for the `BTE_REGULAR` metatype means that all simple values in "regular" entries will be converted to strings: numbers will simply have their "nodetype" changed, and macros will be expanded. Nothing else will be done to the simple values, though—they will not be concatenated, nor will whitespace be collapsed. See the `bt_set_stringopts()` and `bt_parse_*()` functions in section B.3 for more information on the various options for parsing; see section B.5 for details on the post-processing.

B.1.5 Using the Library

The following code is a skeletal example of using the **btparse** library:

```
#include <btparse.h>

int main (void)
{
   bt_initialize ();

   /* process some data */

   bt_cleanup ();
   exit (0);
}
```

Please note the call to `bt_initialize()`; this is very important! Without it, the library may crash or fail mysteriously. You *must* call `bt_initialize()` before calling any other **btparse** functions. `bt_cleanup()` just frees the memory allocated by `bt_initialize()`; if you are careful to call it before exiting, and `bt_free_ast()` on any abstract syntax trees generated by **btparse** when you are done with them, then your program shouldn't have any memory leaks. (Unless they're due to your own code, of course!)

B.2 The BIBTEX Data Language

B.2.1 Synopsis

```
# Lexical grammar, mode 1: top-level
AT                      \@
NEWLINE                 \n
COMMENT                 \%~[\n]*\n
WHITESPACE              [\ \r\t]+
JUNK                    ~[\@\n\ \r\t]+

# Lexical grammar, mode 2: in-entry
NEWLINE                 \n
COMMENT                 \%~[\n]*\n
WHITESPACE              [\ \r\t]+
NUMBER                  [0-9]+
NAME                    [a-z0-9\!\$\&\*\+\-\.\/\:\;\<\>\?\[\]\^\_\`\|]+
LBRACE                  \{
RBRACE                  \}
LPAREN                  \(
RPAREN                  \)
EQUALS                  =
HASH                    \#
COMMA                   ,
QUOTE                   \"

# Lexical grammar, mode 3: strings
# (very hairy -- see text)

# Syntactic grammar:
bibfile : ( entry )*

entry : AT NAME body

body : STRING                   # for comment entries
     | ENTRY_OPEN contents ENTRY_CLOSE

contents : ( NAME | NUMBER ) COMMA fields   # for regular entries
         | fields               # for macro definition entries
         | value                # for preamble entries

fields : field { COMMA fields }
       |

field : NAME EQUALS value

value : simple_value ( HASH simple_value )*

simple_value : STRING
             | NUMBER
             | NAME
```

B.2.2 Introduction

One of the problems with BibTeX is that there is no formal specification of the language. This means that users exploring the arcane corners of the language are largely on their own, and programmers implementing their own parsers are completely on their own—except for observing the behaviour of the original implementation.

Other parser implementors (Nelson Beebe of `bibclean` fame, in particular) have taken the trouble to explain the language accepted by their parser, and in that spirit the following is presented.

If you are unfamiliar with the arcana of regular and context-free languages, you will not have any easy time understanding this. This is *not* an introduction to the BibTeX language; any LaTeX book would be more suitable for learning the data language itself.

B.2.3 Lexical Grammar

The lexical scanner has three distinct modes: top-level, in-entry, and string. Roughly speaking, top-level is the initial mode; we enter in-entry mode on seeing an `@` at top-level; and on seeing the `}` or `)` that ends the entry, we return to top-level. We enter string mode on seeing a `"` or non-entry-delimiting `{` from in-entry mode. Note that the lexical language is both non-regular (because braces must balance) and context-sensitive (because `{` can mean different things depending on its syntactic context). That said, we will use regular expressions to describe the lexical elements, because they are the starting point used by the lexical scanner itself. The rest of the lexical grammar will be informally explained in the text.

From top-level, the following tokens are recognized according to the regular expressions on the right:

```
AT                     \@
NEWLINE                \n
COMMENT                \%~[\n]*\n
WHITESPACE             [\ \r\t]+
JUNK                   ~[\@\n\ \r\t]+
```

(Note that this is PCCTS regular expression syntax, which should be fairly familar to users of other regex engines. One oddity is that a character class is negated as `~[...]` rather than `[^...]`.)

On seeing `at` at top-level, we enter in-entry mode. Whitespace, junk, newlines, and comments are all skipped, with the latter two incrementing a line counter. (Junk is explicitly recognized to allow for `bibtex`'s "implicit comment" scheme.)

From in-entry mode, we recognize newline, comment, and whitespace identically to top-level mode. In addition, the following tokens are recognized:

```
NUMBER                 [0-9]+
NAME                   [a-z0-9\!\$\&\*\+\-\.\/\:\;\<\>\?\[\]\^\_\`\|]+
LBRACE                 \{
RBRACE                 \}
LPAREN                 \(
RPAREN                 \)
EQUALS                 =
HASH                   \#
COMMA                  ,
QUOTE                  \"
```

At this point, the lexical scanner starts to sound suspiciously like a context-free grammar, rather than a collection of independent regular expressions. However, it is necessary to keep this complexity in the scanner because certain characters (`{` and `(` in particular) have very different lexical meanings depending on the tokens that have preceded them in the input stream.

In particular, `{` and `(` are treated as "entry openers" if they follow one `at` and one `name` token, unless the value of the `name` token is `"comment"`. (Note the switch from top-level to in-entry between the two tokens.) In the `@comment` case, the delimiter is considered as starting a string, and we enter string mode. Otherwise, the delimiter is saved, and when we see a corresponding `}` or `)` it is considered an "entry closer". (Braces are balanced for free here because the string lexer takes care of counting brace-depth.)

Anywhere else, `{` is considered as starting a string, and we enter string mode. `"` always starts a string, regardless of context. The other tokens (`name`, `number`, `equals`, `hash`, and `comma`) are recognized unconditionally.

Note that `name` is a catch-all token used for entry types, citation keys, field names, and macro names; because BibTeX has slightly different (largely undocumented) rules for these various elements, a bit of trickery is needed to make things work. As a starting point, consider BibTeX's definition of what's allowed for an entry key: a sequence of any characters *except*

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

plus space. There are a couple of problems with this scheme. First, without specifying the character set from which those "magic 10" characters are drawn, it's a bit hard to know just what is allowed. Second, allowing `@` characters could lead to confusing BibTeX syntax (it doesn't confuse BibTeX, but it might confuse a human reader). Finally, allowing certain characters that are special to TeX means that BibTeX can generate bogus TeX code: try putting a backslash (\) or tilde (~) in a citation key. (This last exception is rather specific to the "generating (La)TeX code from a BibTeX database" application, but since that's the major application for BibTeX databases, then it will presumably be the major application for **btparse**, at least initially. Thus, it makes sense to pay attention to this problem.)

In **btparse**, then, a name is defined as any sequence of letters, digits, underscores, and the following characters:

```
! $ & * + - . / : ; < > ? [ ] ^ _ ` |
```

This list was derived by removing BibTeX's "magic 10" from the set of printable 7-bit ASCII characters (32-126), and then further removing `@`, \, and `~`. This means that **btparse** disallows some of the weirder entry keys that BibTeX would accept, such as `\foo@bar`, but still allows a string with initial digits. In fact, from the above definition it appears that **btparse** would accept a string of all digits as a "name;" this is not the case, though, as the lexical scanner recognizes such a digit string as a number first. There are two problems here: BibTeX entry keys may in fact be entirely numeric, and field names may not begin with a digit. (Those are two of the not-so-obvious differences in BibTeX's handling of keys and field names.) The tricks used to deal with these problems are implemented in the parser rather than the lexical scanner, so are described in section B.2.4 below.

The string lexer recognizes `lbrace`, `rbrace`, `lparen`, and `rparen` tokens in order to count brace- or parenthesis-depth. This is necessary so it knows when to accept a string delimited by braces or parentheses. (Note that a parenthesis-delimited string is only allowed after `@comment`—this is not a normal BibTeX construct.) In addition, it converts each non-space whitespace character (newline, carriage-return, and tab) to a single space. (Sequences of whitespace are not collapsed; that's the domain of string post-processing, which is well removed from the scanner or parser.) Finally, it accepts `"` to delimit quote-delimited strings. Apart from those restrictions, the string lexer accepts anything up to the end-of-string delimiter.

B.2.4 Syntactic Grammar

(The language used to describe the grammar here is the extended Backus-Naur Form (EBNF) used by PCCTS. Terminals are represented by uppercase strings, non-terminals by lowercase strings; terminal names are the same as those given in the lexical grammar above. `( foo )*` means zero or more repetitions of the `foo` production, and `{ foo }` means an optional `foo`.)

A file is just a sequence of zero or more entries:

```
bibfile : ( entry )*
```

An entry is an at-sign, a name (the "entry type"), and the entry body:

```
entry : AT NAME body
```

A body is either a string (this alternative is only tried if the entry type is `"comment"`) or the entry contents:

```
body : STRING                    # for comment entries
     | ENTRY_OPEN contents ENTRY_CLOSE
```

(`ENTRY_OPEN` and `ENTRY_CLOSE` are either `{` and `}` or `(` and `)`, depending what is seen in the input for a particular entry.)

There are three possible productions for the "contents" non-terminal. Only one applies to any given entry, depending on the entry metatype (which in turn depends on the entry type). Currently, **btparse** supports four entry metatypes: comment, preamble, macro definition, and regular. The first two correspond to `@comment` and `@preamble` entries; "macro definition" is for `@string` entries; and "regular" is for all other entry types. (The library will be extended to handle `@modify` and `@alias` entry types, and corresponding "modify" and "alias" metatypes, when BibTeX 1.0 is released and the exact syntax is known.) The "metatype" concept is necessary so that all entry types that aren't specifically recognized fall into the "regular" metatype. It's also convenient not to have to `strcmp` the entry type all the time.

```
contents : ( NAME | NUMBER ) COMMA fields     # for regular entries
         | fields                # for macro definition entries
         | value                 # for preamble entries
```

Note that the entry key is not just a `NAME`, but `( NAME | NUMBER)`. This is necessary because BibTeX allows all-numeric entry keys, but **btparse**'s lexical scanner recognizes such digit strings as `NUMBER` tokens.

`fields` is a comma-separated list of fields, with an optional single trailing comma:

```
fields : field { COMMA fields }
       |
```

A `field` is a single "field = value" assignment:

```
field : NAME EQUALS value
```

Note that NAME here is a restricted version of the "name" token described in section B.2.3 above. Any "name" token will be accepted by the parser, but it is immediately checked to ensure that it doesn't begin with a digit; if so, an artificial syntax error is triggered. (This is for compatibility with BibTeX, which doesn't allow field names to start with a digit.)

A value is a series of simple values joined by '#' characters:

```
value : simple_value ( HASH simple_value )*
```

A simple value is a string, number, or name (for macro invocations):

```
simple_value : STRING
             | NUMBER
             | NAME
```

B.3 Input Routines/Parser Interface

B.3.1 Synopsis

```
void  bt_set_stringopts (bt_metatype_t metatype, ushort options);
AST * bt_parse_entry_s (char *    entry_text,
                        char *    filename,
                        int       line,
                        ushort    options,
                        boolean * status);
AST * bt_parse_entry   (FILE *    infile,
                        char *    filename,
                        ushort    options,
                        boolean * status);
AST * bt_parse_file    (char *    filename,
                        ushort    options,
                        boolean * overall_status);
```

B.3.2 Description

The functions described here are used to read and parse BibTeX data, converting it from raw text to abstract-syntax trees (ASTs).

bt_set_stringopts ()

```
void bt_set_stringopts (bt_metatype_t metatype, ushort options);
```

Set the string-processing options for a particular entry metatype. This affects the entry post-processing done by bt_parse_entry_s(), bt_parse_entry(), and bt_parse_file(). If bt_set_stringopts() is never called, the four metatypes default to the following sets of string options:

```
BTE_REGULAR    BTO_CONVERT | BTO_EXPAND | BTO_PASTE | BTO_COLLAPSE
BTE_COMMENT    0
BTE_PREAMBLE   0
BTE_MACRODEF   BTO_CONVERT | BTO_EXPAND | BTO_PASTE
```

For example,

```
bt_set_stringopts (BTE_COMMENT, BTO_COLLAPSE);
```

will cause the library to collapse whitespace in the value from all comment entries; the AST returned by one of the `bt_parse_*` functions will reflect this change.

bt_parse_entry ()

```
AST * bt_parse_entry (FILE *    infile,
                      char *    filename,
                      ushort    options,
                      boolean * status);
```

Scans and parses the next BibTeX entry in `infile`. You should supply `filename` to help **btparse** generate accurate error messages; the library keeps track of `infile`'s current line number internally, so you don't need to pass that in. `options` should be a bitmap of non-string-processing options (currently, `BTO_NOSTORE` to disable storing macro expansions is the only such option). `*status` will be set to `TRUE` if the entry parsed successfully or with only minor warnings, and `FALSE` if there were any serious lexical or syntactic errors. If `status` is `NULL`, then the parse status will be unavailable to you. Both minor warnings and serious errors are reported on `stderr`.

Returns a pointer to the abstract-syntax tree (AST) describing the entry just parsed, or `NULL` if no more entries were found in `infile` (this will leave `infile` at end-of-file). Do not attempt to second guess `bt_parse_entry()` by detecting end-of-file yourself; it must be allowed to determine this on its own so it can clean up some static data that is preserved between calls on the same file.

`bt_parse_entry()` has two important restrictions that you should know about. First, you should let **btparse** manage all the input on the file; this is for reasons both superficial (so the library knows the current line number in order to generate accurate error messages) and fundamental (the library must be allowed to detect end-of-file in order to cleanup certain static variables and allow you to parse another file). Second, you cannot interleave the parsing of two different files; attempting to do so will result in a fatal error that will crash your program. This is a direct result of the static state maintained between calls of `bt_parse_entry()`.

Because of two distinct "failures" possible for `bt_parse_entry()` (end-of-file, which is expected but means to stop processing the current file; and error-in-input, which is not expected but allows you to continue processing the same file), you should usually call it like this:

```
while (entry = bt_parse_entry (file, filename, options, &ok))
{
   if (ok)
   {
      /* ... process entry ... */
   }
}
```

At the end of this loop, `feof (file)` will be true.

bt_parse_entry_s ()

```
AST * bt_parse_entry_s (char *    entry_text,
                        char *    filename,
                        int       line,
                        ushort    options,
                        boolean * status)
```

Scans and parses a single complete BibTeX entry contained in a string, `entry_text`. If you read this string from a file, you should help **btparse** generate accurate error messages by supplying the name of the file as `filename` and the line number of the beginning of the entry as `line`; otherwise, set `filename` to `NULL` and `line` to `1`. `options` and `status` are the same as for `bt_parse_entry()`.

Returns a pointer to the abstract-syntax tree (AST) describing the entry just parsed, and `NULL` if no entries were found in `entry_text` or if `entry_text` was `NULL`.

You should call `bt_parse_entry_s()` once more than the total number of entries you wish to parse; on the final call, set `entry_text` to `NULL` so the function knows there's no more text to parse. This final call allows it to clean up some structures allocated on the first call. Thus, `bt_parse_entry_s()` is usually used like this:

```
char *  entry_text;
ushort  options = 0;
boolean ok;
AST *   entry_ast;

while (entry_text = get_more_text ())
{
   entry_ast = bt_parse_entry_s (entry_text, NULL, 1, options, &ok);
   if (ok)
   {
      /* ... process entry ... */
   }
}

bt_parse_entry_s (NULL, NULL, 1, options, NULL);    /* cleanup */
```

assuming that `get_more_text()` returns a pointer to the text of an entry to parse, or `NULL` if there's no more text available.

bt_parse_file ()

```
AST * bt_parse_file (char *    filename,
                     ushort    options,
                     boolean * status)
```

Scans and parses an entire BibTeX file. If `filename` is `NULL` or `"-"`, then `stdin` will be read; otherwise, attempts to open the named file. If this attempt fails, prints an error message to `stderr` and returns `NULL`. `options` and `status` are the same as for `bt_parse_entry()`—note that `*status` will be `FALSE` if there were *any* errors in the entire file; for finer granularity of error-checking, you should use `bt_parse_entry()`.

Returns a pointer to a linked list of ASTs representing the entries in the file, or `NULL` if no entries were found in the file. This list can be traversed with `bt_next_entry()`, and the individual entries then traversed as usual (see section B.4).

B.4 AST Traversal/Query

B.4.1 Synopsis

```
AST * bt_next_entry (AST * entry_list,
                     AST * prev_entry)
AST * bt_next_field  (AST * entry, AST * prev, char ** name)
AST * bt_next_value  (AST * head,
                      AST * prev,
                      bt_nodetype_t * nodetype,
                      char ** text)

bt_metatype_t bt_entry_metatype (AST * entry)
char * bt_entry_type (AST * entry)
char * bt_entry_key   (AST * entry)
char * bt_get_text   (AST * node)
```

B.4.2 Description

The functions described here are all used to traverse and query the abstract-syntax trees (ASTs) returned by the input functions described in section B.3. The three "bt_next" functions (`bt_next_entry()`, `bt_next_field()`, and `bt_next_value()`) are used respectively to traverse a list of entries, the list of fields within a particular entry, and the list of simple values associated with a particular field. The other functions are just used to query various nodes in the tree for the useful information contained in them.

B.4.2.1 Traversal functions

bt_next_entry()

```
AST * bt_next_entry (AST * entry_list,
                     AST * prev_entry)
```

Used to traverse the linked list of entries returned by `bt_parse_file()` (see section B.3). On the first call, you should supply `NULL` for `prev_entry`, and a pointer to the head of the list will be returned. On subsequent calls, pass the previous return value as `prev_entry`; the function returns the next entry in the list, or `NULL` if there are no more entries. Also returns `NULL` if either `entry_list` or `prev_entry` are improper.

For example (ignoring error handling and variable declarations):

```
entries = bt_parse_file (filename, options, &status);
entry = NULL;
while (entry = bt_next_entry (entries, entry))
{
   /* process entry */
}
```

bt_next_field()

```
AST * bt_next_field  (AST * entry, AST * prev, char ** name)
```

Used to traverse the list of fields in a regular or macro definition entry. (You should call `bt_entry_metatype()` to determine if you have the right kind of entry before calling `bt_next_field()`.) `entry` should be a pointer to the AST for a single entry, as returned by `bt_parse_entry()`, `bt_parse_entry_s()`, or `bt_next_entry()`. On the first call, supply `NULL` for `prev`; `bt_next_field()` will return a pointer to the first field in `entry`, or `NULL` if `entry` has no fields (for instance, if it's a comment or preamble entry). On subsequent calls, pass the previous return value as `prev`; `bt_next_field()` will keep returning pointers to field sub-ASTs as long as it makes sense. These pointers can then be passed to `bt_next_value()` or `bt_get_text()` to get the field's value.

For example, the loop body in the previous example could be:

```
field = NULL;
while (field = bt_next_field (entry, field))
{
   /* process field */
}
```

bt_next_value()

```
AST * bt_next_value (AST * head,
                     AST * prev,
                     bt_nodetype_t * nodetype,
                     char ** text)
```

Traverses the list of simple values that make up the value of a single field. (Recall that a simple value is either a quoted string, a macro invocation, or a number. A compound value is a list of these separated by `'#'` in the original input. Depending on the string post-processing options used when the data was parsed, the "list of simple values" nature of the original data may be preserved in the AST that you're traversing, in which case you'll need a `bt_next_value()` loop.

`bt_next_value()` works much like `bt_next_entry()` and `bt_next_field()`: on the first call, you supply `NULL` for `prev`, and on subsequent calls you supply the previous return value. Returns `NULL` when there are no more simple values to return. Also sets `*nodetype` and `*text` to the corresponding information from the simple value node. `*nodetype` will be one of `BTAST_STRING`, `BTAST_MACRO`, or `BTAST_NUMBER`; `*text` will point to the same string as the AST node does (it is not copied for you), so don't mess with it.

For example, the loop body in the `bt_next_field()` example could be replaced with:

```
value = NULL;
while (value = bt_next_field (field, value, &nodetype, &text))
{
   switch (nodetype)
   {
      case BTAST_STRING:    /* process the string */
      case BTAST_MACRO:     /* process the macro */
      case BTAST_NUMBER:    /* process the number */
   }
}
```

See also the bt_get_text item.

B.4.2.2 Query functions

bt_entry_metatype()

```
bt_metatype_t bt_entry_metatype (AST * entry)
```

Returns the metatype of an entry. (Recall that the *metatype* is an enumerated type whose values are derived from the specific type of an entry; for instance, an `@comment` entry has type `"comment"` and metatype `BTE_COMMENT`. The type-metatype relationship is similarly obvious for `BTE_PREAMBLE`; `BTE_MACRODEF` corresponds to `@string` entries; and `BTE_REGULAR` corresponds to any other type.)

Returns `BTE_UNKNOWN` if `entry` is invalid (i.e., `NULL` or not a pointer to an entry AST).

bt_entry_type()

```
char * bt_entry_type (AST * entry)
```

Returns the type of an entry. Recall that the type is the name that appears after the `'@'` character in the original input. Returns `NULL` if `entry` is invalid (i.e., `NULL` or not a pointer to an entry AST).

bt_entry_key()

```
char * bt_entry_key (AST * entry)
```

Returns the citation key of a regular entry. (The citation key is the name that appears after the entry-open delimiter in a regular entry.) Returns `NULL` if `entry` is invalid (i.e., `NULL` or not a pointer to the AST for a regular entry).

bt_get_text()

```
char * bt_get_text (AST * node)
```

Performs all string post-processing (macro expansion, concatenation of simple values, and whitespace collapsing) of a compound value and returns the string that results. Can be called either on a field for a regular or macro definition entry (as returned by `bt_next_field()`), or on a comment or preamble entry. Returns `NULL` if called on an invalid AST node.

B.5 Entry and String Post-Processing

B.5.1 Synopsis

```
void bt_postprocess_string (char * s,
                            ushort options)

char * bt_postprocess_value (AST *   value,
                             ushort  options,
                             boolean replace);

char * bt_postprocess_field (AST *   field,
                             ushort  options,
                             boolean replace);

void bt_postprocess_entry (AST *  entry,
                           ushort options);
```

B.5.2 Description

When **btparse** parses a BibTeX entry, it initially stores the results in an abstract syntax tree (AST), in a form exactly mirroring the parsed data. For example, the entry

```
@Article{Jones:1997a,
  AuThOr = "Bob   Jones" # and # "Jim Smith ",
  TITLE = "Feeding Habits of
           the Common Cockroach",
  JoUrNaL = j_ent,
  YEAR = 1997
}
```

would parse to an AST that could be represented as follows:

```
(entry,"Article")
  (key,"Jones:1997a")
  (field,"AuThOr")
    (string,"Bob   Jones")
    (macro,"and")
    (string,"Jim Smith ")
  (field,"TITLE")
    (string,"Feeding Habits of               the Common Cockroach")
  (field,"JoUrNaL")
    (macro,"j_ent")
  (field,"YEAR")
    (number,"1997")
```

The advantage of this form is that all the important information in the entry is readily available by traversing the tree using the functions described in section B.4. This obvious problem is that the data is a little too raw to be immediately useful: entry types and field names are inconsistently capitalized, strings are full of unwanted whitespace, field values not reduced to single strings, and so forth.

All of these problems are addressed by **btparse**'s post-processing functions, described here. Normally, you won't have to call these functions—the library does the Right Thing for you after parsing each entry, and you can customize what exactly the Right Thing is for your application. (For instance, you can tell it to expand macros, but not to concatenate substrings together.) However, it's conceivable that you might wish to move the post-processing into your own code and out of the library's control. More likely, you could have strings that come from something other than BibTeX files that you would like to have treated as BibTeX strings; for that situation, the post-processing functions are essential. Finally, you might just be curious about what exactly happens to your data after it's parsed. If so, you've come to the right place for excruciatingly detailed explanations.

B.5.3 Functions

btparse offers four points of entry to its post-processing code. Of these, probably only the first and last—for processing individual strings and whole entries—will be commonly used.

B.5.3.1 Post-processing entry points To understand why four entry points are offered, an explanation of the sample AST shown above will help. First of all, the whole entry is represented by the `(entry, "Article")` node; this node has the entry key and all its field/value pairs as children. Entry nodes are returned by `bt_parse_entry()` and `bt_parse_entry_s()` (see section B.3) as well as `bt_next_entry()` (which traverses a list of entries returned from `bt_parse_file()`—see section B.4). Whole entries may be post-processed with `bt_postprocess_entry()`.

You may also need to post-process a single field, or just the value associated with it. (The difference is that processing the field can change the field name—e.g. to lowercase—in addition to the field value.) The `(field,"AuThOr")` node above is an example of a field sub-AST, and `(string,"Bob   Jones")` is the first node in the list of simple values representing that field's value. (Recall that a field value is, in general, a list of simple values.) Field nodes are returned by `bt_next_field()`, value nodes by `bt_next_value()`. The former may be passed to `bt_postprocess_field()` for post-processing, the latter to `bt_postprocess_value()`.

Finally, individual strings may wander into your program from many places other than a **btparse** AST. For that reason, `bt_postprocess_string()` is available for post-processing arbitrary strings.

B.5.3.2 Post-processing options All of the post-processing routines have an `options` parameter, which you can use to fine-tune the post-processing. (This is just like the per-metatype string-processing options that you can set before parsing entries; see `bt_set_stringopts()` in section B.3.) Like elsewhere in the library, `options` is a bitmap constructed by or'ing together various predefined constants. These constants and their effects are documented in section B.1.4.3.

bt_postprocess_string ()

```
void bt_postprocess_string (char * s,
                            ushort options)
```

Post-processes an individual string, `s`, which is modified in place. The only post-processing option that makes sense on individual strings is whether to collapse whitespace according to the BibTeX rules; thus, if `options & BTO_COLLAPSE` is false, this function has no effect. (Although it makes a complete pass over the string anyways. This is for future expansion.)

The exact rules for collapsing whitespace are simple: non-space whitespace characters (tabs and newlines mainly) are converted to space, any strings of more than one space within are collapsed to a single space, and any leading or trailing spaces are deleted. (Ensuring that all whitespace is spaces is actually done by **btparse**'s lexical scanner, so strings in **btparse** ASTs will never have whitespace apart from space. Likewise, any strings passed to bt_postprocess_string() should not contain non-space whitespace characters.)

bt_postprocess_value ()

```
char * bt_postprocess_value (AST *   value,
                             ushort  options,
                             boolean replace);
```

Post-processes a single field value, which is the head of a list of simple values as returned by `bt_next_value()`. All of the relevant string-processing options come into play here: conversion of numbers to strings (`BTO_CONVERT`), macro expansion (`BTO_EXPAND`), collapsing of whitespace (`BTO_COLLAPSE`), and string pasting (`BTO_PASTE`). Since pasting substrings together without first expanding macros and converting numbers would be nonsensical, attempting to do so is a fatal error.

If `replace` is true, then the list headed by `value` will be replaced by a list representing the processed value. That is, if string pasting is turned on (`options & BTO_PASTE` is true), then this list will be collapsed to a single node containing the single string that results from pasting together all the substrings. If string pasting is not on, then each node in the list will be left intact, but will have its text replaced by processed text.

If `replace` is false, then a new string will be built on the fly and returned by the function. Note that if pasting is not on in this case, you will only get the last string in the list. (It doesn't really make a lot of sense to post-process a value without pasting unless you're replacing it with the new value, though.)

Returns the string that resulted from processing the whole value, which only makes sense if pasting was on or there was only one value in the list. If a multiple-value list was processed without pasting, the last string in the list is returned (after processing).

Consider what might be done to the value of the `author` field in the above example, which is the concatenation of a string, a macro, and another string. Assume that the macro `and` expands to `" and "`, and that the variable `value` points to the sub-AST for this value. The original sub-AST corresponding to this value is

```
(string,"Bob   Jones")
(macro,"and")
(string,"Jim Smith ")
```

To fully process this value in-place, you would call

```
bt_postprocess_value (value, BTO_FULL, TRUE);
```

This would convert the value to a single-element list,

```
(string,"Bob Jones and Jim Smith")
```

and return the fully-processed string `"Bob Jones and Jim Smith"`. Note that the `and` macro has been expanded, interpolated between the two literal strings, everything pasted together, and finally whitespace collapsed. (Collapsing whitespace before concatenating the strings would be a bad idea.)

(Incidentally, `BTO_FULL` is just a macro for the combination of all possible string-processing options, currently:

```
BTO_CONVERT | BTO_EXPAND | BTO_PASTE | BTO_COLLAPSE
```

There are two other similar shortcut macros: `BTO_MACRO` to express the special string-processing done on macro values, which is the same as `BTO_FULL` except for the absence of `BTO_COLLAPSE`; and `BTO_MINIMAL`, which means no string-processing is to be done.)

Let's say you'd rather preserve the list nature of the value, while expanding macros and converting any numbers to strings. (This conversion is trivial: it just changes the type of the node from `BTAST_NUMBER` to `BTAST_STRING`. "Number" values are always stored as a string of digits, just as they appear in the file.) This would be done with the call

```
bt_postprocess_value
   (value, BTO_CONVERT|BTO_EXPAND|BTO_COLLAPSE,TRUE);
```

which would change the list to

```
(string,"Bob Jones")
(string,"and")
(string,"Jim Smith")
```

Note that whitespace is collapsed here *before* any concatenation can be done; this is probably a bad idea. But you can do it if you wish. (If you get any ideas about cooking up your own value post-processing scheme by doing it in little steps like this, take a look at the source to `bt_postprocess_value()`; it should dissuade you from such a venture.)

bt_postprocess_field ()

```
char * bt_postprocess_field (AST *   field,
                             ushort  options,
                             boolean replace);
```

This is little more than a front-end to `bt_postprocess_value()`; the only difference is that you pass it a "field" AST node (eg. the `(field,"AuThOr")` in the above example), and that it transforms the field name in addition to its value. In particular, the field name is forced to lowercase; this behaviour is (currently) not optional.

Returns the string returned by `bt_postprocess_value()`.

bt_postprocess_entry ()

```
void bt_postprocess_entry (AST *  entry,
                           ushort options);
```

Post-processes all values in an entry. If `entry` points to the AST for a "regular" or "macro definition" entry, then the values are just what you'd expect: everything on the right-hand side of a field or macro "assignment." You can also post-process comment and preamble entries, though. Comment entries are essentially one big string, so only whitespace collapsing makes sense on them. Preambles may have multiple strings pasted together, so all the string-processing options apply to them. (And there's nothing to prevent you from using macros in a preamble.)

B.6 Macro Table

B.6.1 Synopsis

```
void bt_add_macro_value (AST *  assignment,
                         ushort options);
void bt_add_macro_text (char * macro,
                        char * text,
                        char * filename,
                        int    line);
```

```
void bt_delete_macro (char * macro);
void bt_delete_all_macros (void);

int bt_macro_length (char *macro);
char * bt_macro_text (char * macro,
                      char * filename,
                      int line);
```

B.6.2 Description

btparse maintains a single table of all macros (abbreviations) encountered while parsing BibTeX entries. It updates this table whenever it encounters a "macro definition" (`@string`) entry, and refers to it whenever a macro is used in an entry and needs to be expanded. (Macros are not necessarily expanded on input, although this is the default. See section B.5.) Macro definitions are only cleared when **btparse**'s global cleanup function, `bt_cleanup()`, is called. Thus, unless you explicitly call `bt_delete_macro()` or `bt_delete_all_macros()`, macro definitions persist for as long as you use the library—usually, the lifetime of your process.

B.6.3 Functions

You can use the following functions to add macros, delete them, and query their values—thus interfering with **btparse**'s normal operation on the fly.

bt_add_macro_text ()

```
void bt_add_macro_text (char * macro,
                        char * text,
                        char * filename,
                        int    line);
```

Defines a new macro, or redefines an old one. `macro` is the name of the macro, and `text` is the text it should expand to. `filename` and `line` are just used to generate any warnings about the macro definition; if they don't apply, specify `NULL` for `filename` and `0` for `line`. The only such warning occurs when you redefine an old macro: its value is overridden, and `bt_add_macro_text()` issues a warning saying so.

For instance, when parsing this macro definition entry:

```
@string{fubar = "Fouled Up Beyond All Recognition"}
```

the library (in particular, the post-processing code called after an entry is successfully parsed) will ultimately do this:

```
bt_add_macro_text ("fubar", "Fouled Up Beyond All Recognition",
                   filename, line);
```

This in turn will cause the macro `fubar` to be expanded appropriately whenever the post-processing code sees it in any future entries.

bt_add_macro_value ()

```
void bt_add_macro_value (AST *  assignment,
                         ushort options);
```

This function is mainly for internal use by the library, but it's available to you if you ever find yourself with a little bit of AST representing a macro definition, and you want to set the macro yourself (rather than letting the library's post-processing code take care of it for you). `assignment` must be an AST node as returned by `bt_next_field()`. Unlike most other **btparse** functions that take an `options` argument, `options` here tells how the value in `assignment` was post-processed. This is needed because macro values have to be processed in a special way to be valid in future expansions; if this one wasn't processed like that, `bt_add_macro_value()` will do it for you. If you don't know how the value was post-processed, just supply `0` for `options`—that's guaranteed to describe something different from "the right way" for macros, so the post-processing will be done correctly.

The processing done to macro values is mainly to ensure that we can get away with storing just a string in the macro table: macros invoked by the macro are themselves expanded, and all sub-strings are concatenated. For instance, if **btparse** parses these entries:

```
@string{and = " and "}
@string{jim_n_bob = "James Smith" # and # "Bob Jones"}
```

then the value stored for `jim_n_bob` should obviously be the string `"James Smith and Bob Jones"`. To ensure this, **btparse** has to process the value of `and` differently from most BibTeX strings: in particular, whitespace is *not* collapsed before the string is stored. That way, the correct value, `" and "`, is interpolated into the value of `jim_n_bob`. Thus, all macro values have sub-macros expanded and strings concatenated before they are stored, but whitespace is not collapsed until the macro is used in a regular entry.

This function calls `bt_add_macro_text()`, so the same proviso about redefining old macros applies—a warning will be issued, and the old value lost.

bt_delete_macro ()

```
void bt_delete_macro (char * macro);
```

Deletes a macro from the macro table. If `macro` isn't defined, takes no action.

bt_delete_all_macros ()

```
void bt_delete_all_macros (void);
```

Deletes all macros from the macro table.

bt_macro_length ()

```
int bt_macro_length (char *macro);
```

Returns the length of a macro's expansion text. If the macro is undefined, returns 0; no warning is issued.

bt_macro_text ()

```
char * bt_macro_text (char * macro,
                      char * filename,
                      int line);
```

Returns the expansion text of a macro. If the macro is not defined, issues a warning and returns `NULL`. `filename` and `line` are used for generating this warning; if they don't apply (i.e. you're not expanding the macro as a result of finding it in some file), supply `NULL` for `filename` and `0` for `line`.

B.7 Name Splitting

B.7.1 Synopsis

```
bt_stringlist * bt_split_list (char *   string,
                               char *   delim,
                               char *   filename,
                               int      line,
                               char *   description);
void bt_free_list (bt_stringlist *list);
bt_name * bt_split_name (char *  name,
                         char *  filename,
                         int     line,
                         int     name_num);
void bt_free_name (bt_name * name);
```

B.7.2 Description

When BibTeX files are used for their original purpose—bibliographic entries describing scholarly publications—processing lists of names (authors and editors mostly) becomes important. Although such name-processing is outside the general-purpose database domain of most of the **btparse** library, these splitting functions are provided as a concession to reality: most BibTeX data files use the BibTeX conventions for author names, and a library to process that data ought to be capable of processing the names.

Name-processing comes in two stages: first, split up a list of names into individual strings; second, split up each name into "parts" (first, von, last, and jr). The first is actually quite general: you could pick a delimiter (such as `'and'`, used for lists of names) and use it to divide any string into substrings. `bt_split_list()` could then be called to break up the original string and extract the substrings. `bt_split_name()`, however, is quite specific to four-part author names written using BibTeX conventions. (These conventions are described informally in any BibTeX documentation; the description you will find here is more formal and algorithmic—and thus harder to understand.)

See section B.8 for information on turning split-up names back into strings in a variety of ways.

B.7.3 Functions

bt_split_list()

```
bt_stringlist * bt_split_list (char *   string,
                               char *   delim,
                               char *   filename,
                               int      line,
                               char *   description)
```

Splits `string` into substrings delimited by `delim` (a fixed string). The splitting is done according to the rules used by BibTeX for splitting up a list of names, in particular:

- delimiters at beginning or end of string are ignored
- delimiters must be surrounded by whitespace
- matching of delimiters is case insensitive
- delimiters at non-zero brace depth are ignored

For instance, if the delimiter is `"and"`, then the string

```
Candy and Apples AnD {Green Eggs and Ham}
```

splits into three substrings: `"Candy"`, `"Apples"`, and `"{Green Eggs and Ham}"`.

If there are extra delimiters at the extremities of the string—say, an `"and"` at the beginning of the string—then they are included in the first/last string; no warning is currently printed, but this may change. Successive delimiters (`"and and"`) result in a warning and a NULL string being added to the list of substrings. For instance, the string

```
and Joe Q. Blow and and Smith, Jr., John
```

would split into three substrings: `"and Joe Q. Blow"`, `NULL`, and `"Smith, Jr., John"`.

(If these rules seem somewhat odd, don't blame me: I just implemented BibTeX's observed behaviour and added warning messages for one of the more obvious and easily-detected mistakes.)

The substrings are returned as a `bt_stringlist` structure:

```
typedef struct
{
   char *  string;
   int     num_items;
   char ** items;
} bt_stringlist;
```

There is currently no elegant interface to this structure: you just have to poke around in it yourself. The fields are:

`string` a copy of the `string` parameter passed to `bt_split_list()`, but with NUL characters replacing the space after each substring. (This is safe because delimiters must be surrounded by whitespace, which means that each substring is followed by whitespace which is not part of the substring.) You probably shouldn't fiddle with `string`; it's just there so that `bt_free_list()` has something to `free()`.

num_items the number of substrings found in the string passed to `bt_split_list()`.

items an array of `num_items` pointers into `string`. For instance, `items[1]` points to the second substring. Since `string` has been mangled with NUL characters, it is safe to treat `items[i]` as a regular C string.

`filename`, `line`, and `description` are all used for generating warning messages. `filename` and `line` simply describe where the string came from, and `description` is a brief (one word) description of the substrings. For instance, if you are splitting a list of names, supply `"name"` for `description`—that way, warnings will refer to "name X" rather than "substring x".

bt_free_list()

```
void bt_free_list (bt_stringlist *list)
```

Frees a `bt_stringlist` structure as returned by `bt_split_list()`. That is, it frees the copy of the string you passed to `bt_split_list()`, and then frees the structure itself.

bt_split_name()

```
bt_name * bt_split_name (char *   name,
                         char *   filename,
                         int      line,
                         int      name_num)
```

Splits a single BibTeX-style author name into four parts: first, von, last, and jr. This can handle almost all names in the style of the major Western European languages, but not quite. (Alas!)

A name is split by first dividing into tokens; tokens are separated by whitespace or commas at brace-level zero. Thus the name

```
van der Graaf, Horace Q.
```

has five tokens, whereas the name

```
{Foo, Bar, and Sons}
```

consists of a single token.

How tokens are divided into parts depends on the form of the name. If the name has no commas at brace-level zero (as in the second example), then it is assumed to be in either "first last" or "first von last" form. If there are no tokens that start with a lower-case letter, then "first last" form is assumed: the final token is the last name, and all other tokens form the first name. Otherwise, the earliest contiguous sequence of tokens with initial lower-case letters is taken as the 'von' part; if this sequence includes the final token, then a warning is printed and the final token is forced to be the 'last' part.

If a name has a single comma, then it is assumed to be in "von last, first" form. A leading sequence of tokens with initial lower-case letters, if any, forms the 'von' part; tokens between the 'von' and the comma form the 'last' part; tokens following the comma form the 'first' part. Again, if there are no token following a leading sequence of lowercase tokens, a warning is printed and the token immediately preceding the comma is taken to be the 'last' part.

If a name has more than two commas, a warning is printed and the name is treated as though only the first two commas were present.

Finally, if a name has two commas, it is assumed to be in "von last, jr, first" form. (This is the only way to represent a name with a 'jr' part.) The parsing of the name is the same as for a one-comma name, except that tokens between the two commas are taken to be the 'jr' part.

The one case not properly handled by BibTeX name conventions is a name with a 'jr' part not separated from the last name by a comma; for example:

```
Henry Ford Jr.
George Herbert Walker Bush III
```

Both of these would be incorrectly interpreted by both BibTeX and bt_split_name(): the `"Jr."` or `"III"` token would be taken as the last name, and the other tokekens as a two- or four-part first name. The workaround is to shoehorn the 'jr' into the last name:

```
Henry {Ford Jr.}
George Herbert Walker {Bush III}
```

but this will make it impossible to extract the last name on its own, e.g. to generate "author-year" style citations. This design flaw may be fixed in a future version of **btparse**.

The split-up name is returned as a `bt_name` structure:

```
typedef struct
{
   bt_stringlist * tokens;
   char ** parts[BT_MAX_NAMEPARTS];
   int     part_len[BT_MAX_NAMEPARTS];
} bt_name;
```

Again, there's no nice interface to this structure; you'll just have to access the fields individually. They are:

`tokens` the name, broken down into a flat list of tokens. See above for a description of the `bt_stringlist` structure.

`parts` an array of arrays of pointers into the token list. The major dimension of this beast is the "name part;" you should index this dimension using the `bt_namepart` enum. For instance, `parts[BTN_LAST]` is an array of pointers to the tokens comprising the last name; `parts[BTN_LAST][1]` is a `char *`: the second token of the 'last' part; and `parts[BTN_LAST][1][0]` is the first character of the second token of the 'last' part.

`part_len` the length, in tokens, of each part. For instance, you might loop over all tokens in the 'first' part as follows (assuming `name` is a `bt_name *` returned by `bt_split_name()`):

```
for (i = 0; i < name->part_len[BTN_FIRST]; i++)
{
   printf ("token %d of first name: %s\n",
           i, name->parts[BTN_FIRST][i]);
}
```

bt_free_name()

```
void bt_free_name (bt_name * name)
```

Frees the `bt_name` structure created by `bt_split_name()` (including the `bt_stringlist` structure inside the `bt_name`).

B.8 Name Formatting

B.8.1 Synopsis

```
bt_name_format * bt_create_name_format (char * parts,
                                        boolean abbrev_first);
void bt_free_name_format (bt_name_format * format);
void bt_set_format_text (bt_name_format * format,
                         bt_namepart part,
                         char * pre_part,
                         char * post_part,
                         char * pre_token,
                         char * post_token);
void bt_set_format_options (bt_name_format * format,
                            bt_namepart part,
                            boolean abbrev,
                            bt_joinmethod join_tokens,
                            bt_joinmethod join_part);
char * bt_format_name (bt_name * name, bt_name_format * format);
```

B.8.2 Description

After splitting a name into its components parts (represented as a `bt_name` structure), you often want to put it back together again as a single string in a consistent way. **btparse** provides a very flexible way to do this, generally in two stages: first, you create a "name format" which describes how to put the tokens and parts of any name back together, and then you apply the format to a particular name.

The "name format" is encapsulated in a `bt_name_format` structure, which is created with `bt_create_name_format()`. This function includes some clever trickery that means you can usually get away with calling it alone, and not need to do any customization of the format. If you do need to customize the format, though, `bt_set_format_text()` and `bt_set_format_options()` provide that capability.

The format controls the following:

- which name parts are printed, and in what order (e.g. "first von last jr", or "von last jr first")
- the text that precedes and follows each part (e.g. if the first name follows the last name, you probably want a comma before the 'first' part: "Smith, John" rather than "Smith John")

- the text that precedes and follows each token (e.g. if the first name is abbreviated, you may want a period after each token: “J. R. Smith” rather than “J R Smith”)
- the method used to join the tokens of each part together
- the method used to join each part to the following part

All of these except the list of parts to format are kept in arrays indexed by name part: for example, the structure has a field

```
char * post_token[BT_MAX_NAMEPARTS]
```

and `post_token[BTN_FIRST]` (`BTN_FIRST` is from the `bt_namepart enum`) is the string to be added after each token in the first name—for example, `"."` if the first name is to be abbreviated in the conventional way.

Yet another `enum`, `bt_joinmethod`, describes the available methods for joining tokens together. Note that there are *two* sets of join methods in a name format: between tokens within a single part, and between the tokens of two different parts. The first allows you, for example, to change `"J R Smith"` (first name abbreviated with no post-token text but tokens joined by a space) to `"JR Smith"` (the same, but first-name tokens jammed together). The second is mainly used to ensure that “von” and “last” name-parts may be joined with a tie: `"de~Roche"` rather than `"de Roche"`.

The token join methods are:

BTJ_MAYTIE Insert a “discretionary tie” between tokens. That is, either a space or a “tie” is inserted, depending on context. (A “tie,” otherwise known as unbreakable space, is currently hard-coded as `"~"`—from TeX.)

The format is then applied to a particular name by `bt_format_name()`, which returns a new string.

BTJ_SPACE Always insert a space between tokens.

BTJ_FORCETIE Always insert a “tie” (`"~"`) between tokens.

BTJ_NOTHING Insert nothing between tokens—just jam them together.

Tokens are joined together, and thus the choice of whether to insert a “discretionary tie” is made, at two places: within a part and between two parts. Naturally, this only applies when `BTJ_MAYTIE` was supplied as the token-join method; `BTJ_SPACE` and `BTJ_FORCETIE` always insert either a space or tie, and `BTJ_NOTHING` always adds nothing between tokens. Within a part, ties are added after a the first token if it is less than three characters long, and before the last token. Between parts, a tie is added only if the preceding part consisted of single token that was less than three characters long. In all other cases, spaces are inserted. (This implementation slavishly follows BibTeX.)

B.8.3 Functions

bt_create_name_format()

```
bt_name_format * bt_create_name_format (char * parts,
                                         boolean abbrev_first)
```

Creates a name format for a given set of parts, with variations for the most common forms of customization—the order of parts and whether to abbreviate the first name.

The `parts` parameter specifies which parts to include in a formatted name, as well as the order in which to format them. `parts` must be a string of four or fewer characters, each of which denotes one of the four name parts: for instance, `"vljf"` means to format all four parts in "von last jr first" order. No characters outside of the set `"fvlj"` are allowed, and no characters may be repeated. `abbrev_first` controls whether the 'first' part will be abbreviated (i.e., only the first letter from each token will be printed).

In addition to simply setting the list of parts to format and the "abbreviate" flag for the first name, `bt_create_name_format()` initializes the entire format structure so as to minimize the need for further customizations:

- The "token join method"—what to insert between tokens of the same part—is set to `BTJ_MAYTIE` (discretionary tie) for all parts
- The "part join method"—what to insert after the final token of a particular part, assuming there are more parts to come—is set to `BTJ_SPACE` for the 'first', 'last', and 'jr' parts. If the 'von' part is present and immediately precedes the 'last' part (which will almost always be the case), `BTJ_MAYTIE` is used to join 'von' to 'last'; otherwise, 'von' also gets `BTJ_SPACE` for the inter-part join method.
- The abbreviation flag is set to `FALSE` for the 'von', 'last', and 'jr' parts; for 'first', the abbreviation flag is set to whatever you pass in as `abbrev_first`.
- Initially, all "surrounding text" (pre-part, post-part, pre-token, and post-token) for all parts is set to the empty string. Then a few tweaks are done, depending on the `abbrev_first` flag and the order of tokens. First, if `abbrev_first` is `TRUE`, the post-token text for first name is set to `"."`—this changes `"J R Smith"` to `"J. R. Smith"`, which is usually the desired form. (If you *don't* want the periods, you'll have to set the post-token text yourself with `bt_set_format_text()`.)

  Then, if 'jr' is present and immediately after 'last' (almost always the case), the pre-part text for 'jr' is set to `", "`, and the inter-part join method for 'last' is set to `BTJ_NOTHING`. This changes `"John Smith Jr"` (where the space following `"Smith"` comes from formatting the last name with a `BTJ_SPACE` inter-part join method) to `"John Smith, Jr"` (where the `", "` is now associated with `"Jr"`—that way, if there is no 'jr' part, the `", "` will not be printed.)

  Finally, if 'first' is present and immediately follows either 'jr' or 'last' (which will usually be the case in "last-name first" formats), the same sort of trickery is applied: the pre-part text for 'first' is set to `", "`, and the part join method for the preceding part (either 'jr' or 'last') is set to `BTJ_NOTHING`.

While all these rules are rather complicated, they mean that you are usually freed from having to do any customization of the name format. Certainly this is the case if you only need `"fvlj"` and `"vljf"` part orders, only want to abbreviate the first name, want periods after abbreviated tokens, non-breaking spaces in the "right" places, and commas in the conventional places.

If you want something out of the ordinary—for instance, abbreviated tokens jammed together with no puncuation, or abbreviated last names—you'll need to customize the name format a bit with `bt_set_format_text()` and `bt_set_format_options()`.

bt_free_name_format()

```
void bt_free_name_format (bt_name_format * format)
```

Frees a name format created by `bt_create_name_format()`.

bt_set_format_text()

```
void bt_set_format_text (bt_name_format * format,
                         bt_namepart part,
                         char * pre_part,
                         char * post_part,
                         char * pre_token,
                         char * post_token)
```

Allows you to customize some or all of the surrounding text for a single name part. Supply `NULL` for any chunk of text that you don't want to change.

For instance, say you want a name format that will abbreviate first names, but without any punctuation after the abbreviated tokens. You could create and customize the format as follows:

```
format = bt_create_name_format ("fvlj", TRUE);
bt_set_format_text (format,
                    BTN_FIRST,       /* name-part to customize */
                    NULL, NULL,      /* pre- and post- part text */
                    NULL, "");       /* empty string for post-token */
```

Without the `bt_set_format_text()` call, `format` would result in names formatted like `"J. R. Smith"`. After setting the post-token text for first names to `""`, this name would become `"J R Smith"`.

bt_set_format_options()

```
void bt_set_format_options (bt_name_format * format,
                            bt_namepart part,
                            boolean abbrev,
                            bt_joinmethod join_tokens,
                            bt_joinmethod join_part)
```

Allows further customization of a name format: you can set the abbreviation flag and the two token-join methods. Alas, there is no mechanism for leaving a value unchanged; you must set everything with `bt_set_format_options()`.

For example, let's say that just dropping periods from abbreviated tokens in the first name isn't enough; you *really* want to save space by jamming the abbreviated tokens together: `"JR Smith"` rather than `"J R Smith"` Assuming the two calls in the above example have been done, the following will finish the job:

```
bt_set_format_options (format, BTN_FIRST,
                       TRUE,         /* keep same value for abbrev flag */
                       BTJ_NOTHING,  /* jam tokens together */
                       BTJ_SPACE);   /* space after final token of part */
```

Note that we unfortunately had to know (and supply) the current values for the abbreviation flag and post-part join method, even though we were only setting the intra-part join method.

bt_format_name()

```
char * bt_format_name (bt_name * name, bt_name_format * format)
```

Once a name format has been created and customized to your heart's content, you can use it to format any number of names that have been split with `bt_split_name` (see section B.7). Simply pass the name structure and name format structure, and a newly-allocated string containing the formatted name will be returned to you. It is your responsibility to `free()` this string.

B.9 Miscellaneous String Utilities

B.9.1 Synopsis

```
void bt_purify_string (char * string, ushort options);
void bt_change_case (char transform, char * string, ushort options);
```

B.9.2 Description

bt_purify_string()

```
void bt_purify_string (char * string, ushort options);
```

"Purifies" a `string` in the BibTeX way (usually used for generating sort keys). `string` is modified in-place. `options` is currently unused; just set it to zero for future compatibility. Purification consists of copying alphanumeric characters, converting hyphens and ties to space, copying spaces, and skipping (almost) everything else.

"Almost" because "special characters" (used for accented and non-English letters) are handled specially. Recall that a BibTeX special character is any brace-group that starts at brace-depth zero whose first character is a backslash. For instance, the string

```
{\foo bar}Herr M\"uller went from {P{\r r}erov} to {\AA}rhus
```

contains two special characters: `"{\foo bar}"` and `"\AA"`. Neither the `\"u` nor the `\r r` are special characters, because they are not at the right brace depth.

Special characters are handled as follows: if the control sequence (the TeX command that follows the backslash) is recognized as one of LaTeX's "foreign letters" (`\oe`, `\ae`, `\o`, `\l`, `\ae`, `\ss`, plus uppercase versions), then it is converted to a reasonable English approximation by stripping the backslash and converting the second character (if any) to lowercase; thus, `{\AA}` in the above example would become simply `Aa`. All other control sequences in a special character are stripped, as are all non-alphabetic characters.

For example the above string, after "purification," becomes

```
barHerr Muller went from Pr rerov to Aarhus
```

Obviously, something has gone wrong with the word `P{\r r}erov` (a town in the Czech Republic). The accented 'r' should be a special character, starting at brace-depth zero. If the original string were instead

```
{\foo bar}Herr M\"uller went from P{\r r}erov to {\AA}rhus
```

then the purified result would be more sensible:

```
barHerr Muller went from Prerov to Aarhus
```

Note the use of a "nonsense" special character `{\foo bar}`: this trick is often used to put certain text in a string solely for generating sort keys; the text is then ignored when the document is processed by TeX (as long as `\foo` is defined as a no-op TeX macro). This assumes, of course, that the output is eventually processed by TeX; if not, then this trick will backfire on you.

Also, `bt_purify_string()` is adequate for generating sort keys when you want to sort according to English-language conventions. To follow the conventions of other languages, though, a more sophisticated approach will be needed; hopefully, future versions of **btparse** will address this deficiency.

bt_change_case()

```
void bt_change_case (char transform, char * string, ushort options);
```

Converts a string to lowercase, uppercase, or "non-book title capitalization", with special attention paid to BibTeX special characters and other brace-groups. The form of conversion is selected by the single character `transform`: `'u'` to convert to uppercase, `'l'` for lowercase, and `'t'` for "title capitalization". `string` is modified in-place, and `options` is currently unused; set it to zero for future compatibility.

Lowercase and uppercase conversion are obvious, with the proviso that text in braces is treated differently (explained below). Title capitalization simply means that everything is converted to lowercase, except the first letter of the first word, and words immediately following a colon or sentence-ending punctuation. For instance,

```
Flying Squirrels: Their Peculiar Habits. Part One
```

would be converted to

```
Flying squirrels: Their peculiar habits. Part one
```

Text within braces is handled as follows. First, in a "special character" (see above for definition), control sequences that constitute one of LaTeX's non-English letters are converted appropriately—e.g., when converting to lowercase, `\AE` becomes `\ae`). Any other control sequence in a special character (including accents) is preserved, and all text in a special character, regardless of depth and punctuation, is converted to lowercase or uppercase. (For "title capitalization," all text in a special character is converted to lowercase.)

Brace groups that are not special characters are left completely untouched: neither text nor control sequences within non-special character braces are touched.

For example, the string

```
A Guide to \LaTeXe: Document Preparation ...
```

would, when `transform` is `'t'` (title capitalization), be converted to

```
A guide to \latexe: Document preparation ...
```

which is probably not the desired result. A better attempt is

```
A Guide to {\LaTeXe}: Document Preparation ...
```

which becomes

```
A guide to {\LaTeXe}: Document preparation ...
```

However, if you go back and re-read the description of `bt_purify_string()`, you'll discover that `{\LaTeXe}` here is a special character, but not a non-English letter: thus, the control sequence is stripped. Thus, a sort key generated from this title would be

```
A Guide to  Document Preparation
```

...oops! The right solution (and this applies to any title with a TeX command that becomes actual text) is to bury the control sequence at brace-depth two:

```
A Guide to {{\LaTeXe}}: Document Preparation ...
```

Appendix C Perl Library Documentation

C.1 Text::BibTeX

C.1.1 Synopsis

```
use Text::BibTeX;

$bibfile = new Text::BibTeX::File "foo.bib";
$newfile = new Text::BibTeX::File ">newfoo.bib";

while ($entry = new Text::BibTeX::Entry $bibfile)
{
   next unless $entry->parse_ok;

        .             # hack on $entry contents, using various
        .             # Text::BibTeX::Entry methods
        .

   $entry->write ($newfile);
}
```

C.1.2 Description

The Text::BibTeX module serves mainly as a high-level introduction to the Text::BibTeX library, for both code and documentation purposes. The code loads the two fundamental modules for processing BibTeX files (Text::BibTeX::File and Text::BibTeX::Entry), and this documentation gives a broad overview of the whole library that isn't available in the documentation for the individual modules that comprise it.

In addition, the Text::BibTeX module provides a number of miscellaneous functions that are useful in processing BibTeX data (especially the kind that comes from bibliographies as defined by BibTeX 0.99, rather than generic database files). These functions don't generally fit in the object-oriented class hierarchy centred around the Text::BibTeX::Entry class, mainly because they are specific to bibliographic data and operate on generic strings (rather than being tied to a particular BibTeX entry). These are also documented here, in section C.1.7.

Note that every module described here begins with the Text::BibTeX prefix. For brevity, I have dropped this prefix from most class and module names in the rest of this manual page (and in most of the other manual pages in the library).

C.1.3 Modules and Classes

The Text::BibTeX library includes a number of modules, many of which provide classes. Usually, the relationship is simple and obvious: a module provides a class of the same name—for instance, the Text::BibTeX::Entry module provides the Text::BibTeX::Entry class. There are a few exceptions, though: most obviously, the Text::BibTeX module doesn't provide any classes itself, it merely loads two modules (Text::BibTeX::Entry and Text::BibTeX::File) that do. The other exceptions are mentioned in the descriptions below, and discussed in detail in the documentation for the respective modules.

The modules are presented roughly in order of increasing specialization: the first three are essential for any program that processes BibTeX data files, regardless of what kind of data they hold. The later modules are specialized for use with bibliographic databases, and serve both to emulate BibTeX 0.99's standard styles and to provide an example of how to define a database structure through such specialized modules. Each module is fully documented in its respective manual page.

Text::BibTeX Loads the two fundamental modules (Entry and File), and provides a number of miscellaneous functions that don't fit anywhere in the class hierarchy.

Text::BibTeX::File Provides an object-oriented interface to BibTeX database files. In addition to the obvious attributes of filename and filehandle, the "file" abstraction manages properties such as the database structure and options for it.

Text::BibTeX::Entry Provides an object-oriented interface to BibTeX entries, which can be parsed from `File` objects, arbitrary filehandles, or strings. Manages all the properties of a single entry: type, key, fields, and values. Also serves as the base class for the *structured entry classes* (described in detail in section C.5).

Text::BibTeX::Value Provides an object-oriented interface to *values* and *simple values*, high-level constructs that can be used to represent the strings associated with each field in an entry. Normally, field values are returned simply as Perl strings, with macros expanded and multiple strings "pasted" together. If desired, you can instruct Text::BibTeX to return Text::BibTeX::Value objects, which give you access to the original form of the data.

Text::BibTeX::Structure Provides the Structure and StructuredEntry classes, which serve primarily as base classes for the two kinds of classes that define database structures. Read this man page for a comprehensive description of the mechanism for implementing Perl classes analogous to BibTeX "style files".

Text::BibTeX::Bib Provides the BibStructure and BibEntry classes, which serve two purposes: they fulfill the same role as the standard style files of BibTeX 0.99, and they give an example of how to write new database structures. These ultimately derive from, respectively, the Structure and StructuredEntry classes provided by the Structure module.

Text::BibTeX::BibSort One of the BibEntry class's base classes: handles the generation of sort keys for sorting prior to output formatting.

Text::BibTeX::BibFormat One of the BibEntry class's base classes: handles the formatting of bibliographic data for output in a markup language such as LaTeX.

Text::BibTeX::Name A class used by the Bib structure and specific to bibliographic data as defined by BibTeX itself: parses individual author names into "first", "von", "last", and "jr" parts.

Text::BibTeX::NameFormat Also specific to bibliographic data: puts split-up names (as parsed by the Name class) back together in a custom way.

For a first time through the library, you'll probably want to confine your reading to section C.2 and section C.3. The other modules will come in handy eventually, especially if you need to emulate BibTeX in a fairly fine grained way (e.g. parsing names, generating sort keys). But for the simple database hacks that are the bread and butter of the Text::BibTeX library, the File and Entry classes are the bulk of what you'll need. You may also find some of the material in this manual page useful, namely section C.1.5 and section C.1.6.

C.1.4 Exports

The Text::BibTeX module has a number of optional exports, most of them constant values described in section C.1.5 below. The default exports are a subset of these constant values that are used particularly often, the "entry metatypes" (also accessible via the export tag `metatypes`). Thus, the following two lines are equivalent:

```
use Text::BibTeX;
use Text::BibTeX qw(:metatypes);
```

Some of the various subroutines provided by the module are also exportable. `bibloop`, `split_list`, `purify_string`, and `change_case` are all useful in everyday processing of BibTeX data, but don't really fit anywhere in the class hierarchy. They may be imported from Text::BibTeX using the `subs` export tag. `check_class` and `display_list` are also exportable, but only by name; they are not included in any export tag. (These two mainly exist for use by other modules in the library.) For instance, to use Text::BibTeX and import the entry metatype constants and the common subroutines:

```
use Text::BibTeX qw(:metatypes :subs);
```

Another group of subroutines exists for direct manipulation of the macro table maintained by the underlying C library. These functions (see section C.1.7.4, below) allow you to define, delete, and query the value of BibTeX macros (or "abbreviations"). They may be imported *en masse* using the `macrosubs` export tag:

```
use Text::BibTeX qw(:macrosubs);
```

C.1.5 Constant Values

The Text::BibTeX module makes a number of constant values available. These correspond to the values of various enumerated types in the underlying C library, **btparse**, and their meanings are more fully explained in the **btparse** documentation.

Each group of constants is optionally exportable using an export tag given in the descriptions below.

Entry metatypes `BTE_UNKNOWN`, `BTE_REGULAR`, `BTE_COMMENT`, `BTE_PREAMBLE`, `BTE_MACRODEF`. The `metatype` method in the Entry class always returns one of these values. The latter three describe, respectively, `comment`, `preamble`, and `string` entries; `BTE_REGULAR` describes all other entry types. `BTE_UNKNOWN` should never be seen (it's mainly useful for C code that might have to detect half-baked data structures). See also section B.1. Export tag: `metatypes`.

AST node types `BTAST_STRING`, `BTAST_MACRO`, `BTAST_NUMBER`. Used to distinguish the three kinds of simple values—strings, macros, and numbers. The SimpleValue class' `type` method always returns one of these three values. See also section C.4, section B.1. Export tag: `nodetypes`.

Name parts `BTN_FIRST`, `BTN_VON`, `BTN_LAST`, `BTN_JR`, `BTN_NONE`. Used to specify the various parts of a name after it has been split up. These are mainly useful when using the NameFormat class. See also section B.7 and section B.8. Export tag: `nameparts`.

Join methods `BTJ_MAYTIE`, `BTJ_SPACE`, `BTJ_FORCETIE`, `BTJ_NOTHING`. Used to tell the NameFormat class how to join adjacent tokens together; see section C.7 and section B.8. Export tag: `joinmethods`.

C.1.6 Utility Functions

Text::BibTeX provides several functions that operate outside of the normal class hierarchy. Of these, only `bibloop` is likely to be of much use to you in writing everyday BibTeX-hacking programs; the other two (`check_class` and `display_list`) are mainly provided for the use of other modules in the library. They are documented here mainly for completeness, but also because they might conceivably be useful in other circumstances.

bibloop (ACTION, FILES [, DEST])

Loops over all entries in a set of BibTeX files, performing some caller-supplied action on each entry. FILES should be a reference to the list of filenames to process, and ACTION a reference to a subroutine that will be called on each entry. DEST, if given, should be a Text::BibTeX::File object (opened for output) to which entries might be printed.

The subroutine referenced by ACTION is called with exactly one argument: the Text::BibTeX::Entry object representing the entry currently being processed. Information about both the entry itself and the file where it originated is available through this object; see section C.3. The ACTION subroutine is only called if the entry was successfully parsed; any syntax errors will result in a warning message being printed, and that entry being skipped. Note that *all* successfully parsed entries are passed to the ACTION subroutine, even `preamble`, `string`, and `comment` entries. To skip these pseudo-entries and only process "regular" entries, then your action subroutine should look something like this:

```
sub action {
   my $entry = shift;
   return unless $entry->metatype == BTE_REGULAR;
   # process $entry ...
}
```

If your action subroutine needs any more arguments, you can just create a closure (anonymous subroutine) as a wrapper, and pass it to `bibloop`:

```
sub action {
   my ($entry, $extra_stuff) = @_;
   # ...
}

my $extra = ...;
Text::BibTeX::bibloop (sub { &action ($_[0], $extra) }, \@files);
```

If the ACTION subroutine returns a true value and DEST was given, then the processed entry will be written to DEST.

check_class (PACKAGE, DESCRIPTION, SUPERCLASS, METHODS)

Ensures that a PACKAGE implements a class meeting certain requirements. First, it inspects Perl's symbol tables to ensure that a package named PACKAGE actually exists. Then, it ensures that the class named by PACKAGE derives from SUPERCLASS (using the universal method `isa`). This derivation might be through multiple inheritance, or through several generations of a class hierarchy; the only requirement is that SUPERCLASS is somewhere in PACKAGE's tree of base classes. Finally, it checks that PACKAGE provides each method listed in METHODS (a reference to a list of method names). This is done with the universal method `can`, so the methods might actually come from one of PACKAGE's base classes.

DESCRIPTION should be a brief string describing the class that was expected to be provided by PACKAGE. It is used for generating warning messages if any of the class requirements are not met.

This is mainly used by the supervisory code in Text::BibTeX::Structure, to ensure that user-supplied structure modules meet the rules required of them.

display_list (LIST, QUOTE)

Converts a list of strings to the grammatical conventions of a human language (currently, only English rules are supported). LIST must be a reference to a list of strings. If this list is empty, the empty string is returned. If it has one element, then just that element is returned. If it has two elements, then they are joined with the string `" and "` and the resulting string is returned. Otherwise, the list has N elements for $N >= 3$; elements 1..N-1 are joined with commas, and the final element is tacked on with an intervening `", and "`.

If QUOTE is true, then each string is encased in single quotes before anything else is done.

This is used elsewhere in the library for two very distinct purposes: for generating warning messages describing lists of fields that should be present or are conflicting in an entry, and for generating lists of author names in formatted bibliographies.

C.1.7 Miscellaneous Functions

In addition to loading the File and Entry modules, Text::BibTeX loads the XSUB code which bridges the Perl modules to the underlying C library, **btparse**. This XSUB code provides a number of miscellaneous utility functions, most of which are put into other packages in the Text::BibTeX family for use by the corresponding classes. (For instance, the XSUB code loaded by Text::BibTeX provides a function `Text::BibTeX::Entry::parse`, which is actually documented as the `parse` method of the Text::BibTeX::Entry class—see section C.3. However, for completeness this function—and all the other functions that become available when you `use Text::BibTeX`—are at least mentioned here. The only functions from this group that you're ever likely to use are described in section C.1.7.2.

C.1.7.1 Startup/shutdown functions These just initialize and shutdown the underlying C library. Don't call either one of them; the Text::BibTeX startup/shutdown code takes care of it as appropriate. They're just mentioned here for completeness.

initialize ()

cleanup ()

C.1.7.2 Generic string-processing functions

split_list (STRING, DELIM [, FILENAME [, LINE [, DESCRIPTION]]])

Splits a string on a fixed delimiter according to the BibTeX rules for splitting up lists of names. With BibTeX, the delimiter is hard-coded as `"and"`; here, you can supply any string. Instances of DELIM in

STRING are considered delimiters if they are at brace-depth zero, surrounded by whitespace, and not at the beginning or end of STRING; the comparison is case-insensitive. See section B.7 for full details of how splitting is done (it's *not* the same as Perl's `split` function).

Returns the list of strings resulting from splitting STRING on DELIM.

purify_string (STRING [, OPTIONS])

"Purifies" STRING in the BibTeX way (usually for generation of sort keys). See section B.9 for details; note that, unlike the C interface, `purify_string` does *not* modify STRING in-place. A purified copy of the input string is returned.

OPTIONS is currently unused.

change_case (TRANFORM, STRING [, OPTIONS])

Transforms the case of STRING according to TRANSFORM (a single character, one of `'u'`, `'l'`, or `'t'`). See section B.9 for details; again, `change_case` differs from the C interface in that STRING is not modified in-place—the input string is copied, and the transformed copy is returned.

C.1.7.3 Entry-parsing functions Although these functions are provided by the Text::BibTeX module, they are actually in the Text::BibTeX::Entry package. That's because they are implemented in C, and thus loaded with the XSUB code that Text::BibTeX loads; however, they are actually methods in the Text::BibTeX::Entry class. Thus, they are documented as methods in section C.3.

parse (ENTRY_STRUCT, FILENAME, FILEHANDLE)

parse_s (ENTRY_STRUCT, TEXT)

C.1.7.4 Macro table functions These functions allow direct access to the macro table maintained by **btparse**, the C library underlying Text::BibTeX. In the normal course of events, macro definitions always accumulate, and are only defined as a result of parsing a macro definition (`@string`) entry. **btparse** never deletes old macro definitions for you, and doesn't have any built-in default macros. If, for example, you wish to start fresh with new macros for every file, use `delete_all_macros`. If you wish to pre-define certain macros, use `add_macro_text`. (But note that the `Bib` structure, as part of its mission to emulate BibTeX 0.99, defines the standard "month name" macros for you.)

See also section B.6 in the **btparse** documentation for a description of the C interface to these functions.

add_macro_text (MACRO, TEXT [, FILENAME [, LINE]])

Defines a new macro, or redefines an old one. MACRO is the name of the macro, and TEXT is the text it should expand to. FILENAME and LINE are just used to generate any warnings about the macro definition. The only such warning occurs when you redefine an old macro: its value is overridden, and `add_macro_text()` issues a warning saying so.

delete_macro (MACRO)

Deletes a macro from the macro table. If MACRO isn't defined, takes no action.

delete_all_macros ()

Deletes all macros from the macro table.

macro_length (MACRO)

Returns the length of a macro's expansion text. If the macro is undefined, returns 0; no warning is issued.

macro_text (MACRO [, FILENAME [, LINE]])

Returns the expansion text of a macro. If the macro is not defined, issues a warning and returns `undef`. FILENAME and LINE, if supplied, are used for generating this warning; they should be supplied if you're looking up the macro as a result of finding it in a file.

C.1.7.5 Name-parsing functions

These are both private functions for the use of the Name class, and therefore are put in the `Text::BibTeX::Name` package. You should use the interface provided by that class for parsing names in the BibTeX style.

_split (NAME_STRUCT, NAME, FILENAME, LINE, NAME_NUM, KEEP_CSTRUCT)

free (NAME_STRUCT)

C.1.7.6 Name-formatting functions

These are private functions for the use of the NameFormat class, and therefore are put in the `Text::BibTeX::NameFormat` package. You should use the interface provided by that class for formatting names in the BibTeX style.

create ([PARTS [, ABBREV_FIRST]])

free (FORMAT_STRUCT)

_set_text (FORMAT_STRUCT, PART, PRE_PART, POST_PART, PRE_TOKEN, POST_TOKEN)

_set_options (FORMAT_STRUCT, PART, ABBREV, JOIN_TOKENS, JOIN_PART)

format_name (NAME_STRUCT, FORMAT_STRUCT)

C.1.8 Availability

The latest version of Text::BibTeX (and **btparse**, its sister library) is always available from

```
ftp://ftp.bic.mni.mcgill.ca/pub/btOOL/
```

in Text-BibTeX-x.y.tar.gz, where x.y is the version number. You will also find the latest version of **btparse**, the C library underlying Text::BibTeX, at that location. **btparse** is necessary to build Text::BibTeX, and must be downloaded separately.

Both libraries are also available on CTAN (the Comprehensive TeX Archive Network, `ftp://ctan.tug.org/tex-archive/`) and CPAN (the Comprehensive Perl Archive Network, `http://www.perl.com/CPAN-local/`). Look in /biblio/bibtex/utils/btparse on CTAN, and /modules/by-module/Text on CPAN.

Up-to-date information on Text::BibTeX and **btparse** is available at

```
http://www.bic.mni.mcgill.ca/~greg/btOOL/
```

C.2 Text::BibTeX::File

C.2.1 Synopsis

```
use Text::BibTeX;     # this loads Text::BibTeX::File

$bib = new Text::BibTeX::File "foo.bib" or die "foo.bib: $!\n";
# or:
$bib = new Text::BibTeX::File;
$bib->open ("foo.bib") || die "foo.bib: $!\n";

$bib->set_structure ($structure_name,
                     $option1 => $value1, ...);

$at_eof = $bib->eof;

$bib->close;
```

C.2.2 Description

Text::BibTeX::File provides an object-oriented interface to BibTeX files. Its most obvious purpose is to keep track of a filename and filehandle together for use by the Text::BibTeX::Entry module (which is much more interesting). In addition, it allows you to specify certain options which are applicable to a whole database (file), rather than having to specify them for each entry in the file. Currently, you can specify the *database structure* and some *structure options*. These concepts are fully documented in section C.5.

C.2.3 Methods

C.2.3.1 Object creation, file operations

new ([FILENAME [,MODE [,PERMS]]])

Creates a new Text::BibTeX::File object. If FILENAME is supplied, passes it to the `open` method (along with MODE and PERMS if they are supplied). If the `open` fails, `new` fails and returns false; if the `open` succeeds (or if FILENAME isn't supplied), `new` returns the new object reference.

open (FILENAME [,MODE [,PERMS]])

Opens the file specified by FILENAME, possibly using MODE and PERMS. See the IO::File manual page for full semantics; this `open` is just a front end for `IO::File::open`.

close ()

Closes the filehandle associated with the object. If there is no such filehandle (i.e., `open` was never called on the object), does nothing.

eof ()

Returns the end-of-file state of the filehandle associated with the object: a true value means we are at the end of the file.

C.2.3.2 Object properties

set_structure (STRUCTURE [, OPTION => VALUE, ...])

Sets the database structure for a BibTeX file. At the simplest level, this means that entries from the file are expected to conform to certain field requirements as specified by the *structure module*. It also gives you full access to the methods of the particular *structured entry class* for this structure, allowing you to perform operations specific to this kind of database. See section C.5.5 for all the consequences of setting the database structure for a Text::BibTeX::File object.

structure ()

Returns the name of the database structure associated with the object (as set by `set_structure`).

preserve_values ([PRESERVE])

Sets the "preserve values" flag, to control all future parsing of entries from this file. If PRESERVE isn't supplied, returns the current state of the flag. See section C.4 for details on parsing in "value preservation" mode.

C.3 Text::BibTeX::Entry

C.3.1 Synopsis

```
use Text::BibTeX;            # do not use Text::BibTeX::Entry alone!

# ...assuming that $bibfile and $newbib are both objects of class
# Text::BibTeX::File, opened for reading and writing (respectively):

# Entry creation/parsing methods:
$entry = new Text::BibTeX::Entry;
$entry->read ($bibfile);
$entry->parse ($filename, $filehandle);
$entry->parse_s ($entry_text);

# or:
$entry = new Text::BibTeX::Entry $bibfile;
$entry = new Text::BibTeX::Entry $filename, $filehandle;
$entry = new Text::BibTeX::Entry $entry_text;

# Entry query methods
warn "error in input" unless $entry->parse_ok;
$metatype = $entry->metatype;
$type = $entry->type;

# if metatype is BTE_REGULAR or BTE_MACRODEF:
$key = $entry->key;                  # only for BTE_REGULAR metatype
$num_fields = $entry->num_fields;
@fieldlist = $entry->fieldlist;
$has_title = $entry->exists ('title');
$title = $entry->get ('title');
# or:
($val1,$val2,...$valn) = $entry->get ($field1, $field2, ..., $fieldn);
```

```
# if metatype is BTE_COMMENT or BTE_PREAMBLE:
$value = $entry->value;

# Author name methods
@authors = $entry->split ('author');
($first_author) = $entry->names ('author');

# Entry modification methods
$entry->set_type ($new_type);
$entry->set_key ($new_key);
$entry->set ('title', $new_title);
# or:
$entry->set ($field1, $val1, $field2, $val2, ..., $fieldn, $valn);
$entry->delete (@fields);
$entry->set_fieldlist (\@fieldlist);

# Entry output methods
$entry->write ($newbib);
$entry->print ($filehandle);
$entry_text = $entry->print_s;

# Miscellaneous methods
$entry->warn ($entry_warning);
# or:
$entry->warn ($field_warning, $field);
```

C.3.2 Description

Text::BibTeX::Entry does all the real work of reading and parsing BibTeX files. (Well, actually it just provides an object-oriented Perl front-end to a C library that does all that. But that's not important right now.)

BibTeX entries can be read either from Text::BibTeX::File objects (using the `read` method), or directly from a filehandle (using the `parse` method), or from a string (using `parse_s`). The first is preferable, since you don't have to worry about supplying the filename, and because of the extra functionality provided by the Text::BibTeX::File class. Currently, this means that you may specify the *database structure* to which entries are expected to conform via the File class. This lets you ensure that entries follow the rules for required fields and mutually constrained fields for a particular type of database, and also gives you access to all the methods of the *structured entry class* for this database structure. See section C.5 for details on database structures.

Once you have the entry, you can query it or change it in a variety of ways. The query methods are `parse_ok`, `type`, `key`, `num_fields`, `fieldlist`, `exists`, and `get`. Methods for changing the entry are `set_type`, `set_key`, `set_fieldlist`, `delete`, and `set`.

Finally, you can output BibTeX entries, again either to an open Text::BibTeX::File object, a filehandle or a string. (A filehandle or File object must, of course, have been opened in write mode.) Output to a File object is done with the `write` method, to a filehandle via `print`, and to a string with `print_s`. Using the File class is recommended for future extensibility, although it currently doesn't offer anything extra.

C.3.3 Methods

C.3.3.1 Entry creation/parsing methods

new ([SOURCE])

Creates a new Text::BibTeX::Entry object. If the SOURCE parameter is supplied, it must be one of the following: a Text::BibTeX::File (or descendant class) object, a filename/filehandle pair, or a string. Calls `read` to read from a Text::BibTeX::File object, `parse` to read from a filehandle, and `parse_s` to read from a string.

A filehandle can be specified as a GLOB reference, or as an `IO::Handle` (or descendants) object, or as a `FileHandle` (or descendants) object. (But there's really no point in using `FileHandle` objects, since Text::BibTeX requires Perl 5.004, which always includes the `IO` modules.) You can *not* pass in the name of a filehandle as a string, though, because Text::BibTeX::Entry conforms to the `use strict` pragma (which disallows such symbolic references).

The corresponding filename should be supplied in order to allow for accurate error messages; if you simply don't have the filename, you can pass `undef` and you'll get error messages without a filename. (It's probably better to rearrange your code so that the filename is available, though.)

Thus, the following are equivalent to read from a file named by `$filename` (error handling ignored):

```
# good ol' fashioned filehandle and GLOB ref
open (BIBFILE, $filename);
$entry = new Text::BibTeX::Entry ($filename, \*BIBFILE);

# newfangled IO::File thingy
$file = new IO::File $filename;
$entry = new Text::BibTeX::Entry ($filename, $file);
```

But using a Text::BibTeX::File object is simpler and preferred:

```
$file = new Text::BibTeX::File $filename;
$entry = new Text::BibTeX::Entry $file;
```

Returns the new object, unless SOURCE is supplied and reading/parsing the entry fails (e.g., due to end of file) – then it returns false.

read (BIBFILE)

Reads and parses an entry from BIBFILE, which must be a Text::BibTeX::File object (or descendant). The next entry will be read from the file associated with that object.

Returns the same as `parse` (or `parse_s`): false if no entry found (e.g., at end-of-file), true otherwise. To see if the parse itself failed (due to errors in the input), call the `parse_ok` method.

parse (FILENAME, FILEHANDLE)

Reads and parses the next entry from FILEHANDLE. (That is, it scans the input until an '@' sign is seen, and then slurps up to the next '@' sign. Everything between the two '@' signs [including the first one, but not the second one – it's pushed back onto the input stream for the next entry] is parsed as a BibTeX entry, with the simultaneous construction of an abstract syntax tree [AST]. The AST is traversed to ferret out the most interesting information, and this is stuffed into a Perl hash, which coincidentally is the Text::BibTeX::Entry object you've been tossing around. But you don't need to know any of that – I just figured if you've read this far, you might want to know something about the inner workings of this module.)

The success of the parse is stored internally so that you can later query it with the `parse_ok` method. Even in the presence of syntax errors, you'll usually get something resembling your input, but it's usually not wise to try to do anything with it. Just call `parse_ok`, and if it returns false then silently skip to the next entry. (The error messages printed out by the parser should be quite adequate for the user to figure out what's wrong. And no, there's currently no way for you to capture or redirect those error messages – they're always printed to `stderr` by the underlying C code. That should change in future releases.)

If no '@' signs are seen on the input before reaching end-of-file, then we've exhausted all the entries in the file, and `parse` returns a false value. Otherwise, it returns a true value – even if there were syntax errors. Hence, it's important to check `parse_ok`.

The FILENAME parameter is only used for generating error messages, but anybody using your program will certainly appreciate your setting it correctly!

parse_s (TEXT)

Parses a BibTeX entry (using the above rules) from the string TEXT. The string is not modified; repeatedly calling `parse_s` with the same string will give you the same results each time. Thus, there's no point in putting multiple entries in one string.

C.3.3.2 Entry query methods

parse_ok ()

Returns false if there were any serious errors encountered while parsing the entry. (A "serious" error is a lexical or syntax error; currently, warnings such as "undefined macro" result in an error message being printed to `stderr` for the user's edification, but no notice is available to the calling code.)

type ()

Returns the type of the entry. (The 'type' is the word that follows the '@' sign; e.g. 'article', 'book', 'inproceedings', etc. for the standard BibTeX styles.)

metatype ()

Returns the metatype of the entry. (The 'metatype' is a numeric value used to classify entry types into four groups: comment, preamble, macro definition (`@string` entries), and regular (all other entry types). Text::BibTeX exports four constants for these metatypes: `BTE_COMMENT`, `BTE_PREAMBLE`, `BTE_MACRODEF`, and `BTE_REGULAR`.)

key ()

Returns the key of the entry. (The key is the token immediately following the opening '–' or '(' in "regular" entries. Returns `undef` for entries that don't have a key, such as macro definition (`@string`) entries.)

num_fields ()

Returns the number of fields in the entry. (Note that, currently, this is *not* equivalent to putting `scalar` in front of a call to `fieldlist`. See below for the consequences of calling `fieldlist` in a scalar context.)

fieldlist ()

Returns the list of fields in the entry. In a scalar context, returns a reference to the object's own list of fields. That way, you can change or reorder the field list with minimal interference from the class. I'm not entirely sure if this is a good idea, so don't rely on it existing in the future; feel free to play around with it and let me know if you get bitten in dangerous ways or find this enormously useful.

exists (FIELD)

Returns true if a field named FIELD is present in the entry, false otherwise.

get (FIELD, ...)

Returns the value of one or more FIELDs, as a list of values. For example:

```
$author = $entry->get ('author');
($author, $editor) = $entry->get ('author', 'editor');
```

If a FIELD is not present in the entry, `undef` will be returned at its place in the return list. However, you can't completely trust this as a test for presence or absence of a field; it is possible for a field to be present but undefined. Currently this can only happen due to certain syntax errors in the input, or if you pass an undefined value to `set`, or if you create a new field with `set_fieldlist` (the new field's value is implicitly set to `undef`).

Normally, the field value is what the input looks like after "maximal processing"–quote characters are removed, whitespace is collapsed (the same way that BibTeX itself does it), macros are expanded, and multiple tokens are pasted together. (See section B.5 for details on the post-processing performed by **btparse**.)

For example, if your input file has the following:

```
@string{of = "of"}
@string{foobars = "Foobars"}

@article{foobar,
  title = {   The Mating Habits      } # of # " Adult   " # foobars
}
```

then using `get` to query the value of the `title` field from the `foobar` entry would give the string "The Mating Habits of Adult Foobars".

However, in certain circumstances you may wish to preserve the values as they appear in the input. This is done by setting a `preserve_values` flag at some point; then, `get` will return not strings but Text::BibTeX::Value objects. Each Value object is a list of Text::BibTeX::SimpleValue objects, which in turn consists of a simple value type (string, macro, or number) and the text of the simple value. Various ways to set the `preserve_values` flag and the interface to both Value and SimpleValue objects are described in section C.4.

value ()

Retuns the single string associated with `@comment` and `@preamble` entries. For instance, the entry

```
@preamble{" This is   a preamble" #
          {---the concatenation of several strings}}
```

would return a value of "This is a preamble—the concatenation of several strings".

If this entry was parsed in "value preservation" mode, then `value` acts like `get`, and returns a Value object rather than a simple string.

C.3.3.3 Author name methods This is the only part of the module that makes any assumption about the nature of the data, namely that certain fields are lists delimited by a simple word such as "and", and that the delimited sub-strings are human names of the "First von Last" or "von Last, Jr., First" style used by BibTeX. If you are using this module for anything other than bibliographic data, you can most likely forget about these two methods. However, if you are in fact hacking on BibTeX-style bibliographic data, these could come in very handy – the name-parsing done by BibTeX is not trivial, and the list-splitting would also be a pain to implement in Perl because you have to pay attention to brace-depth. (Not that it wasn't a pain to implement in C – it's just a lot more efficient than a Perl implementation would be.)

Incidentally, both of these methods assume that the strings being split have already been "collapsed" in the BibTeX way, i.e. all leading and trailing whitespace removed and internal whitespace reduced to single spaces. This should always be the case when using these two methods on a Text::BibTeX::Entry object, but these are actually just front ends to more general functions in Text::BibTeX. (More general in that you supply the string to be parsed, rather than supplying the name of an entry field.) Should you ever use those more general functions directly, you might have to worry about collapsing whitespace; see section C.1 (the `split_list` and `split_name` functions in particular) for more information.

Please note that the interface to author name parsing is experimental, subject to change, and open to discussion. Please let me know if you have problems with it, think it's just perfect, or whatever.

split (FIELD [, DELIM [, DESC]])

Splits the value of FIELD on DELIM (default: 'and'). Don't assume that this works the same as Perl's builtin `split` just because the names are the same: in particular, DELIM must be a simple string (no regexps), and delimiters that are at the beginning or end of the string, or at non-zero brace depth, or not surrounded by whitespace, are ignored. Some examples might illuminate matters:

```
if field F is...                 then split (F) returns...
'Name1 and Name2'                ('Name1', 'Name2')
'Name1 and and Name2'            ('Name1', undef, 'Name2')
'Name1 and'                      ('Name1 and')
'and Name2'                      ('and Name2')
'Name1 {and} Name2 and Name3'    ('Name1 {and} Name2', 'Name3')
'{Name1 and Name2} and Name3'    ('{Name1 and Name2}', 'Name3')
```

Note that a warning will be issued for empty names (as in the second example above). A warning ought to be issued for delimiters at the beginning or end of a string, but currently this isn't done. (Hmmm.)

DESC is a one-word description of the substrings; it defaults to 'name'. It is only used for generating warning messages.

names (FIELD)

Splits FIELD as described above, and further splits each name into four components: first, von, last, and jr.

Returns a list of Text::BibTeX::Name objects, each of which represents one name. Use the `part` method to query these objects; see section C.6 for details on the interface to name objects (and on name-parsing as well).

For example if this entry:

```
@article{foo,
         author = {John Smith and
                   Hacker, J. Random and
                   Ludwig van Beethoven and
                   {Foo, Bar and Company}}}
```

has been parsed into a Text::BibTeX::Entry object `$entry`, then

```
@names = $entry->names ('author');
```

will put a list of Text::BibTeX::Name objects in `@names`. These can be queried individually as described in section C.6; for instance,

```
@last = $names[0]->part ('last');
```

would put the list of tokens comprising the last name of the first author into the `@last` array: `('Smith')`.

C.3.3.4 Entry modification methods

set_type (TYPE)

Sets the entry's type.

set_key (KEY)

Sets the entry's key.

set (FIELD, VALUE, ...)

Sets the value of field FIELD. (VALUE might be `undef` or unsupplied, in which case FIELD will simply be set to `undef` – this is where the difference between the `exists` method and testing the definedness of field values becomes clear.)

Multiple (FIELD, VALUE) pairs may be supplied; they will be processed in order (i.e. the input is treated like a list, not a hash). For example:

```
$entry->set ('author', $author);
$entry->set ('author', $author, 'editor', $editor);
```

VALUE can be either a simple string or a Text::BibTeX::Value object; it doesn't matter if the entry was parsed in "full post-processing" or "preserve input values" mode.

delete (FIELD)

Deletes field FIELD from an entry.

set_fieldlist (FIELDLIST)

Sets the entry's list of fields to FIELDLIST, which must be a list reference. If any of the field names supplied in FIELDLIST are not currently present in the entry, they are created with the value `undef` and a warning is printed. Conversely, if any of the fields currently present in the entry are not named in the list of fields supplied to `set_fields`, they are deleted from the entry and another warning is printed.

C.3.3.5 Entry output methods

write (BIBFILE)

Prints a BibTeX entry on the filehandle associated with BIBFILE (which should be a Text::BibTeX::File object, opened for output). Currently the printout is not particularly human-friendly; a highly configurable pretty-printer will be developed eventually.

print (FILEHANDLE)

Prints a BibTeX entry on FILEHANDLE.

print_s ()

Prints a BibTeX entry to a string, which is the return value.

C.3.3.6 Miscellaneous methods

warn (WARNING [, FIELD])

Prepends a bit of location information (filename and line number(s)) to WARNING, appends a newline, and passes it to Perl's `warn`. If FIELD is supplied, the line number given is just that of the field; otherwise, the range of lines for the whole entry is given. (Well, almost – currently, the line number of the last field is used as the last line of the whole entry. This is a bug.)

For example, if lines 10-15 of file foo.bib look like this:

```
@article{homer97,
  author = {Homer Simpson and Ned Flanders},
  title = {Territorial Imperatives in Modern Suburbia},
  journal = {Journal of Suburban Studies},
  year = 1997
}
```

then, after parsing this entry to `$entry`, the calls

```
$entry->warn ('what a silly entry');
$entry->warn ('what a silly journal', 'journal');
```

would result in the following warnings being issued:

```
foo.bib, lines 10-14: what a silly entry
foo.bib, line 13: what a silly journal
```

line ([FIELD])

Returns the line number of FIELD. If the entry was parsed from a string, this still works–it's just the line number relative to the start of the string. If the entry was parsed from a file, this works just as you'd expect it to: it returns the absolute line number with respect to the whole file. Line numbers are one-based.

If FIELD is not supplied, returns a two-element list containing the line numbers of the beginning and end of the whole entry. (Actually, the "end" line number is currently inaccurate: it's really the the line number of the last field in the entry. But it's better than nothing.)

filename ()

Returns the name of the file from which the entry was parsed. Only works if the file is represented by a Text::BibTeX::File object—if you just passed a filename/filehandle pair to `parse`, you can't get the filename back. (Sorry.)

C.4 Text::BibTeX::Value

C.4.1 Synopsis

```
use Text::BibTeX;

$entry = new Text::BibTeX::Entry;

# set the 'preserve_values' flag to 1 for this parse
$entry->parse ($filename, $filehandle, 1);

# 'get' method now returns a Text::BibTeX::Value object
# rather than a string
$value = $entry->get ($field);

# query the 'Value' object (list of SimpleValue objects)
@all_values = $value->values;
$first_value = $value->value (0);
$last_value = $value->value (-1);

# query the simple value objects -- type will be one of BTAST_STRING,
# BTAST_MACRO, or BTAST_NUMBER
use Text::BibTex (':nodetypes');   # import "node type" constants
$is_macro = ($first_value->type == BTAST_MACRO);
$text = $first_value->text;
```

C.4.2 Description

The Text::BibTeX::Value module provides two classes, Text::BibTeX::Value and Text::BibTeX::SimpleValue, which respectively give you access to BibTeX "compound values" and "simple values". Recall that every field value in a BibTeX entry is the concatenation of one or more simple values, and that each of those simple values may be a literal string, a macro (abbreviation), or a number. Normally with Text::BibTeX, field values are "fully processed," so that you only have access to the string that results from expanding macros, converting numbers to strings, concatenating all sub-strings, and collapsing whitespace in the resulting string.

For example, in the following entry:

```
@article{homer97,
  author = "Homer Simpson" # and # "Ned Flanders",
  title = {Territorial Imperatives in Modern Suburbia},
  journal = jss,
  year = 1997
}
```

we see the full range of options. The `author` field consists of three simple values: a string, a macro (`and`), and another string. The `title` field is a single string, and the `journal` and `year` fields are, respectively, a single macro and a single number. If you parse this entry in the usual way:

```
$entry = new Text::BibTeX::Entry $entry_text;
```

then the `get` method on `$entry` would return simple strings. Assuming that the `and` macro is defined as `" and "`, then

```
$entry->get ('author')
```

would return the Perl string `"Homer Simpson and Ned Flanders"`.

However, you can also request that the library preserve the input values in your entries, i.e. not lose the information about which values use macros, which values are composed of multiple simple values, and so on. There are two ways to make this request: per-file and per-entry. For a per-file request, use the `preserve_values` method on your File object:

```
$bibfile = new Text::BibTeX::File $filename;
$bibfile->preserve_values (1);

$entry = new Text::BibTeX::Entry $bibfile;
$entry->get ($field);        # returns a Value object

$bibfile->preserve_values (0);
$entry = new Text::BibTeX::Entry $bibfile;
$entry->get ($field);        # returns a string
```

If you're not using a File object, or want to control things at a finer scale, then you have to pass in the `preserve_values` flag when invoking `read`, `parse`, or `parse_s` on your Entry objects:

```
# no File object, parsing from a string
$entry = new Text::BibTeX::Entry;
$entry->parse_s ($entry_text, 0);  # preserve_values=0 (default)
$entry->get ($field);        # returns a string

$entry->parse_s ($entry_text, 1);
$entry->get ($field);        # returns a Value object
```

```
# using a File object, but want finer control
$entry->read ($bibfile, 0);  # now get will return strings (default)
$entry->read ($bibfile, 1);  # now get will return Value objects
```

A compound value, usually just called a value, is simply a list of simple values. The Text::BibTeX::Value class (hereinafter abbreviated as Value) provides a simple interface to this list; you can request the whole list, or an individual member of the list. The SimpleValue class gives you access to the innards of each simple value, which consist of the *type* and the *text*. The type just tells you if this simple value is a string, macro, or number; it is represented using the Perl translation of the "node type" enumeration from C. The possible types are `BTAST_STRING`, `BTAST_NUMBER`, and `BTAST_MACRO`. The text is just what appears in the original entry text, be it a string, number, or macro.

For example, we could parse the above entry in "preserve values" mode as follows:

```
$entry->parse_s ($entry_text, 1);   # preserve_values is 1
```

Then, using the `get` method on `$entry` would return not a string, but a Value object. We can get the list of all simple values using the `values` method, or a single value using `value`:

```
$author = $entry->get ('author');   # now a Text::BibTeX::Value object
@all_values = $author->values;      # array of Text::BibTeX::SimpleValue
$second = $author->value (1);       # same as $all_values[1]
```

The simple values may be queried using the Text::BibTeX::SimpleValue methods, `type` and `text`:

```
$all_values[0]->type;               # returns BTAST_STRING
$second->type;                      # returns BTAST_MACRO

$all_values[0]->text;               # "Homer Simpson"
$second->text;                      # "and" (NOT the macro expansion!)

$entry->get ('year')->value (0)->text;   # "1997"
```

C.4.3 Methods

Normally, you won't need to create Value or SimpleValue objects—they'll be created for you when an entry is parsed, and returned to you by the `get` method in the Entry class. Thus, the query methods (`values` and `value` for the Value class, `type` and `text` for SimpleValue) are probably all you need to worry about. If you wish, though, you can create new values and simple values using the two classes' respective constructors. You can also put newly-created Value objects back into an existing Entry object using the `set` entry method; it doesn't matter how the entry was parsed, this is acceptable anytime.

C.4.3.1 Text::BibTeX::Value methods

new (SVAL, ...)

Creates a new Value object from a list of simple values. Each simple value, SVAL, may be either a SimpleValue object or a reference to a two-element list containing the type and text of the simple value. For example, one way to recreate the `author` field of the example entry in section C.4.2 would be:

```
$and_macro = new Text::BibTeX::SimpleValue (BTAST_MACRO, 'and');
$value = new Text::BibTeX::Value
   ([BTAST_STRING, 'Homer Simpson'],
    $and_macro,
    [BTAST_STRING, 'Ned Flanders']);
```

The resulting Value object could then be installed into an entry using the `set` method of the Entry class.

values ()

Returns the list of SimpleValue objects that make up a Value object.

value (NUM)

Returns the NUM'th SimpleValue object from the list of SimpleValue objects that make up a Value object. This is just like a Perl array reference: NUM is zero-based, and negative numbers count from the end of the array.

C.4.3.2 Text::BibTeX::SimpleValue methods

new (TYPE, TEXT)

Creates a new SimpleValue object with the specified TYPE and TEXT. TYPE must be one of the allowed types for BibTeX simple values, i.e. `BTAST_STRING`, `BTAST_NUMBER`, or `BTAST_MACRO`. You'll probably want to import these constants from Text::BibTeX using the `nodetypes` export tag:

```
use Text::BibTeX qw(:nodetypes);
```

TEXT may be any string. Note that if TYPE is `BTAST_NUMBER` and TEXT is not a string of digits, the SimpleValue object will be created anyways, but a warning will be issued. No warning is issued about non-existent macros.

type ()

Returns the type of a simple value. This will be one of the allowed "node types" as described under the new item above.

text ()

Returns the text of a simple value. This is just the text that appears in the original entry—unexpanded macro name, or unconverted number. (Of course, converting numbers doesn't make any difference from Perl; in fact, it's all the same in C too, since the C code just keeps numbers as strings of digits. It's simply a matter of whether the string of digits is represented as a string or a number, which you might be interested in knowing if you want to preserve the structure of the input as much possible.)

C.5 Text::BibTeX::Structure

C.5.1 Synopsis

```
# Define a 'Foo' structure for BibTeX databases: first, the
# structure class:
```

```
package Text::BibTeX::FooStructure;
@ISA = ('Text::BibTeX::Structure');

sub known_option
{
   my ($self, $option) = @_;

   ...
}

sub default_option
{
   my ($self, $option) = @_;

   ...
}

sub describe_entry
{
   my $self = shift;

   $self->set_fields ($type,
                      \@required_fields,
                      \@optional_fields,
                      [$constraint_1, $constraint_2, ...]);
   ...
}

# Now, the structured entry class

package Text::BibTeX::FooEntry;
@ISA = ('Text::BibTeX::StructuredEntry');

# define whatever methods you like
```

C.5.2 Description

The module Text::BibTeX::Structure provides two classes that form the basis of the **btOOL** "structure module" system. This system is how database structures are defined and imposed on BibTeX files, and provides an elegant synthesis of object-oriented techniques with BibTeX-style database structures. Nothing described here is particularly deep or subtle; anyone familar with object-oriented programming should be able to follow it. However, a fair bit of jargon in invented and tossed around, so pay attention.

A *database structure*, in **btOOL** parlance, is just a set of allowed entry types and the rules for fields in each of those entry types. Currently, there are three kinds of rules that apply to fields: some fields are *required*, meaning they must be present in every entry for a given type; some are *optional*, meaning they may be present, and will be used if they are; other fields are members of *constraint sets*, which are explained in section C.5.3.1 below.

A **btOOL** structure is implemented with two classes: the *structure class* and the *structured entry class*. The former defines everything that applies to the structure as a whole (allowed types and field rules). The latter provides methods that operate on individual entries which conform (or are supposed to conform) to the structure. The two classes provided by the Text::BibTeX::Structure module are Text::BibTeX::Structure and Text::BibTeX::StructuredEntry; these serve as base classes for, respectively, all structure classes and all structured entry classes. One canonical structure is provided as an example with **btOOL**: the `Bib` structure, which (via the BibStructure and BibEntry classes) provides the same functionality as the standard style files of BibTeX 0.99. It is hoped that other programmers will write new bibliography-related structures, possibly deriving from the `Bib` structure, to emulate some of the functionality that is available through third-party BibTeX style files.

The purpose of this manual page is to describe the whole "structure module" system. It is mainly for programmers wishing to implement a new database structure for data files with BibTeX syntax; if you are interested in the particular rules for the BibTeX-emulating `Bib` structure, see section C.8.

Please note that the `Text::BibTeX` prefix is dropped from most module and class names in this manual page, except where necessary.

C.5.3 Structure Classes

Structure classes have two roles: to define the list of allowed types and field rules, and to handle *structure options*.

C.5.3.1 Field lists and constraint sets Field lists and constraint sets define the database structure for a particular entry type: that is, they specify the rules which an entry must follow to conform to the structure (assuming that entry is of an allowed type). There are three components to the field rules for each entry type: a list of required fields, a list of optional fields, and *field constraints*. Required and optional fields should be obvious to anyone with BibTeX experience: all required fields must be present, and any optional fields that are present have some meaning to the structure. (One could conceive of a "strict" interpretation, where any field not mentioned in the official definition is disallowed; this would be contrary to the open spirit of BibTeX databases, but could be useful in certain applications where a stricter level of control is desired. Currently, **btOOL** does not offer such an option.)

Field constraints capture the "one or the other, but not both" type of relationships present for some entry types in the BibTeX standard style files. Most BibTeX documentation glosses over the distinction between mutually constrained fields and required/optional fields. For instance, one of the standard entry types is `book`, and "`author` or `editor`" is given in the list of required fields for that type. The meaning of this is that an entry of type `book` must have *either* the `author` or `editor` fields, but not both. Likewise, the "`volume` or `number`" are listed under the "optional fields" heading for `book` entries; it would be more accurate to say that every `book` entry may have one or the other, or neither, of `volume` or `number`—but not both.

btOOL attempts to clarify this situation by creating a third category of fields, those that are mutually constrained. For instance, neither `author` nor `editor` appears in the list of required fields for the `inbook` type according to **btOOL**; rather, a field constraint is created to express this relationship:

```
[1, 1, ['author', 'editor']]
```

That is, a field constraint is a reference to a three-element list. The last element is a reference to the *constraint set*, the list of fields to which the constraint applies. (Calling this a set is a bit inaccurate, as there are conditions in which the order of fields matters—see the `check_field_constraints` method in section C.5.8.) The first two elements are the minimum and maximum number of fields from the constraint set that must be present for an entry to conform to the constraint. This constraint thus expresses that there must be exactly one (¿= 1 and ¡= 1) of the fields `author` and `editor` in a `book` entry.

The "either one or neither, but not both" constraint that applies to the `volume` and `number` fields for `book` entries is expressed slightly differently:

```
[0, 1, ['volume', 'number']]
```

That is, either 0 or 1, but not the full 2, of `volume` and `number` may be present.

It is important to note that checking and enforcing field constraints is based purely on counting which fields from a set are actually present; this mechanism can't capture "x must be present if y is" relationships.

The requirements imposed on the actual structure class are simple: it must provide a method `describe_entry` which sets up a fancy data structure describing the allowed entry types and all the field rules for those types. The Structure class provides methods (inherited by a particular structure class) to help particular structure classes create this data structure in a consistent, controlled way. For instance, the `describe_structure` method in the BibTeX 0.99-emulating BibStructure class is quite simple:

```
sub describe_entry
{
   my $self = shift;

   # series of 13 calls to $self->set_fields (one for each standard
   # entry type)
}
```

One of those calls to the `set_fields` method defines the rules for `book` entries:

```
$self->set_fields ('book',
                   [qw(title publisher year)],
                   [qw(series address edition month note)],
                   [1, 1, [qw(author editor)]],
                   [0, 1, [qw(volume number)]]);
```

The first field list is the list of required fields, and the second is the list of optional fields. Any number of field constraints may follow the list of optional fields; in this case, there are two, one for each of the constraints (`author`/`editor` and `volume`/`number`) described above. At no point is a list of allowed types explicitly supplied; rather, each call to `set_fields` adds one more allowed type.

New structure modules that derive from existing ones will probably use the `add_fields` method (and possibly `add_constraints`) to augment an existing entry type. Adding new types should be done with `set_fields`, though.

C.5.3.2 Structure options The other responsibility of structure classes is to handle *structure options*. These are scalar values that let the user customize the behaviour of both the structure class and the structured entry class. For instance, one could have an option to enable "extended structure", which might add on a bunch of new entry types and new fields. (In this case, the `describe_entry` method would have to pay attention to this option and modify its behaviour accordingly.) Or, one could have options to control how the structured entry class sorts or formats entries (for bibliography structures such as `Bib`).

The easy way to handle structure options is to provide two methods, `known_option` and `default_option`. These return, respectively, whether a given option is supported, and what its default value is. (If your structure doesn't support any options, you can just inherit these methods from the Structure class. The default `known_option` returns false for all options, and its companion `default_option` crashes with an "unknown option" error.)

Once `known_option` and `default_option` are provided, the structure class can sit back and inherit the more visible `set_options` and `get_options` methods from the Structure class. These are the methods actually used to modify/query options, and will be used by application programs to customize the structure module's behaviour, and by the structure module itself to pay attention to the user's wishes.

Options should generally have pure string values, so that the generic set_options method doesn't have to parse user-supplied strings into some complicated structure. However, `set_options` will take any scalar value, so if the structure module clearly documents its requirements, the application program could supply a structure that meets its needs. Keep in mind that this requires cooperation between the application and the structure module; the intermediary code in Text::BibTeX::Structure knows nothing about the format or syntax of your structure's options, and whatever scalar the application passes via `set_options` will be stored for your module to retrieve via `get_options`.

As an example, the `Bib` structure supports a number of "markup" options that allow applications to control the markup language used for formatting bibliographic entries. These options are naturally paired, as formatting commands in markup languages generally have to be turned on and off. The `Bib` structure thus expects references to two-element lists for markup options; to specify LaTeX 2e-style emphasis for book titles, an application such as `btformat` would set the `btitle_mkup` option as follows:

```
$structure->set_options (btitle_mkup => ['\emph{', '}']);
```

Other options for other structures might have a more complicated structure, but it's up to the structure class to document and enforce this.

C.5.4 Structured Entry Classes

A *structured entry class* defines the behaviour of individual entries under the regime of a particular database structure. This is the *raison d'être* for any database structure: the structure class merely lays out the rules for entries to conform to the structure, but the structured entry class provides the methods that actually operate on individual entries. Because this is completely open-ended, the requirements of a structured entry class are much less rigid than for a structure class. In fact, all of the requirements of a structured entry class can be met simply by inheriting from Text::BibTeX::StructuredEntry, the other class provided by the Text::BibTeX::Structure module. (For the record, those requirements are: a structured entry class must provide the entry parse/query/manipulate methods of the Entry class, and it must provide the `check`, `coerce`, and `silently_coerce` methods of the StructuredEntry class. Since StructuredEntry inherits from Entry, both of these requirements are met "for free" by structured entry classes that inherit from Text::BibTeX::StructuredEntry, so naturally this is the recommended course of action!)

There are deliberately no other methods required of structured entry classes. A particular application (eg. `btformat` for bibliography structures) will require certain methods, but it's up to the application and the structure module to work out the requirements through documentation.

C.5.5 Class Interactions

Imposing a database structure on your entries sets off a chain reaction of interactions between various classes in the Text::BibTeX library that should be transparent when all goes well. It could prove confusing if things go wrong and you have to go wading through several levels of application program, core Text::BibTeX classes, and some structure module.

The justification for this complicated behaviour is that it allows you to write programs that will use a particular structured module without knowing the name of the structure when you write the program. Thus, the user can supply a database structure, and ultimately the entry objects you manipulate will be blessed into a class supplied by the structure module. A short example will illustrate this.

Typically, a Text::BibTeX-based program is based around a kernel of code like this:

```
$bibfile = new Text::BibTeX::File "foo.bib";
while ($entry = new Text::BibTeX::Entry $bibfile)
{
   # process $entry
}
```

In this case, nothing fancy is happening behind the scenes: the `$bibfile` object is blessed into the Text::BibTeX::File class, and `$entry` is blessed into Text::BibTeX::Entry. This is the conventional behaviour of Perl classes, but it is not the only possible behaviour. Let us now suppose that `$bibfile` is expected to conform to a database structure specified by `$structure` (presumably a user-supplied value, and thus unknown at compile-time):

```
$bibfile = new Text::BibTeX::File "foo.bib";
$bibfile->set_structure ($structure);
while ($entry = new Text::BibTeX::Entry $bibfile)
{
   # process $entry
}
```

A lot happens behind the scenes with the call to `$bibfile`'s `set_structure` method. First, a new structure object is created from `$structure`. The structure name implies the name of a Perl module—the structure module—which is `require`'d by the Structure constructor. (The main consequence of this is that any compile-time errors in your structure module will not be revealed until a `Text::BibTeX::File::set_structure` or `Text::BibTeX::Structure::new` call attempts to load it.)

Recall that the first responsibility of a structure module is to define a structure class. The "structure object" created by the `set_structure` method call is actually an object of this class; this is the first bit of trickery—the structure object (buried behind the scenes) is blessed into a class whose name is not known until run-time.

Now, the behaviour of the `Text::BibTeX::Entry::new` constructor changes subtly: rather than returning an object blessed into the Text::BibTeX::Entry class as you might expect from the code, the object is blessed into the structured entry class associated with `$structure`.

For example, if the value of `$structure` is `"Foo"`, that means the user has supplied a module implementing the `Foo` structure. (Ordinarily, this module would be called Text::BibTeX::Foo—but you can customize this.) Calling the `set_structure` method on `$bibfile` will attempt to create a new structure object via the Text::BibTeX::Structure constructor, which loads the structure module Text::BibTeX::Foo. Once this module is successfully loaded, the new object is blessed into its structure class, which will presumably be called Text::BibTeX::FooStructure (again, this is customizable). The new object is supplied with the user's structure options via the `set_options` method (usually inherited), and then it is asked to describe the actual entry layout by calling its `describe_entry` method. This, in turn, will usually call the inherited `set_fields` method for each entry type in the database structure. When the Structure constructor is finished, the new structure object is stored in the File object (remember, we started all this by calling `set_structure` on a File object) for future reference.

Then, when a new Entry object is created and parsed from that particular File object, some more trickery happens. Trivially, the structure object stored in the File object is also stored in the Entry object. (The idea is that entries could belong to a database structure independently of any file, but usually they will just get the structure that was assigned to their database file.) More importantly, the new Entry object is re-blessed into the structured entry class supplied by the structure module—presumably, in this case, Text::BibTeX::FooEntry (also customizable).

Once all this sleight-of-hand is accomplished, the application may treat its entry objects as objects of the structured entry class for the `Foo` structure—they may call the check/coerce methods inherited from Text::BibTeX::StructuredEntry, and they may also call any methods specific to entries for this particular database structure. What these methods might be is up to the structure implementor to decide and document; thus, applications may be specific to one particular database structure, or they may work on all structures that supply certain methods. The choice is up to the application developer, and the range of options open to him depends on which methods structure implementors provide.

C.5.6 Example

For example code, please refer to the source of the Bib module and the `btcheck`, `btsort`, and `btformat` applications supplied with Text::BibTeX.

C.5.7 Methods 1: Base Structure Class

The first class provided by the Text::BibTeX::Structure module is Text::BibTeX::Structure. This class is intended to provide methods that will be inherited by user-supplied structure classes; such classes should not override any of the methods described here (except `known_option` and `default_option`) without very good reason. Furthermore, overriding the `new` method would be useless, because in general applications won't know the name of your structure class—they can only call `Text::BibTeX::Structure::new` (usually via `Text::BibTeX::File::set_structure`).

Finally, there are three methods that structure classes should implement: `known_option`, `default_option`, and `describe_entry`. The first two are described in section C.5.3.2 above, the latter in section C.5.3.1. Note that `describe_entry` depends heavily on the `set_fields`, `add_fields`, and `add_constraints` methods described here.

C.5.7.1 Constructor/simple query methods

new (STRUCTURE, [OPTION => VALUE, ...])

Constructs a new structure object—*not* a Text::BibTeX::Structure object, but rather an object blessed into the structure class associated with STRUCTURE. More precisely:

- Loads (with `require`) the module implementing STRUCTURE. In the absence of other information, the module name is derived by appending STRUCTURE to `"Text::BibTeX::"`—thus, the module Text::BibTeX::Bib implements the `Bib` structure. Use the pseudo-option `module` to override this module name. For instance, if the structure `Foo` is implemented by the module Foo:

```
$structure = new Text::BibTeX::Structure
   ('Foo', module => 'Foo');
```

This method `dies` if there are any errors loading/compiling the structure module.

- Verifies that the structure module provides a structure class and a structured entry class. The structure class is named by appending `"Structure"` to the name of the module, and the structured entry class by appending `"Entry"`. Thus, in the absence of a `module` option, these two classes (for the `Bib` structure) would be named Text::BibTeX::BibStructure and Text::BibTeX::BibEntry. Either or both of the default class names may be overridden by having the structure module return a reference to a hash (as opposed to the traditional `1` returned by modules). This hash could then supply a `structure_class` element to name the structure class, and an `entry_class` element to name the structured entry class.

  Apart from ensuring that the two classes actually exist, `new` verifies that they inherit correctly (from Text::BibTeX::Structure and Text::BibTeX::StructuredEntry respectively), and that the structure class provides the required `known_option`, `default_option`, and `describe_entry` methods.
- Creates the new structure object, and blesses it into the structure class. Supplies it with options by passing all (OPTION, VALUE) pairs to its `set_options` method. Calls its `describe_entry` method, which should list the field requirements for all entry types recognized by this structure. `describe_entry` will most likely use some or all of the `set_fields`, `add_fields`, and `add_constraints` methods—described below—for this.

name ()

Returns the name of the structure described by the object.

entry_class ()

Returns the name of the structured entry class associated with this structure.

C.5.7.2 Field structure description methods

add_constraints (TYPE, CONSTRAINT, ...)

Adds one or more field constraints to the structure. A field constraint is specified as a reference to a three-element list; the last element is a reference to the list of fields affected, and the first two elements are the minimum and maximum number of fields from the constraint set allowed in an entry of type TYPE. See section C.5.3.1 for a full explanation of field constraints.

add_fields (TYPE, REQUIRED [, OPTIONAL [, CONSTRAINT, ...]])

Adds fields to the required/optional lists for entries of type TYPE. Can also add field constraints, but you can just as easily use `add_constraints` for that.

REQUIRED and OPTIONAL, if defined, should be references to lists of fields to add to the respective field lists. The CONSTRAINTs, if given, are exactly as described for `add_constraints` above.

set_fields (TYPE, REQUIRED [, OPTIONAL [, CONSTRAINTS, ...]])

Sets the lists of required/optional fields for entries of type TYPE. Identical to `add_fields`, except that the field lists and list of constraints are set from scratch here, rather than being added to.

C.5.7.3 Field structure query methods

types ()

Returns the list of entry types supported by the structure.

known_type (TYPE)

Returns true if TYPE is a supported entry type.

known_field (TYPE, FIELD)

Returns true if FIELD is in the required list, optional list, or one of the constraint sets for entries of type TYPE.

required_fields (TYPE)

Returns the list of required fields for entries of type TYPE.

optional_fields ()

Returns the list of optional fields for entries of type TYPE.

field_constraints ()

Returns the list of field constraints (in the format supplied to `add_constraints`) for entries of type TYPE.

C.5.7.4 Option methods

known_option (OPTION)

Returns false. This is mainly for the use of derived structures that don't have any options, and thus don't need to provide their own `known_option` method. Structures that actually offer options should override this method; it should return true if OPTION is a supported option.

default_option (OPTION)

Crashes with an "unknown option" message. Again, this is mainly for use by derived structure classes that don't actually offer any options. Structures that handle options should override this method; every option handled by `known_option` should have a default value (which might just be `undef`) that is returned by `default_option`. Your `default_options` method should crash on an unknown option, perhaps by calling `SUPER::default_option` (in order to ensure consistent error messages). For example:

```
sub default_option
{
   my ($self, $option) = @_;
   return $default_options{$option}
      if exists $default_options{$option};
   $self->SUPER::default_option ($option);   # crash
}
```

The default value for an option is returned by `get_options` when that options has not been explicitly set with `set_options`.

set_options (OPTION => VALUE, ...)

Sets one or more option values. (You can supply as many `OPTION =$>$ VALUE` pairs as you like, just so long as there are an even number of arguments.) Each OPTION must be handled by the structure module (as indicated by the `known_option` method); if not `set_options` will `croak`. Each VALUE may be any scalar value; it's up to the structure module to validate them.

get_options (OPTION, ...)

Returns the value(s) of one or more options. Any OPTION that has not been set by `set_options` will return its default value, fetched using the `default_value` method. If OPTION is not supported by the structure module, then your program either already crashed (when it tried to set it with `set_option`), or it will crash here (thanks to calling `default_option`).

C.5.8 Methods 2: Base Structured Entry Class

The other class provided by the Structure module is StructuredEntry, the base class for all structured entry classes. This class inherits from Entry, so all of its entry query/manipulation methods are available. StructuredEntry adds methods for checking that an entry conforms to the database structure defined by a structure class.

It only makes sense for StructuredEntry to be used as a base class; you would never create standalone StructuredEntry objects. The superficial reason for this is that only particular structured-entry classes have an actual structure class associated with them, StructuredEntry on its own doesn't have any information about allowed types, required fields, field constraints, and so on. For a deeper understanding, consult section C.5.5 above.

Since StructuredEntry derives from Entry, it naturally operates on BibTeX entries. Hence, the following descriptions refer to "the entry"—this is just the object (entry) being operated on. Note that these methods are presented in bottom-up order, meaning that the methods you're most likely to actually use—`check`, `coerce`, and `silently_coerce` are at the bottom. On a first reading, you'll probably want to skip down to them for a quick summary.

structure ()

Returns the object that defines the structure the entry to which is supposed to conform. This will be an instantiation of some structure class, and exists mainly so the check/coerce methods can query the structure about the types and fields it recognizes. If, for some reason, you wanted to query an entry's structure about the validity of type `foo`, you might do this:

```
# assume $entry is an object of some structured entry class, i.e.
# it inherits from Text::BibTeX::StructuredEntry
$structure = $entry->structure;
$foo_known = $structure->known_type ('foo');
```

check_type ([WARN])

Returns true if the entry has a valid type according to its structure. If WARN is true, then an invalid type results in a warning being printed.

check_required_fields ([WARN [, COERCE]])

Checks that all required fields are present in the entry. If WARN is true, then a warning is printed for every missing field. If COERCE is true, then missing fields are set to the empty string.

This isn't generally used by other code; see the `check` and `coerce` methods below.

check_field_constraints ([WARN [, COERCE]])

Checks that the entry conforms to all of the field constraints imposed by its structure. Recall that a field constraint consists of a list of fields, and a minimum and maximum number of those fields that must be present in an entry. For each constraint, `check_field_constraints` simply counts how many fields in the constraint's field set are present. If this count falls below the minimum or above the maximum for that constraint and WARN is true, a warning is issued. In general, this warning is of the form "between x and y of fields foo, bar, and baz must be present". The more common cases are handled specially to generate more useful and human-friendly warning messages.

If COERCE is true, then the entry is modified to force it into conformance with all field constraints. How this is done depends on whether the violation is a matter of not enough fields present in the entry, or of too many fields present. In the former case, just enough fields are added (as empty strings) to meet the requirements of the constraint; in the latter case, fields are deleted. Which fields to add or delete is controlled by the order of fields in the constraint's field list.

An example should clarify this. For instance, a field constraint specifying that exactly one of `author` or `editor` must appear in an entry would look like this:

```
[1, 1, ['author', 'editor']]
```

Suppose the following entry is parsed and expected to conform to this structure:

```
@inbook{unknown:1997a,
  title = "An Unattributed Book Chapter",
  booktitle = "An Unedited Book",
  publisher = "Foo, Bar \& Company",
  year = 1997
}
```

If `check_field_constraints` is called on this method with COERCE true (which is done by any of the `full_check`, `coerce`, and `silently_coerce` methods), then the `author` field is set to the empty string. (We go through the list of fields in the constraint's field set in order – since `author` is the first missing field, we supply it; with that done, the entry now conforms to the `author`/`editor` constraint, so we're done.)

However, if the same structure was applied to this entry:

```
@inbook{smith:1997a,
  author = "John Smith",
  editor = "Fred Jones",
  ...
}
```

then the `editor` field would be deleted. In this case, we allow the first field in the constraint's field list—`author`. Since only one field from the set may be present, all fields after the first one are in violation, so they are deleted.

Again, this method isn't generally used by other code; rather, it is called by `full_check` and its friends below.

full_check ([WARN [, COERCE]])

Returns true if an entry's type and fields are all valid. That is, it calls `check_type`, `check_required_fields`, and `check_field_constraints`; if all of them return true, then so does `full_check`. WARN and COERCE are simply passed on to the three `check_*` methods: the first controls the printing of warnings, and the second decides whether we should modify the entry to force it into conformance.

check ()

Checks that the entry conforms to the requirements of its associated database structure: the type must be known, all required fields must be present, and all field constraints must be met. See `check_type`, `check_required_fields`, and `check_field_constraints` for details.

Calling `check` is the same as calling `full_check` with WARN true and COERCE false.

coerce ()

Same as `check`, except entries are coerced into conformance with the database structure—that is, it's just like `full_check` with both WARN and COERCE true.

silently_coerce ()

Same as `coerce`, except warnings aren't printed—that is, it's just like `full_check` with WARN false and COERCE true.

C.6 Text::BibTeX::Name

C.6.1 Synopsis

```
$name = new Text::BibTeX::Name;
$name->split('J. Random Hacker');
# or:
$name = new Text::BibTeX::Name ('J. Random Hacker');

@firstname_tokens = $name->part ('first');
$lastname = join (' ', $name->part ('last'));

$format = new Text::BibTeX::NameFormat;
# ...customize $format...
$formatted = $name->format ($format);
```

C.6.2 Description

Text::BibTeX::Name provides an abstraction for BibTeX-style names and some basic operations on them. A name, in the BibTeX world, consists of a list of *tokens* which are divided amongst four *parts*: 'first', 'von', 'last', and 'jr'.

Tokens are separated by whitespace or commas at brace-level zero. Thus the name

```
van der Graaf, Horace Q.
```

has five tokens, whereas the name

```
{Foo, Bar, and Sons}
```

consists of a single token.

How tokens are divided into parts depends on the form of the name. If the name has no commas at brace-level zero (as in the second example), then it is assumed to be in either "first last" or "first von last" form. If there are no tokens that start with a lower-case letter, then "first last" form is assumed: the final token is the last name, and all other tokens form the first name. Otherwise, the earliest contiguous sequence of tokens with initial lower-case letters is taken as the 'von' part; if this sequence includes the final token, then a warning is printed and the final token is forced to be the 'last' part.

If a name has a single comma, then it is assumed to be in "von last, first" form. A leading sequence of tokens with initial lower-case letters, if any, forms the 'von' part; tokens between the 'von' and the comma form the 'last' part; tokens following the comma form the 'first' part. Again, if there are no token following a leading sequence of lowercase tokens, a warning is printed and the token immediately preceding the comma is taken to be the 'last' part.

If a name has more than two commas, a warning is printed and the name is treated as though only the first two commas were present.

Finally, if a name has two commas, it is assumed to be in "von last, jr, first" form. (This is the only way to represent a name with a 'jr' part.) The parsing of the name is the same as for a one-comma name, except that tokens between the two commas are taken to be the 'jr' part.

C.6.3 Examples

The names `'van der Graaf, Horace Q.'` and `'Horace Q. van der Graaf'` split into identical sets of token lists:

```
first => ('Horace', 'Q.')
von   => ('van', 'der')
last  => ('Graaf')
```

with no 'jr' part.

Since `'{Foo, Bar, and Sons}'` consists of a single token with no commas at brace-level zero (if there were any, it would have more than one token!), it falls under the "first last" rule: the whole name becomes the only token in the 'last' part, and no other parts exist.

C.6.4 Methods

new (CLASS [, NAME [, FILENAME, LINE, NAME_NUM]])

Creates a new Text::BibTeX::Name object. If NAME is supplied, it must be a string containing a single name, and it will be be passed to the `split` method for further processing. FILENAME, LINE, and NAME_NUM, if present, are all also passed to `split` to allow better error messages.

split (NAME [, FILENAME, LINE, NAME_NUM])

Splits NAME (a string containing a single name) into tokens and subsequently into the four parts of a BibTeX-style name (first, von, last, and jr). (Each part is a list of tokens, and tokens are separated by whitespace or commas at brace-depth zero. See above for full details on how a name is split into its component parts.)

The token-lists that make up each part of the name are then stored in the Text::BibTeX::Name object for later retrieval or formatting with the `part` and `format` methods.

part (PARTNAME)

Returns the list of tokens in part PARTNAME of a name previously split with `split`. For example, suppose a Text::BibTeX::Name object is created and initialized like this:

```
$name = new Text::BibTeX::Name;
$name->split ('Charles Louis Xavier Joseph de la Vall{\'e}e Poussin');
```

Then this code:

```
$name->part ('von');
```

would return the list `('de','la')`.

format (FORMAT)

Formats a name according to the specifications encoded in FORMAT, which should be a Text::BibTeX::NameFormat (or descendant) object. (In short, it must supply a method `apply` which takes a Text::BibTeX::NameFormat object as its only argument.) Returns the formatted name as a string.

See section C.7 for full details on formatting names.

C.7 Text::BibTeX::NameFormat

C.7.1 Synopsis

```
$format = new Text::BibTeX::NameFormat ($parts, $abbrev_first);

$format->set_text ($part, $where => $text, ...);

$format->set_options ($part, $abbrev, $join_tokens, $join_part

$formatted_name = $format->apply ($name);
```

C.7.2 Description

After splitting a name into its components parts (represented as a Text::BibTeX::Name object), you often want to put it back together again as a single string formatted in a consistent way. Text::BibTeX::NameFormat provides a very flexible way to do this, generally in two stages: first, you create a "name format" which describes how to put the tokens and parts of any name back together, and then you apply the format to a particular name.

The "name format" is encapsulated in a Text::BibTeX::NameFormat object. The `new` constructor includes some clever behind-the-scenes trickery that means you can usually get away with calling it alone, and not need to do any customization of the format. If you do need to customize the format, though, the `set_text()` and `set_options()` methods provide that capability.

Note that Text::BibTeX::NameFormat is a fairly direct translation of the name-formatting C interface in the **btparse** library. This manual page is meant to provide enough information to use the Perl class, but for more details and examples, consult section B.8.

C.7.3 Constants

Two enumerated types for dealing with names and name formatting have been brought from C into Perl. In the **btparse** documentation, you'll see references to `bt_namepart` and `bt_joinmethod`. The former lists the four "parts" of a BibTeX name: first, von, last, and jr; its values (in both C and Perl) are `BTN_FIRST`, `BTN_VON`, `BTN_LAST`, and `BTN_JR`. The latter lists the ways in which `bt_format_name()` (the C function that corresponds to Text::BibTeX::NameFormat's `apply` method) can join adjacent tokens together: `BTJ_MAYTIE`, `BTJ_SPACE`, `BTJ_FORCETIE`, and `BTJ_NOTHING`. Both sets of values may be imported from the Text::BibTeX module, using the import tags `nameparts` and `joinmethods`. For instance:

```
use Text::BibTeX qw(:nameparts :joinmethods);
use Text::BibTeX::Name;
use Text::BibTeX::NameFormat;
```

The "name part" constants are used to specify surrounding text or formatting options on a per-part basis: for instance, you can supply the "pre-token" text, or the "abbreviate" flag, for a single part without affecting other parts. The "join methods" are two of the three formatting options that you can set for a part: you can control how to join the individual tokens of a name (`"JR Smith"`, or `"J R Smith"`, or `"J~R Smith"`, and you can control how the final token of one part is joined to the next part (`"la Roche"` versus `"la~Roche"`).

C.7.4 Methods

new (PARTS, ABBREV_FIRST)

Creates a new name format, with the two most common customizations: which parts to include (and in what order), and whether to abbreviate the first name. PARTS should be a string with at most four characters, one representing each part that you want to occur in a formatted name. For example, `"fvlj"` means to format names in "first von last jr" order, while `"vljf"` denotes "von last jr first." ABBREV_FIRST is just a boolean value: false to print out the first name in full, and true to abbreviate it with periods after each token and discretionary ties between tokens. All intra- and inter-token punctuation and spacing is independently controllable with the `set_text` and `set_options` methods, although these will rarely be necessary—sensible defaults are chosen for everything, based on the PARTS and ABBREV_FIRST values that you supply. See the description of `bt_create_name_format()` in section B.8 for full details of the choices made.

set_text (PART, WHERE, TEXT, ...)

Allows you to customize some or all of the surrounding text for a single name part. WHERE must be one of `"pre_part"`, `"post_part"`, `"pre_token"`, or `"post_token"`, and TEXT is a chunk of text to go in the place described by WHERE. You may have as many (WHERE, TEXT) pairs as you desire, but having more than four would be rather pointless.

For instance, say you want a name format that will abbreviate first names, but without any punctuation after the abbreviated tokens. You could create and customize the format as follows:

```
$format = new Text::BibTeX::NameFormat ("fvlj", 1);
$format->set_text (BTN_FIRST, post_token => "");
```

Without the `set_text()` call, `$format` would result in names formatted like `"J. R. Smith"`. After setting the post-token text for first names to `""`, this name would become `"J R Smith"`.

set_options (PART, ABBREV, JOIN_TOKENS, JOIN_PART)

Allows further customization of a name format: you can set the abbreviation flag and the two token-join methods. Alas, there is no mechanism for leaving a value unchanged; you must set everything with `set_options`.

For example, let's say that just dropping periods from abbreviated tokens in the first name isn't enough; you *really* want to save space by jamming the abbreviated tokens together: `"JR Smith"` rather than `"J R Smith"` Assuming the two calls in the above example have been done, the following will finish the job:

```
$format->set_options (BTN_FIRST,
                      1,              # keep same value for abbrev flag
                      BTJ_NOTHING,    # jam tokens together
                      BTJ_SPACE);     # space after final token of part
```

Note that we unfortunately had to know (and supply) the current values for the abbreviation flag and post-part join method, even though we were only setting the intra-part join method.

apply (NAME)

Once a name format has been created and customized to your heart's content, you can use it to format any number of names using the `apply` method. NAME must be a Text::BibTeX::Name object (i.e., a pre-split name); `apply` returns a string containing the parts of the name formatted according to the Text::BibTeX::NameFormat structure it is called on.

C.7.5 Examples

Although the process of splitting and formatting names may sound complicated and convoluted from reading the above (along with section C.6), it's actually quite simple. There are really only three steps to worry about: split the name (create a Text::BibTeX::Name object), create and customize the format (Text::BibTeX::NameFormat object), and apply the format to the name.

The first step is covered in section C.6; here's a brief example:

```
$orig_name = 'Charles Louis Xavier Joseph de la Vall{\'e}e Poussin';
$name = new Text::BibTeX::Name $orig_name;
```

The various parts of the name can now be accessed through Text::BibTeX::Name methods; for instance `$name-$>$part('von')` returns the list `("de","la")`.

Creating the name format is equally simple:

```
$format = new Text::BibTeX::NameFormat ('vljf', 1);
```

creates a format that will print the name in "von last jr first" order, with the first name abbreviated. And for no extra charge, you get the right punctuation at the right place: a comma before any 'jr' or 'first' tokens, and periods after each 'first' token.

For instance, we can perform no further customization on this format, and apply it immediately to `$name`. There are in fact two ways to do this, depending on whether you prefer to think of it in terms of "Applying the format to a name" or "formatting a name". The first is done with Text::BibTeX::NameFormat's `apply` method:

```
$formatted_name = $format->apply ($name);
```

while the second uses Text::BibTeX::Name's `format` method:

```
$formatted_name = $name->format ($format);
```

which is implemented using of `Text::BibTeX::NameFormat::apply`. In either case, the result with the example name and format shown is

```
de~la Vall{\'e}e~Poussin, C.~L. X.~J.
```

Note the strategic insertion of TeX "ties" (non-breakable spaces) at sensitive spots in the name. (The exact rules for insertion of discretionary ties are given in section B.8.)

C.8 Text::BibTeX::Bib

C.8.1 Description

Text::BibTeX::Bib implements the database structure for bibliographies as defined by the standard styles of BibTeX 0.99. It does this by providing two classes, BibStructure and BibEntry (the leading Text::BibTeX is implied, and will be omitted for the rest of this manual page). These two classes, being specific to bibliographic data, are outside of the core Text::BibTeX class hierarchy, but are distributed along with it as they provide a canonical example of a specific database structure using classes derived from the core hierarchy.

BibStructure, which derives from the Structure class, deals with the structure as a whole: it handles structure options and describes all the types and fields that make up the database structure. If you're interested in writing your own database structure modules, the standard interface for both of these is described in section C.5; if you're just interested in finding out the exact database structure or the options supported by the `Bib` structure, you've come to the right place. (However, you may have to wade through a bit of excess verbiage due to this module's dual purpose: first, to reimplement the standard styles of BibTeX 0.99, and second, to provide an example for other programmers wishing to implement new or derived database structure modules.)

BibEntry derives from the StructuredEntry class and provides methods that operate on individual entries presumed to come from a database conforming to the structure defined by the BibStructure class. (Actually, to be completely accurate, BibEntry inherits from two intermediate classes, BibSort and BibFormat. These two classes just exist to reduce the amount of code in the Bib module, and thanks to the magic of inheritance, their existence is usually irrelevant. But you might want to consult those two classes if you're interested in the gory details of sorting and formatting entries from BibTeX 0.99-style bibliography databases.)

C.8.2 Structure Options

BibStructure handles several user-supplied "structure options" and methods for dealing with them. The options currently supported by the Bib database structure, and the values allowed for them, are:

`sortby` How to sort entries. Valid values: `name` (sort on author names, year, and title), `year` (sort on year, author names, and title). Sorting on "author names" is a bit more complicated than just using the `author` field; see section C.10 for details. Default value: `name`.

`namestyle` How to print author (and editor) names: `full` for unabbreviated first names, `abbrev` for first names abbreviated with periods, `nopunct` for first names abbreviated with space but no periods, and `nospace` to abbreviate without space or periods. Default value: `full`.

`nameorder` The order in which to print names: `first` for "first von last jr" order, and `last` for "von last jr first" order. Default value: `first`.

`atitle_lower` A boolean option: if true, non-book titles will be changed to "sentence capitalization:" words following colons and sentence-ending punctuation will be capitalized, and everything else at brace-depth zero will be changed to lowercase. Default value: true.

`labels` The type of bibliographic labels to generate: `numeric` or `alpha`. (Alphabetic labels are not yet implemented, so this option is currently ignored.) Default value: `numeric`.

Also, several "markup options" are supported. Markup options are distinct because they don't change how text is extracted from the database entries and subsequently mangled; rather, they supply bits of markup that go around the database-derived text. Markup options are always two-element lists: the first to "turn on" some feature of the markup language, and the second to turn it off. For example, if your target language is LaTeX2e and you want journal names emphasized, you would supply a list reference `['\emph{','}']` for the `journal_mkup` option. If you were instead generating HTML, you might supply `['<emph`','¡/emph¿']¿. To keep the structure module general with respect to markup languages, all markup options are empty by default. (Or, rather, they are all references to lists consisting of two empty strings.)

`name_mkup` Markup to add around the list of author names.

`atitle_mkup` Markup to add around non-book (article) titles.

`btitle_mkup` Markup to add around book titles.

`journal_mkup` Markup to add around journal names.

C.8.2.1 Option methods As required by the Text::BibTeX::Structure module, Text::BibTeX::Bib provides two methods for handling options: `known_option` and `default_option`. (The other two option methods, `set_options` and `get_options`, are just inherited from Text::BibTeX::Structure.)

known_option (OPTION)

Returns true if OPTION is one of the options on the above list.

default_option (OPTION)

Returns the default value of OPTION, or `croaks` if OPTION is not a supported option.

C.8.3 Database Structure

The other purpose of a structure class is to provide a method, `describe_entry`, that lists the allowed entry types and the known fields for the structure. Programmers wishing to write their own database structure module should consult section C.5 for the conventions and requirements of this method; the purpose of the present documentation is to describe the `Bib` database structure.

The allowed entry types, and the fields recognized for them, are:

`article` Required fields: `author`, `title`, `journal`, `year`. Optional fields: `volume`, `number`, `pages`, `month`, `note`.

`book` Required fields: `title`, `publisher`, `year`. Optional fields: `series`, `address`, `edition`, `month`, `note`. Constrained fields: exactly one of `author`, `editor`; at most one of `volume`, `number`.

`booklet` Required fields: `title`. Optional fields: `author`, `howpublished`, `address`, `month`, `year`, `note`.

`inbook` Required fields: `publisher`, `year`. Optional fields: `series`, `type`, `address`, `edition`, `month`, `note`. Constrained fields: exactly one of `author`, `editor`; at least one of `chapter`, `pages`; at most one of `volume`, `number`.

`incollection` Required fields: `author`, `title`, `booktitle`, `publisher`, `year`. Optional fields: `editor`, `series`, `type`, `chapter`, `pages`, `address`, `edition`, `month`, `note`. Constrained fields: at most one of `volume`, `number`.

`inproceedings` Required fields: `author`, `title`, `booktitle`, `year`. Optional fields: `editor`, `series`, `pages`, `address`, `month`, `organization`, `publisher`, `note`. Constrained fields: at most one of `volume`, `number`.

`manual` Required fields: `title`. Optional fields: `author`, `organization`, `address`, `edition`, `month`, `year`, `note`.

`mastersthesis` Required fields: `author`, `title`, `school`, `year`. Optional fields: `type`, `address`, `month`, `note`.

`misc` Required fields: none. Optional fields: `author`, `title`, `howpublished`, `month`, `year`, `note`.

`phdthesis` Required fields: `author`, `title`, `school`, `year`. Optional fields: `type`, `address`, `month`, `note`.

`proceedings` Required fields: `title`, `year`. Optional fields: `editor`, `series`, `address`, `month`, `organization`, `publisher`, `note`. Constrained fields: at most one of `volume`, `number`.

`techreport` Required fields: `author`, `title`, `school`, `year`. Optional fields: `type`, `number`, `address`, `month`, `note`.

`unpublished` Required fields: `author`, `title`, `note`. Optional fields: `month`, `year`.

C.8.4 Structured Entry Class

The second class provided by the Text::BibTeX::Bib module is BibEntry (again, a leading Text::BibTeX is implied). This being a structured entry class, it derives from StructuredEntry. The conventions and requirements for such a class are documented in section C.5 for the benefit of programmers implementing their own structure modules.

If you wish to write utilities making use of the `Bib` database structure, then you should call one of the "officially supported" methods provided by the BibEntry class. Currently, there are only two of these: `sort_key` and `format`. These are actually implemented in the BibSort and BibFormat classes, respectively, which are base classes of BibEntry. Thus, see section C.10 and section C.9 for details on these two methods.

C.9 Text::BibTeX::BibFormat

C.9.1 Synopsis

```
# Assuming $entry comes from a database of the 'Bib' structure
# (i.e., that it's blessed into the BibEntry class, which inherits
# the format method from BibFormat):
@blocks = $entry->format;
```

C.9.2 Description

The Text::BibTeX::BibFormat class is a base class of Text::BibTeX::BibEntry for formatting bibliography entries. It thus performs the main job of any program that would hope to supplant BibTeX itself; the other important job (sorting) is handled by its companion class, Text::BibTeX::BibSort.

BibFormat (the Text::BibTeX prefix will be dropped for brevity) pays attention to almost all of the structure options described in section C.8; it only ignores those that cover sorting, currently just `sortby`. In particular, all of the "markup" options control what language is generated by BibFormat; if none of those options are set, then it will generate plain, unmarked text.

The only method in BibFormat's documented interface (so far) is `format`. (The class defines many other methods, but these should not be necessary to outsiders, so they are undocumented and subject to change.)

C.9.3 Methods

format ()

Formats a single entry for inclusion in the bibliography of some document. The exact processing performed is highly dependent on the entry type and the fields present; in general, you should be able to join `format`'s outputs together to create a single paragraph for inclusion in a document of whatever markup language you're working with.

Returns a list of "blocks," which can either be jammed together like sentences (for a traditional "tight" bibliography) or printed on separate lines (for an "open" bibliography format). Each block is a reference to a list of sentences; sentences should be joined together with an intervening period. Each sentence is either a single string or a list of clauses; clauses should be joined together with an intervening comma. Each clause is just a simple string.

See the source code for `btformat` for an example of how to use the output of `format`.

C.10 Text::BibTeX::BibSort

C.10.1 Synopsis

```
# Assuming $entry comes from a database of the 'Bib' structure
# (i.e., that it's blessed into the BibEntry class, which inherits
# the sort_key method from BibSort):
$sort_key = $entry->sort_key;
```

C.10.2 Description

Text::BibTeX::BibSort is a base class of Text::BibTeX::BibEntry for generating sort keys from bibliography entries. It could in principle (and, someday, might) offer a wide range of highly customizable sort-key generators. Currently, though, it provides only a single method (`sort_key`) for public use, and that method only pays attention to one structure option, `sortby`.

C.10.3 Methods

sort_key ()

Generates a sort key for a single bibliographic entry. Assumes this entry conforms to the Bib database structure. The nature of this sort key is controlled by the `sortby` option, which can be either `"name"` or `"year"`. (The `namestyle` also has a role, in determining how author/editor names are formatted for inclusion in the sort key.)

For by-name sorting (which is how BibTeX's standard styles work), the sort key consists of one of the `author`, `editor`, `organization`, or `key` fields (depending on the entry type and which fields are actually present), followed by the year and the title. All fields are drastically simplified to produce the sort key: non-English letters are mercilessly anglicized, non-alphabetic characters are stripped, and everything is forced to lowercase. (The first two steps are done by the `purify_string` routine; see section C.1.7.2 for a brief description, and the descripton of the C function `bt_purify_string()` in section B.9 for all the gory details.)

References

[1] Nelson Beebe. Bibliography prettyprinting and syntax checking. *TUGboat*, 14(4):395–419, December 1993.

[2] Tim Bunce and Andreas König. The Perl 5 Module List. Published electronically via USENET and the World-Wide Web. `http://www.perl.com/CPAN/modules/00modlist.long.html`.

[3] Comprehensive Perl Archive Network (CPAN). FTP archive. Accessed via `http://www.perl.com/CPAN/`.

[4] Comprehensive TEX Archive Network (CTAN). FTP archive. `ftp://ctan.tug.org/tex-archive/`.

[5] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LATEX Companion*, section 13.3.2, pages 386–391. Addison-Wesley, Reading, Massachusetts, 1994. bibunits is available on CTAN [4] in macros/latex209/contrib/misc/.

[6] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LATEX Companion*, section 13.9, pages 419–420. Addison-Wesley, Reading, Massachusetts, 1994. `custom-bib` is available on CTAN [4] in macros/latex/contrib/supported/.

[7] David Hull. bib2html. World-Wide Web site. `http://pertsserver.cs.uiuc.edu/~hull/bib2html/`.

[8] Gerd Neugebauer. BibTool. World-Wide Web site. `http://www.uni-koblenz.de/~gerd/ftp/BibTool/`.

[9] Terence Parr. *Language Translation Using PCCTS & C++*. Automata, 1997.

[10] Oren Patashik, 3 June 1997. Personal communication.

[11] Oren Patashnik. BIBTEX 1.0. *TUGboat*, 15(3):269–273, September 1994.

[12] Randal Schwartz. *Learning Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd edition, 1997.

[13] Sriram Srinivasan. *Advanced Perl Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.

[14] Larry Wall, Tom Christiansen, and Randal Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd edition, 1996.