Bibliography Prettyprinting and Syntax Checking

Nelson H. F. Beebe

Contents

List of Tables

1 Introduction

BibTeX [18, Appendix B] is a convenient tool for solving the vexing issue of bibliography formatting. The user identifies fields of bibliography entries via field/value pairs and provides a unique citation key and a document type for each entry. A simple string substitution facility makes it easy to reuse frequently-occurring strings. A typical example looks like this:

```
@String{pub-AW =
           "Ad{\-d}i{\-s}on-Wes{\-l}ey"}

@Book{Lamport:LDP85,
  author =    "Leslie Lamport",
  title =     "{\LaTeX}---A Document
              Preparation System---User's
```

```
              Guess and Reference Manual",
  publisher = pub-AW,
  year =      "1985",
  ISBN =      "0-201-15790-X",
}
```

The TEX file contains citations of the form `\cite{Lamport:LDP85}`, together with a `\bibliographystyle` command to choose a citation and bibliography style, and a `\bibliography` command to specify which BIBTEX files are to be used. TEX records this information in an auxiliary file.

A subsequent BIBTEX job step reads this auxiliary file, extracts the requested bibliographic entries from the specified BIBTEX files, and outputs the entries into a bibliography file formatted according to the specified style. Several dozen such styles are currently available to help cope with the bizarre variations in bibliography formats that publishers have invented.

In a second TEX step, the `\cite` commands are not correctly expandable until the `\bibliography` command is processed and the bibliography file output by BIBTEX is read. However, at that point, the desired form of the citations is finally known, and at the end of the job, an updated auxiliary file is written.

A third TEX step finally has the necessary information from the auxiliary file and the bibliography file to correctly typeset the `\cite` commands and the bibliography in the specified style.

With the GNU Emacs text editor [7, 27], powerful BIBTEX editing support makes it simple to generate bibliography entry descriptions via templates that can be inserted with a couple of keystrokes, or on workstations, selected from a pop-up menu. This editor is freely available on UNIX, VAX VMS, and the larger members of the IBM PC family under PC-DOS.

The major benefits of using BIBTEX are the potential for data reuse, the separation of form and content (like the descriptive markup of LATEX and SGML[6, 31]), and the many stylistic variants of the typeset bibliography. During the preparation of this article, a scan of our Mathematics Department workstation file system located about 14 000 TEX files, and 445 BIBTEX files. The latter contained about 870 000 lines and almost 94 000 bibliography entries. These files form a valuable resource that authors and researchers can use to track and properly cite literature in their publications.

During my term as TUG President, I initiated a project to collect BIBTEX styles and bibliography data base files of material related to TEX and its uses, and electronic document production and typography in general. This dynamic collection also covers a few journals, including more than 1000 entries for *TUGboat*. A snapshot of part of the collection was published in the 1991 TUG Resource Directory [4, 5].

One drawback of BIBTEX is that errors in a bibliography file, such as unmatched quotation marks around a value string, can sometimes be hard to locate, because the current version of the program raises an error at the end of a scan when internal tables overflow after gobbling several thousand characters of input. The result is that the error location is completely bogus, and actually lies much earlier in the file. We can hope that this serious deficiency will be remedied in the final version of BIBTEX, 1.0, which is expected to appear when the LATEX 3.0 development is completed.

Another drawback is that such bibliography files are normally prepared by human typists, and consequently there are formatting variations that reduce readability, and inconsistencies that persist into the final typeset bibliography. Some examples of such inconsistencies are variations in naming of publishers and journals, spacing around author and editor initials, and variations in letter case in titles. In addition, there are usually numerous typographical errors of omission, doubling, spelling, transcription, translation, and transposition.

In the fall of 1990, faced with a growing collection of BIBTEX files, I set out to write a software tool to deal with these problems. This program is called `bibclean`. It is a syntax checker, portability verifier, and prettyprinter, and was made freely available in 1991. In the fall of 1992, after considerable experience with the first version, I embarked on a set of enhancements that produced major version 2.0, and the purpose of this paper is to describe the new version, and to widely advertise its existence to the TEX community.

2 BIBTEX needs improvement

BIBTEX, like TEX, assumes that its input is prepared correctly, and works best when that is the case. Both programs attempt to recover from errors, but that recovery may be unsuccessful, and errors may be detected only after lengthy processing. In neither case is the output of these programs suitable for input to them. That is, their knowledge of how their input streams are to be parsed is available only to them, and cannot be applied independently and used by other software. Both programs have a hazily-defined input syntax, and TEX's is extensible, making it even harder to give a precise description to the user.

The trend of compiler technology development of the last two decades, largely on UNIX systems, has been to separate the compilation task into several steps.

The first is generally called *lexical analysis*, or lexing. It breaks the input stream up into identifiable tokens that can be represented by small integer constants and constant strings.

The second step is called *parsing*, which involves the verification that the tokens streaming from the lexer conform to the grammatical requirements of the language, that is, that they make sense.

As parsing proceeds, an intermediate representation is prepared that is suitable for the third step, namely, *code generation* or *interpretation*.

This division into subtasks diminishes the complexity of writing a compiler, reduces its memory requirements, and importantly, partitions the job into two parts: a language-dependent, but *architecture-independent*, part consisting of lexing and parsing, and a language-independent, but *architecture-dependent*, part where code is generated or interpreted.

This makes it possible to write a front end for each language, and a back end for each architecture, and by combining them, obtain compilers for all languages and all architectures. The most successful example of this approach at present is almost certainly the Free Software Foundation's GNU Project compilers, which support all common computer architectures with the back ends, and C, C++, and Objective C with the front ends. Additional front ends for several other popular languages are in preparation.

When a lexer is available as a separate program, its output can be conveniently used by other programs for tasks such as database lookup, floating-point precision conversion, language translation, linguistic analysis, portability verification, prettyprinting, and checking of grammar, syntax, and spelling.

In response to a command-line request, `bibclean` will function as a lexer instead of as a prettyprinter. An example is given later in Section 6.

3 Run-time options

On several operating systems, `bibclean` is run by a command of the form

```
bibclean [options] bibfile(s) >newfile
```

One or more bibliography files can be specified; if none are given, input is taken from the standard input stream. A specific example is:

```
bibclean -no-fix-name mybib.bib >mybib.new
```

Command-line switches may be abbreviated to a unique leading prefix, and letter case is not significant. All options are parsed before any input bibliography files are read, no matter what their order on the command line. Options that correspond to a *yes/no* setting of a flag have a form with a prefix `no-` to set the flag to *no*. For such options, the last setting determines the flag value used. This is significant when options are also specified in initialization files (see Section 5).

On VAX VMS and IBM PC-DOS, the leading hyphen on option names may be replaced by a slash; however, the hyphen option prefix is always recognized.

`-author` Display an author credit on the standard error unit, `stderr`. Sometimes an executable program is separated from its documentation and source code; this option provides a way to recover from that.

`-error-log filename` Redirect `stderr` to the indicated file, which will then contain all of the error and warning messages. This option is provided for those systems that have difficulty redirecting `stderr`.

`-help` or `-?` Display a help message on `stderr`, giving a sample command usage, and option descriptions similar to the ones here.

`-init-file filename` Provide an explicit value pattern initialization file. It will be processed after any system-wide and job-wide initialization files found on the `PATH` (for VAX VMS, `SYS$SYSTEM`) and `BIBINPUTS` search paths, respectively, and may override them. It in turn may be overridden by a subsequent file-specific initialization file. The initialization file name can be changed at compile time, or at run time through a setting of the environment variable `BIBCLEANINI`, but defaults to `.bibcleanrc` on UNIX, and to `bibclean.ini` elsewhere. For further details, see Section 5.

`-max-width nnn` Normally, `bibclean` limits output line widths to 72 characters, and in the interests of consistency, that value should not be changed. Occasionally, special-purpose applications may require different maximum line widths, so this option provides that capability. The number following the option name can be specified in decimal, octal (starting with 0), or hexadecimal (starting with 0x). A zero or negative value is interpreted to mean unlimited, so `-max-width 0` can be used to ensure that each field/value pair appears on a single line.

When `-no-prettyprint` requests `bibclean` to act as a lexical analyzer, the default line width is unlimited, unless overridden by this option.

When `bibclean` is prettyprinting, line wrapping will be done only at a space. Consequently, an extremely long non-blank character sequence may result in the output exceeding the requested line width. Such sequences are extremely unlikely to occur, at least in English-language text, since even the 45-letter giant [16, p. 451] *pneumonoultramicroscopicsilicovolcanoconiosis* will fit in `bibclean`'s standard 72-character output line, and so will 58-letter Welsh city names.

When `bibclean` is lexing, line wrapping is done by inserting a backslash-newline pair when the specified maximum is reached, so no line length will ever exceed the maximum.

`-[no-]check-values` With the positive form, apply heuristic pattern matching to field values in order to detect possible errors (e.g. `year = "192"` instead of `year = "1992"`), and issue warnings when unexpected patterns are found.

This checking is usually beneficial, but if it produces too many bogus warnings for a particular bibliography file, you can disable it with the negative form of this option. Default: *yes.*

`-[no-]delete-empty-values` With the positive form, remove all field/value pairs for which the value is an empty string. This is helpful in cleaning up bibliographies generated from text editor templates. Compare this option with `-[no-]remove-OPT-prefixes` described below. Default: *no.*

`-[no-]file-position` With the positive form, give detailed file position information in warning and error messages. Default: *no.*

`-[no-]fix-font-changes` With the positive form, supply an additional brace level around font changes in titles to protect against downcasing by some BibTeX styles. Font changes that already have more than one level of braces are not modified.

For example, if a title contains the Latin phrase `{\em Dictyostelium Discoideum}` or `{\em {D}ictyostelium {D}iscoideum}`, then downcasing will incorrectly convert the phrase to lower-case letters. Most BibTeX users are surprised that bracing the initial letters does not prevent the downcase action. The correct coding is `{{\em Dictyostelium Discoideum}}`. However, there are also legitimate cases where an extra level of bracing wrongly protects from downcasing. Consequently, `bibclean` will normally not supply an extra level of braces, but if you have a bibliography where the extra braces are routinely missing, you can use this option to supply them.

If you think that you need this option, it is strongly recommended that you apply `bibclean` to your bibliography file with and without `-fix-font-changes`, then compare the two output files to ensure that extra braces are not being supplied in titles where they should not be present. You will have to decide which of the two output files is the better choice, then repair the incorrect title bracing by hand.

Since font changes in titles are uncommon, except for cases of the type which this option is designed to correct, it should do more good than harm. Default: *no.*

`-[no-]fix-initials` With the positive form, insert a space after a period following author initials. Default: *yes.*

`-[no-]fix-names` With the positive form, reorder author and editor name lists to remove commas at brace level zero, placing first names or initials before last names. Default: *yes.*

`-[no-]par-breaks` With the negative form, a paragraph break (either a formfeed, or a line containing only spaces) is not permitted in value strings, or between field/value pairs. This may be useful to quickly trap runaway strings arising from mismatched delimiters. Default: *yes.*

`-[no-]prettyprint` Normally, `bibclean` functions as a prettyprinter. However, with the negative form of this option, it acts as a lexical analyzer instead, producing a stream of lexical tokens. See Section 6 for further details. Default: *yes.*

`-[no-]print-patterns` With the positive form, print the value patterns read from initialization files as they are added to internal tables. Use this option to check newly-added patterns, or to see what patterns are being used.

When `bibclean` is compiled with native pattern-matching code (the default), these patterns are the ones that will be used in checking value strings for valid syntax, and all of them are specified in initialization files, rather than hard-coded into the program. For further details, see Section 5. Default: *no.*

`-[no-]read-init-files` With the negative form, suppress loading of system-, user-, and file-specific initialization files. Initializations will come only from those files explicitly given by `-init-file filename` options. Default: *yes.*

`-[no-]remove-OPT-prefixes` With the positive form, remove the `OPT` prefix from each field name where the corresponding value is not an empty string. The prefix `OPT` must be entirely in upper-case to be recognized.

This option is for bibliographies generated with the help of the GNU Emacs BibTeX editing support, which generates templates with optional fields identified by the `OPT` prefix. Although the function `M-x bibtex-remove-OPT` normally bound to the keystrokes `C-c C-o` does the job, users often forget, with the result that BibTeX does not recognize the field name, and ignores the value string. Compare this option with `-[no-]delete-empty-values` described above. Default: *no*.

`-[no-]scribe` With the positive form, accept input syntax conforming to the Scribe document system. The output will be converted to conform to BibTeX syntax. See Section 8 for further details. Default: *no*.

`-[no-]trace-file-opening` With the positive form, record in the error log file the names of all files which `bibclean` attempts to open. Use this option to identify where initialization files are located. Default: *no*.

`-[no-]warnings` With the positive form, allow all warning messages. The negative form is not recommended since it may mask problems that should be repaired. Default: *yes*.

`-version` Display the program version number on `stderr`. This will also include an indication of who compiled the program, the host name on which it was compiled, the time of compilation, and the type of string-value matching code selected, when that information is available to the compiler.

4 Prettyprinting

A prettyprinter for any language must be able to deal with more than just those files that strictly conform to the language grammar. For programming languages, most compilers implement language extensions that prettyprinters must recognize and try to deal with gracefully. `bibclean` recognizes two such input languages: BibTeX and Scribe.

Ideally, a prettyprinter should be able to produce output even in the presence of input errors, displaying it in such a way as to make the location of the errors more evident. `bibclean` provides detailed error and warning messages to help pinpoint errors. With the `-file-position` command-line option, it will flag the byte, column, and line, positions of the start and end of the current token in both input and output files.

Here is a summary of the actions taken by `bibclean` on its input stream.

- Space between entries is discarded, and replaced by a single blank line.
- Space around string concatenation operators is standardized.
- Leading and trailing space in value strings is discarded, and embedded multiple spaces are collapsed to a single space.
- String lengths are tested against the limit in standard BibTeX, and warnings issued when the limit is exceeded. The standard limit has proven to be too small in practice, and many sites install enlarged versions of BibTeX. Perhaps BibTeX version 1.0 will use more realistic values, or eliminate string length limits altogether.
- Outer parentheses in entries are standardized to braces.
- Braced value strings are standardized to quoted value strings.
- Field/value pairs are output on separate lines, wrapping long lines to not exceed a user-definable standard width whenever possible.
- A trailing comma is supplied after the last field/value assignment. This is convenient if assignments are later reordered during editing.
- `-fix-font-changes` provides for protecting value string text inside font changes from downcasing.
- Brace-level zero upper-case acronyms in titles are braced to protect from downcasing.
- `-no-par-breaks` provides a way to check for blank lines in string values, which may be indicative of unclosed delimiter errors.
- Umlaut accents, `\"x`, inside value strings at brace-level zero are converted to `{\"x}`. This has been found to be a common user error. BibTeX requires embedded quotes to be nested inside braces.
- Letter-case usage in entry and field names is standardized, so for example, `mastersthesis` and `MASTERSTHESIS` become `MastersThesis`.
- ISBNand ISSN checksums are validated. BibTeX style files that recognize field names for them are available in the TUG bibliography collection, and the bibliography for this document uses them.

- Name modifiers like Jr, Sr, etc. are recognized and handled by `-fix-names`, and names are put into a standard order, so that `Bach, P. D. Q.` becomes `P. D. Q. Bach`.
- With `-fix-initials`, uniform spacing is supplied after brace-level zero initials in personal names.
- With `-check-values`, citation key and field values are matched against patterns to catch irregularities and possible errors.
- Dates of the month, like `"July 14"`, are converted to use month abbreviations, `jul # " 14"`.
- Page number ranges are converted to use en-dashes, instead of hyphens or em-dashes.
- With `-check-values`, year numbers are checked against patterns, then if no match is found, the year values are checked against reasonable limits.
- With `-trace-file-opening`, file open attempts are logged. This helps in the diagnosis of problems such as missing files, or incorrect file permissions.
- On lexing or parsing errors, `bibclean` attempts to resynchronize by flushing the input until it finds the next line containing an initial `@` character preceded by nothing other than optional white space.
- When an `@` character begins a line, a new bibliography entry is assumed to have started. The current brace balance is then tested to make sure it is zero. A non-zero brace level is strongly suggestive of an error, so `bibclean` issues an error message, and zeros the brace level.
- At end-of-file, the brace level is tested. A non-zero brace level is very likely an error, and occasions an error message.

5 Pattern matching and initialization files

`bibclean` can be compiled with one of three different types of pattern matching; the choice is made by the installer at compile time:

- The original version uses explicit hand-coded tests of value-string syntax.
- The second version uses regular-expression pattern-matching host library routines together with regular-expression patterns that come entirely from initialization files.
- The third version uses special patterns that come entirely from initialization files.

The second and third versions are the ones of most interest here, because they allow the user to control what values are considered acceptable. However, command-line options can also be specified in initialization files, no matter which pattern-matching choice was selected.

When `bibclean` starts, it searches for initialization files, finding the first one in the system executable program search path (on UNIX and IBM PC-DOS, `PATH`) and the first one in the `BIBINPUTS` search path, and processes them in turn. Then, when command-line arguments are processed, any additional files specified by `-init-file filename` options are also processed. Finally, immediately before each named bibliography file is processed, an attempt is made to process an initialization file with the same name, but with the extension changed to `.ini`. The default extension can be changed by a setting of the environment variable `BIBCLEANEXT`. This scheme permits system-wide, user-wide, session-wide, and file-specific initialization files to be supported.

When input is taken from `stdin`, there is no file-specific initialization.

For precise control, the `-no-init-files` option suppresses all initialization files except those explicitly named by `-init-file filename` options, either on the command line, or in requested initialization files.

Recursive execution of initialization files with nested `-init-file filename` options is permitted; if the recursion is circular, `bibclean` will finally get a non-fatal initialization file open failure after opening too many files. This terminates further initialization file processing. As the recursion unwinds, the files are all closed, then execution proceeds normally.

An initialization file may contain empty lines, comments from percent to end of line (just like TeX), option switches, and field/pattern or field/pattern/message assignments. Leading and trailing spaces are ignored. This is best illustrated by the short example in Table 1. Long logical lines can be split into multiple physical lines by breaking at a backslash-newline pair; the backslash-newline pair is discarded. This processing happens while characters are being read, before any further interpretation of the input stream.

Each logical line must contain a complete option (and its value, if any), or a complete field/pattern pair, or a field/pattern/message triple.

Comments are stripped during the parsing of the field, pattern, and message values. The comment start symbol is not recognized inside quoted strings, so it can be freely used in such strings.

Comments on logical lines that were input as multiple physical lines via the backslash-newline

Table 1: Sample bibclean initialization file.

```
%% Start with our departmental patterns
-init-file /u/math/bib/.bibcleanrc

%% Make some small additions
chapter = "\"D\""           %% 23

pages   = "\"D--D\""        %% 23--27

volume  = "\"D \\an\\d D\"" %% 11 and 12

year    = \
   "\"dddd, dddd, dddd\"" \
   "Multiple years specified."
                     %% 1989, 1990, 1991

-no-fix-names %% do not modify
              %% author/editor lists
```

convention must appear on the last physical line; otherwise, the remaining physical lines will become part of the comment.

Pattern strings must be enclosed in quotation marks; within such strings, a backslash starts an escape mechanism that is commonly used in UNIX software. The recognized escape sequences are given in Table 2. Backslash followed by any other character produces just that character. Thus, `\"` produces a quotation mark, and `\\` produces a single backslash.

Table 2: Escape sequences in quoted strings.

`\a`	alarm bell (octal 007)
`\b`	backspace (octal 010)
`\f`	formfeed (octal 014)
`\n`	newline (octal 012)
`\r`	carriage return (octal 015)
`\t`	horizontal tab (octal 011)
`\v`	vertical tab (octal 013)
`\ooo`	character number octal `ooo` (e.g. `\012` is linefeed). Up to 3 octal digits may be used.
`\0xhh`	character number hexadecimal `hh` (e.g. `\0x0a` is linefeed). `xhh` may be in either letter case. Any number of hexadecimal digits may be used.

An ASCII NUL (`\0`) in a string will terminate it; this is a feature of the C programming language in which `bibclean` is implemented.

Field/pattern pairs can be separated by arbitrary space, and optionally, either an equals sign or colon functioning as an assignment operator. Thus, the following are equivalent:

```
pages="\"D--D\""
pages:"\"D--D\""
pages "\"D--D\""
  pages = "\"D--D\""
  pages : "\"D--D\""
pages   "\"D--D\""
```

Each field name can have an arbitrary number of patterns associated with it; however, they must be specified in separate field/pattern assignments.

An empty pattern string causes previously-loaded patterns for that field name to be forgotten. This feature permits an initialization file to completely discard patterns from earlier initialization files.

Patterns for value strings are represented in a tiny special-purpose language that is both convenient and suitable for bibliography value-string syntax checking. While not as powerful as the language of regular-expression patterns, its parsing can be portably implemented in less than 3% of the code in a widely-used regular-expression parser (the GNU `regexp` package).

The patterns are represented by the special characters given in Table 3.

The `X` pattern character is very powerful, but generally inadvisable, since it will match almost anything likely to be found in a BibTeX value string. The reason for providing pattern matching on the value strings is to uncover possible errors, not mask them.

There is no provision for specifying ranges or repetitions of characters, but this can usually be done with separate patterns. It is a good idea to accompany the pattern with a comment showing the kind of thing it is expected to match. Here is a portion of an initialization file giving a few of the patterns used to match `number` value strings:

```
number = "\"D\""          %% 23
number = "\"A AD\""       %% PN LPS5001
number = "\"A D(D)\""     %% RJ 34(49)
number = "\"A D\""        %% XNSS 288811
number = "\"A D\\.D\""    %% Version 3.20
number = "\"A-A-D-D\""    %% UMIAC-TR-89-11
number = "\"A-A-D\""      %% CS-TR-2189
number = "\"A-A-D\\.D\"" %% CS-TR-21.7
```

For a bibliography that contains only `Article` entries, this list should probably be reduced to just the first pattern, so that anything other than a digit string fails the pattern-match test. This is easily

Table 3: Initialization file pattern characters.

`␣`	one or more spaces
`a`	exactly one letter
`A`	one or more letters
`d`	exactly one digit
`D`	one or more digits
`r`	exactly one Roman numeral
`R`	one or more Roman numerals (i.e. a Roman number)
`w`	exactly one word (one or more letters and digits)
`W`	one or more words, separated by space, beginning and ending with a word
`.`	one 'special' character, one of the characters `␣!#()*+,-./:;?[]~`, a subset of punctuation characters that are typically used in string values
`:`	one or more 'special' characters
`X`	one or more 'special'-separated words, beginning and ending with a word
`\x`	exactly one `x` (`x` is any character), possibly with an escape sequence interpretation given earlier
`x`	exactly the character `x` (`x` is anything but one of these pattern characters: `aAdDrRwW.:␣\`)

done by keeping bibliography-specific patterns in a corresponding file with extension `.ini`, since that file is read automatically. You should be sure to use empty pattern strings in this pattern file to discard patterns from earlier initialization files.

The value strings passed to the pattern matcher contain surrounding quotes, so the patterns should also. However, you could use a pattern specification like `"\"D"` to match an initial digit string followed by anything else; the omission of the final quotation mark `\"` in the pattern allows the match to succeed without checking that the next character in the value string is a quotation mark.

Because the value strings are intended to be processed by TeX, the pattern matching ignores braces, and TeX control sequences, together with any space following those control sequences. Spaces around braces are preserved. This convention allows the pattern fragment `A-AD-D` to match the value string `TN-K\slash 27-70`, because the value is implicitly collapsed to `TN-K27-70` during the matching operation.

`bibclean`'s normal action when a string value fails to match any of the corresponding patterns is to issue a warning message similar to this: `Unexpected value in ``year = "192"''`. In most cases, that is sufficient to alert the user to a problem. In some cases, however, it may be desirable to associate a different message with a particular pattern. This can be done by supplying a message string following the pattern string. Format items `%%` (single percent), `%e` (entry name), `%f` (field name), `%k` (citation key), and `%v` (string value) are available to get current values expanded in the messages. Here is an example:

```
chapter = "\"D:D\"" \
     "Colon found in ``%f = %v''" %% 23:2
```

To be consistent with other messages output by `bibclean`, the message string should not end with punctuation.

If you wish to make the message an error, rather than just a warning, begin it with a query (?), like this:

```
chapter = "\"D:D\"" \
    "?Colon found in ``%f = %v''" %% 23:2
```

The query will *not* be included in the output message.

Escape sequences are supported in message strings, just as they are in pattern strings. You can use this to advantage for fancy things, such as terminal display mode control. If you rewrite the previous example as

```
chapter = "\"D:D\"" \
          "?\033[7mColon found \
in ``%f = %v''\033[0m" %% 23:2
```

the error message will appear in inverse video on display screens that support ANSI terminal control sequences. Such practice is not normally recommended, since it may have undesirable effects on some output devices. Nevertheless, you may find it useful for restricted applications.

For some types of bibliography fields, `bibclean` contains special-purpose code to supplement or replace the pattern matching:

- `ISBN` and `ISSN` field values are handled this way because their validation requires evaluation of checksums that cannot be expressed by simple patterns; no patterns are even used in these two cases.
- When `bibclean` is compiled with pattern-matching code support, `chapter`, `number`, `pages`, and `volume` values are checked only by pattern matching.
- `month` values are first checked against the standard BibTeX month name abbreviations, and only if no match is found are patterns then used.

- year values are first checked against patterns, then if no match is found, the year numbers are found and converted to integer values for testing against reasonable bounds.

Values for other fields are checked only against patterns. You can provide patterns for any field you like, even ones bibclean does not already know about. New ones are simply added to an internal table that is searched for each string to be validated.

The special field, key, represents the bibliographic citation key. It can be given patterns, like any other field. Here is an initialization file pattern assignment that will match an author name, a colon, an alphabetic string, and a two-digit year:

```
key = "A:Add"   %% Knuth:TB86
```

Notice that no quotation marks are included in the pattern, because the citation keys are not quoted. You can use such patterns to help enforce uniform naming conventions for citation keys, which is increasingly important as your bibliography data base grows.

6 Lexical analysis

The command-line option -no-prettyprint requests bibclean to function as a lexical analyzer instead of as a prettyprinter. Its output is then a stream of lines, each of which contains one token. For the bibliography entries shown in Section 1, here is what the output looks like; the long lines have been wrapped by a backslash-newline to fit in these narrow journal columns:

```
# line 1 "stdin"
2       AT      "@"
18      STRING  "String"
11      LBRACE  "{"
1       ABBREV  "pub-AW"
6       EQUALS  "="
# line 2 "stdin"
19      VALUE   "\"Ad{\\-d}i{\\-s}on-Wes{\
\\-l}ey\""
15      RBRACE  "}"
# line 4 "stdin"
13      NEWLINE "\n"
13      NEWLINE "\n"
2       AT      "@"
5       ENTRY   "Book"
11      LBRACE  "{"
10      KEY     "Lamport:LDP85"
3       COMMA   ","
13      NEWLINE "\n"
# line 5 "stdin"
7       FIELD   "author"
6       EQUALS  "="
19      VALUE   "\"Leslie Lamport\""
3       COMMA   ","
13      NEWLINE "\n"
# line 6 "stdin"
7       FIELD   "title"
6       EQUALS  "="
# line 8 "stdin"
19      VALUE   "\"{\\LaTeX}---{A} Docume\
nt Preparation System---User's Guide and \
Reference Manual\""
3       COMMA   ","
13      NEWLINE "\n"
# line 9 "stdin"
7       FIELD   "publisher"
6       EQUALS  "="
1       ABBREV  "pub-AW"
3       COMMA   ","
13      NEWLINE "\n"
# line 10 "stdin"
7       FIELD   "year"
6       EQUALS  "="
19      VALUE   "\"1985\""
3       COMMA   ","
13      NEWLINE "\n"
# line 11 "stdin"
7       FIELD   "ISBN"
6       EQUALS  "="
19      VALUE   "\"0-201-15790-X\""
3       COMMA   ","
13      NEWLINE "\n"
# line 12 "stdin"
15      RBRACE  "}"
# line 13 "stdin"
13      NEWLINE "\n"
```

Each line begins with a small integer token type number for the convenience of computer programs, then a token type name for human readers, followed by a quoted token string.

Lines beginning with a sharp, #, are ANSI/ISO Standard C preprocessor line-number directives [3, Section 3.8.4] to record the input line number and file name.

There are currently 19 token types defined in the documentation that accompanies bibclean. Because BibTeX styles can define new field names, there is little point in the lexical analyzer of attempting to classify field names more precisely; that job is left for other software.

Inside quoted strings, the ANSI/ISO Standard C [3, Section 3.1.3.4] backslash escape sequences shown in Table 2 on page 401 are used to encode non-printable characters. In this way, a multi-line string value can be represented on a single line. This is convenient for string-searching applications. If the long output lines prove a problem on some systems, the `-max-width nnn` command-line option can be used to wrap lines at a specified column number by the insertion of a backslash-newline pair.

As a simple example of how this token stream might be processed, the UNIX command pipeline

```
bibclean -no-prettyprint mylib.bib | \
    awk '$2 == "KEY" {print $3}' | \
    sed -e 's/"//g' | \
    sort
```

will extract a sorted list of all citation keys in the file `mylib.bib`.

As a more complex example, consider locating duplicate abbreviations and citation keys in a large collection of bibliography files. This is a daunting task if it must be done by visual scanning of the files. It took me less than 10 minutes to write and debug a 35-line `nawk` [1] program (15 lines of comments, 20 of code) that processed the token stream from `bibclean` and printed warnings about such duplicates.

The processing steps can be represented by the simple UNIX pipeline

```
bibclean -no-prettyprint bibfiles | \
    tr '[A-Z]' '[a-z]' | \
    nawk -f bibdup.awk
```

which is most conveniently encapsulated in a command script so that it can be invoked more simply as

```
bibdup *.bib
```

to produce output like this:

```
Duplicate string abbreviation ["pub-aw"]:
        # line 1 "ll.bib"
        # line 141 "master.bib"
Duplicate key ["lamport:ldp85"]:
        # line 4 "ll.bib"
        # line 4172 "master.bib"
...
```

BibTeX's grammar is somewhat hazy, so it is not easy to perform a lexical analysis without some context sensitivity. `bibclean` therefore produces the lexical token stream merely as an alternate output format. In particular, this means that any requested run-time formatting options will have been applied to the tokens *before* they are output to the lexical token stream. For example, a Scribe bibliography file can be converted to a BibTeX token stream so that software that processes `bibclean`'s output need not be Scribe-aware.

7 Portability

`bibclean` is written in ANSI/ISO Standard C [3] with great care taken to produce maximum portability. It has been successfully tested with more than 30 different compilers on all major workstation, and one mainframe, UNIX systems, plus VAX VMS, PC-DOS, OS/2, and Atari TOS.

The C programming language has become the language of choice today for most personal computer and UNIX software development, and the increasing availability of C implementations conforming to the 1989 Standard [3] makes it easier to write code that will compile and run without modification on a wide variety of systems.

C does not have Pascal's problems with character strings and dynamic memory allocation that forced Don Knuth to implement the `WEB` string pool feature and to use compile-time array allocation in the TeX software development. C's rich operator syntax, its powerful run-time library, and generally excellent operating-system interfaces have made it widely popular. More than a million copies of the first edition of *The C Programming Language* book [13] have been sold, and the second edition [14] may do even better.

Nevertheless, C has some serious problems. Philippe Kahn, the founder of Borland International, has called C a *write-only* language. Two books have been written about its syntactical peculiarities [9, 17], and one of them has already appeared in a second edition.

The only way to overcome these problems is meticulous care in programming, and experience with as many compilers and computer architectures as possible. Several books offer valuable advice on C portability [10, 11, 19, 23, 24, 26, 29].

C++ [8, 30] is an extension of C to support object-oriented programming, and has an enthusiastic following. ANSI/ISO standardization efforts are in progress, sadly while the language is still evolving.

From the point of view of a C programmer, the advantage of C++ over C is its much stricter checking of type conversions and intermodule interfaces. `bibclean` has been carefully written to be compilable under C++ as well as C, and to date, has been tested with more than a dozen C++ and Objective C (another C superset) compilers.

All of the extra features of the C++ language are strictly avoided, because using them would seriously limit `bibclean`'s portability. Not only is the syntax of the C++ language under evolution, but the C++ class libraries are for the most part *completely dependent* on the particular implementation. Microsoft's 1020-page documentation of its C++ class library is 10% larger than that of its C run-time library.

Nevertheless, I *strongly recommend* use of C++ compilers in preference to C compilers, so as to catch bugs at compile time that would otherwise not be found until post-mortem dump time, or when the code is ported to a new architecture.

8 Scribe bibliography format

The Scribe document formatting system [25] greatly influenced LaTeX and BibTeX, as well as the GNU Emacs TeXinfo system.

With care, it is possible to share bibliography files between Scribe and BibTeX. Nevertheless, there are some differences, so here is a summary of features of the Scribe bibliography file format. We record them because they are difficult to determine from the published manual, and because readers may sometimes acquire files in this format without having prior exposure to Scribe.

1. Letter case is not significant in field names and entry names, but case is preserved in value strings.
2. In field/value pairs, the field and value may be separated by one of three characters: `=`, `/`, or `␣` (space). Space may optionally surround these separators.
3. Value delimiters are any of these seven pairs: `{ }`, `[ ]`, `( )`, `< >`, `' '`, `" "`, and `` ` ` ``.
4. Value delimiters may not be nested, even though with the first four delimiter pairs, nested balanced delimiters would be unambiguous.
5. Delimiters can be omitted around values that contain only letters, digits, sharp (`#`), ampersand (`&`), period (`.`), and percent (`%`).
6. Outside of delimited values, a literal at-sign (`@`) is represented by doubled at-signs (`@@`).
7. Bibliography entries begin with `@name`, as for BibTeX, but any of the seven Scribe value delimiter pairs may be used to surround the values in field/value pairs. As in (4), nested delimiters are forbidden.
8. Arbitrary space may separate entry names from the following delimiters.
9. `@Comment` is a special command whose delimited value is discarded. As in (4), nested delimiters are forbidden.
10. The special form

    ```
    @Begin{comment}
    ...
    @End{comment}
    ```

    permits encapsulating arbitrary text containing any characters or delimiters, other than `@End{comment}`. Any of the seven delimiter pairs may be used around the word `comment` following the `@Begin` or `@End`; the delimiters in the two cases need not be the same, and consequently, `@Begin{comment}`/`@End{comment}` pairs may not be nested.
11. The `key` field is required in each bibliography entry.
12. A backslashed quote in a string will be assumed to be a TeX accent, and braced appropriately. While such accents do not conform to Scribe syntax, Scribe-format bibliographies have been found that appear to be intended for TeX processing.

Because of this loose syntax, `bibclean`'s normal error detection heuristics are less effective, and consequently, Scribe mode input is not the default; it must be explicitly requested.

9 Recommendations for BibTeX design

The documentation available for BibTeX leaves several points about the input syntax unclear, and I had to obtain answers to the following questions by experiment:

- Can an at-sign occur inside a `@Comment{...}`? *No.*
- Can string abbreviation names be used on the right-hand side of string definitions? *Yes.*
- Can the argument of `@String` be empty? *No.*
- Can a citation key be omitted in an entry? *No.*
- Can the list of assignments in an entry be empty? *Yes.*
- Can a `@Comment{...}` occur between arbitrary tokens? *No.*
- Are newlines preserved in the argument of a `@Preamble{...}`? The answer is relevant if the user includes TeX comments in the preamble material. *No.*

I view the experimental answers to these questions as pure happenstance, and could reasonably argue for the opposite answers to the ones obtained.

Grammar

The most important recommendation that I can make for the next version of BibTeX is that it *must* have a rigorous grammar, including a well-defined comment syntax.

The grammar can almost be of the simple class LL(0) [2], requiring no lookahead during parsing, and one-character lookahead during lexical analysis. However, the presence of the string concatenation operator complicates things sufficiently to require at least an LL(1) grammar.

Such grammars are straightforward to handle with either hand-coded parsers, or with parsers automatically generated from grammar files by compiler development tools like the UNIX `lex` [20] and `yacc` [12, 21, 22, 28] programs, or the Free Software Foundation equivalents, `flex` and `bison`.

`yacc` and `bison` implement LALR(1) parsers; the acronym stands for "Look-Ahead at most 1 token with a Left-to-Right derivation". These are simpler than the LR(k) grammars introduced by none other than the author of TEX in the fundamental paper on the theory of parsing [15]. Nevertheless, they are sufficient for a broad class of language grammars, including most major programming languages, and importantly, they produce compact, efficient, fast, and reliable parsers. LL(1) grammars are a special case of LALR(1) grammars, and we will later define a BIBTEX grammar in LALR(1) form in Section 11.

Comment syntax

The comment syntax should preferably be identical to that of TEX, so that a comment runs from percent to end-of-line, and then *additionally gobbles all leading horizontal space on the next line, up to, but not including, its end-of-line*. This permits breaking of long lines without having to destroy indentation that is so necessary for readability. Percent-initiated comments are already supported in BIBTEX style files, though such comments end after the first following newline.

For SCRIBE compatibility, BIBTEX should also support a `@Comment{...}` entry type. This will require additions to *all* BIBTEX style files, since the entry types are known there, and not in the BIBTEX code itself. BIBTEX 0.99c already knows about `@Comment{...}`, but the `WEB` code section "Process a `comment` command" will have to be extended to deal with the grammar changes.

It is important that BIBTEX not discard `@Comment{...}` entries, because it would then not be possible to write a BIBTEX style file that converted a bibliography file to another format without loss of information. One such style already exists to convert BIBTEX files to UNIX `bib/refer` format.

Characters in names

The characters that can appear in key, entry, and field names *must* be defined by enumeration, rather than by exclusion, as is currently done [18, Section B.1.3]. The reason is that character sets vary between computers, and the new, and very much larger, ISO10646M character set may be widely available in this decade. These variations make the set of admissible name characters vary between systems, compromising portability. I strongly recommend following the conventions for identifiers in widely-used programming languages to define the grammar of key, entry, and field names. It seems to me that letters, digits, colon, hyphen, and possibly plus and slash, should be adequate, and names should be required to begin with a letter. 'Letter' here should include *only* the 26 Roman letters 'A' through 'Z', because allowing letters from other alphabets opens a horrid can of worms that will seriously impact portability of bibliography files until the computer world has a single uniform character set.

I tested this set of characters against 92 500 entries in local bibliography files, and found only a few keys that used other characters: the new ones were period and apostrophe (e.g. O'Malley:TB92). They might therefore be permitted as well, though I would prefer to omit them, and retrofit changes in a few citation keys.

The characters permitted in citation keys should be the same as those in entry and field names, so as to avoid user confusion.

Error reporting

When BIBTEX begins to collect a token, it should record the current line number. When an unclosed string later causes internal buffer overflow, it could report something like `String buffer overflow on input lines 24--82` that would better help locate the offending string by giving its starting and ending line numbers.

To simplify error recovery in such cases, BIBTEX could additionally require that the `@` character that starts a new entry must be the first non-space character on a line.

File inclusion

BIBTEX sorely needs a file inclusion facility. With BIBTEX 0.99c, this feature is available in a crude fashion by listing several files in the `\bibliography` command. However, this is not sufficiently general, and requires unnecessary knowledge on the part of the user of the bibliography.

The author of a BIBTEX file should be free to restructure it into subfiles without requiring modifications to all documents that use it. File inclusion is important to allow sharing of common material, such as `@String{...}` definitions.

SCRIBE uses the form

```
@Include{filename}
```

and BibTeX should too. It must be possible to nest file inclusions to a reasonable depth, at least five levels.

10 A lexical grammar for BibTeX

To test the recommendations of Section 9, I wrote and tested a `lex` grammar for BibTeX. It took just 22 rules to identify the 19 basic token types. The complete `lex` file was about 510 lines long, with about 340 lines of C code mostly concerned with the input and output of strings, and 120 lines of function and variable declarations. After `lex` processing, the complete C program was about 1130 lines long; with `flex`, it is 1700 lines long. This program is named `biblex`, and its output is compatible with that of `bibclean` with the `-no-prettyprint` option. However, it offers none of `bibclean`'s other services.

The `lex` grammar is presented in this section in the style of literate programming, with grammar rules interspersed with descriptive text. The index at the end of this document provides an essential feature of a literate program. To my knowledge, no `WEB` facility yet exists for `lex` and `yacc`, so this literate program must be handcrafted.

File structure

A `lex` file has this general structure:

```
definitions
%%
rules
%%
user functions
```

C declarations and definitions can be included in the declarations part if they are enclosed in `%{` and `%}`. Such text is copied verbatim to the output code file, together with additional `lex`-supplied header code.

Running `lex` on this file produces a C file that can be compiled and linked with a main program from the `lex` library to produce a working lexical analyzer. Alternatively, the user can write a customized main program which is linked with the `lex`-generated code to make a functional lexer.

In the following subsections, we describe the contents of the definitions and rules parts, but omit the user functions, since they are not relevant to understanding the grammar.

Macro definitions

The `lex` grammar begins with macro definitions. `lex` macros are single letters followed by a regular expression that defines them.

In regular expressions, square brackets delimit sets of characters, hyphen is used for character ranges inside sets, asterisk means zero or more of the preceding pattern, and plus means one or more. A period represents any character other than a newline.

`lex` macro names are braced to request expansion when they are used in grammar rules.

The first macro, `N`, represents the set of characters permitted in BibTeX names of abbreviations, citation keys, entries, and fields. If this set is ever modified, this is the *only* place where that job has to be done.

```
N     [A-Za-z][---A-Za-z0-9:.+/']*
```

It is not reasonable to make this set differ for these four different uses, because the differences are insufficient to distinguish between them lexically. We'll see later that we have to examine surrounding context to tell them apart.

Macro `O` represents the set of open delimiters that start a BibTeX entry argument. We could extend this grammar for Scribe by adding additional characters to the set.

```
O     [({]
```

Macro `W` represents a single horizontal space character.

```
W     [ \f\r\t\013]
```

Notice that we include formfeed, `\f`, and vertical tab, `\v`, in the set of horizontal space characters, even though they produce vertical motion on an output device. The reason is that we want to treat them just like blanks, and distinguish them from newlines, which are handled separately. `lex` does not recognize the escape sequence `\v`, so we have to reencode it in octal as `\013`.

Carriage return, `\r`, is not normally used in UNIX text files, but is common in some other operating systems. On the Apple Macintosh, carriage return is used instead of newline as an end-of-line marker. Fortunately, this will be transparent to us, because the C language requires [3, Section 2.2.2] that the implementation map host line terminators to newline on input, and newline back to host line terminators on output, so we will never see carriage returns on that system.

The last macro, `S`, represents optional horizontal space.

```
S     {W}*
```

Format of grammar rules

The remainder of the grammar consists of pairs of regular expression patterns and C code to execute when the pattern is matched. lex uses a "maximal munch" strategy in matching the longest possible sequence to handle the case where two rules have common leading patterns.

In the grammar file, the pairs are each written on a single line, but we wrap lines here to fit in the narrow journal columns, with the backslash-newline convention used earlier.

@ token

The first grammar rule says that an @ character should be recognized as the token named TOKEN_AT.

```
[@]     RETURN (out_token(TOKEN_AT));
```

On a successful match, the output function optionally emits the token, then returns its argument as a function value which the lexer in turn returns to the parser.

The C return statement is hidden inside the RETURN macro, because for yacc and bison, we need to bias bibclean's small integer token values to move them beyond the range of character ordinals.

Comment, Include, Preamble, and String tokens

The next four rules ignore letter case in matching the words Comment, Include, Preamble, or String. If they follow an @ character, they are identified as special tokens; otherwise, they are regarded as string abbreviation names.

```
[Cc][Oo][Mm][Mm][Ee][Nn][Tt] \
    RETURN ((last_token == TOKEN_AT) ?
             out_token(TOKEN_COMMENT) :
             out_token(TOKEN_ABBREV));

[Ii][Nn][Cc][Ll][Uu][Dd][Ee]/{S}{O} \
    RETURN ((last_token == TOKEN_AT) ?
        out_token(TOKEN_INCLUDE) :
        out_token(TOKEN_ABBREV));

[Pp][Rr][Ee][Aa][Mm][Bb][Ll][Ee]/{S}{O} \
    RETURN ((last_token == TOKEN_AT) ?
        out_token(TOKEN_PREAMBLE) :
        out_token(TOKEN_ABBREV));

[Ss][Tt][Rr][Ii][Nn][Gg]/{S}{O} \
    RETURN ((last_token == TOKEN_AT) ?
        out_token(TOKEN_STRING) :
        out_token(TOKEN_ABBREV));
```

Although lex supports examination of trailing context in order to identify tokens more precisely, the presence of arbitrary whitespace and in-line comments in this grammar makes it impossible to use this feature. The output routines remember the last non-space, non-comment token seen in order to make use of leading context to assist in token identification.

Abbreviation, entry, field, and key tokens

Several token types are recognized by a match with the name macro, N. Since the same set of characters can occur in abbreviations, entry names, field names, and key names, we have to use the record of leading context to distinguish between the various possibilities.

```
{N} {
        if (last_object == TOKEN_STRING)
            RETURN(out_token(TOKEN_ABBREV));
        switch (last_token)
        {
        case TOKEN_COMMA:
            RETURN(out_token(TOKEN_FIELD));
        case TOKEN_LBRACE:
            RETURN(out_token(TOKEN_KEY));
        case TOKEN_AT:
            RETURN(out_token(TOKEN_ENTRY));
        default:
            RETURN(out_token(TOKEN_ABBREV));
        }
    }
```

In the event of errors in the input stream, this identification of token types may be unreliable; such errors will be detected later in the parsing program.

Digit string

A *digit string* is an undelimited value string. The output function will supply the missing quotation mark delimiters, so that all strings take a standard form.

```
[0-9]+    RETURN (out_protected_string( \
    TOKEN_VALUE));
```

In-line comment token

A percent initiates an *in-line comment* that continues to the end of line and then over all leading horizontal space on the next line.

```
[%].*[\n]{S} \
    RETURN (out_token(TOKEN_INLINE));
```

Because this pattern marks the start of a new token, the previous token has already been terminated. Thus, an line-line comment *cannot* split a token. The same is true for TeX macros, though not for ordinary TeX text.

String concatenation token

A sharp sign is the BibTeX *string concatenation operator*.

```
[#]     RETURN (out_token(TOKEN_SHARP));
```

Delimited string token

A quotation mark initiates a *delimited string*.

```
["]     RETURN (out_string());
```

The complete string must be collected by the C function `out_string()` because regular expressions cannot count balanced delimiters.

BibTeX's quoted string syntax is a little unusual, in that an embedded quote is not represented by double quotes, as in Fortran, or by an escape sequence, as in C, but rather by putting the quote character in braces.

Brace tokens

Left and right *braces* are recognized as single tokens.

```
[{]     RETURN (out_lbrace());

[}]     RETURN (out_rbrace());
```

The output functions keep track of the current brace level to distinguish between outer braces delimiting a BibTeX entry, and inner braces delimiting a string value, and return `TOKEN_LBRACE`, `TOKEN_LITERAL`, `TOKEN_RBRACE`, or `TOKEN_STRING`, depending on preceding context.

`TOKEN_LITERAL` is used for the argument of a `Comment` and `Include` entries, and contains the delimiting braces.

Parenthesis tokens

In order to simplify the parser grammar, we remap outer *parentheses* delimiting arguments of BibTeX entries to *braces*. However, if the parentheses are not preceded by a valid entry name, they are output instead as single-character tokens of type `TOKEN_LITERAL`. They cannot legally occur in this context, but that error will be detected during the parsing stage. During lexical analysis, we do not want to have any error conditions.

```
[(]     RETURN (out_lparen());

[)]     RETURN (out_rparen());
```

To support Scribe, we would need to add patterns for other delimiters here.

Assignment and separator tokens

The *assignment operator* and *assignment separator* are returned as single tokens.

```
[=]     RETURN (out_token(TOKEN_EQUALS));

[,]     RETURN (out_token(TOKEN_COMMA));
```

Newline token

A *newline* is returned as a separate token because we want to be able to preserve line boundaries so that filter tools that make minimal perturbations on the input stream can be constructed.

```
[\n]    RETURN (out_token(TOKEN_NEWLINE));
```

Horizontal space token

Consecutive horizontal space characters are returned as a single space token, for the same reason that newlines are recognized as distinct tokens by the preceding rule.

```
{W}+    RETURN (out_token(TOKEN_SPACE));
```

Unclassifiable tokens

Finally, we have a catch-all rule: any character not recognized by one of the preceding rules is returned as a literal single-character token, and will cause an error during the parsing. The regular-expression character period matches anything but a newline, and we already have a rule for newline.

```
.    RETURN (out_token(TOKEN_LITERAL));
```

Lexical grammar summary

We now have a complete lexical grammar suitable for `lex` that can complete tokenize an arbitrary input stream containing any character values whatever.

The associated C code functions normalize entries by changing outer parentheses to braces, brace string delimiters to quotes, and undelimited digit strings to quoted strings.

All string tokens of type `TOKEN_VALUE` output by the lexer will contain surrounding quotes, and any nested quotes will be braced, with proper care taken to handle `\"` accent control sequences properly.

All special characters inside the quoted strings will be represented by the escape sequences given in Table 2 on page 401. Thus, even with a binary input stream, the output of the lexer will contain only printable characters.

It must be observed that `lex` is not capable of handling all 256 8-bit characters. In particular, it treats an ASCII NUL (`\0`) in a string as an end-of-file condition. Older versions of `lex` are not *8-bit clean*; they will not reliably handle characters 128–255. This latter deficiency is being remedied by the X/Open Consortium activities to internationalize and standard UNIX applications [32].

11 A parsing grammar for BibTeX

To complete the job, I wrote a `yacc` grammar for BibTeX. This was considerably more work than the `lex` grammar, mostly due to my relative inexperience with writing LALR(1) grammars, and it took several days to understand the process well enough to eliminate the grammatical ambiguities that initially plagued me.

The final complete `yacc` program is about 270 lines long, and produces a parser of 760 (`yacc`) to 1000 (`bison`) lines, excluding the lexer. The grammar contains just 35 rules. Ten of these rules could be eliminated if we arranged for the lexer to discard space and in-line comments, but for a prettyprinter and other BibTeX tools, they must be preserved. This parsing program is called `bibparse`; it can be used with the output of either `bibclean -no-prettyprint`, or `biblex`.

The complete BibTeX grammar is given below, expressed as `yacc` rules, again in literate programming style. It must be augmented by about 180 lines of C code to provide a working parser.

File structure

A `yacc` file has this general structure:

```
declarations
%%
rules
%%
user functions
```

C declarations and definitions can be included in the declarations part if they are enclosed in `%{` and `%}`. Such text is copied verbatim to the output code file, together with additional `yacc`-supplied header code.

Running `yacc` on this file produces a C file that can be compiled and linked with the lexical analyzer code to produce a working parser.

In the following subsections, we describe the contents of the declarations and rules parts, but omit the declaration C code and the user functions, since they are not relevant to understanding the grammar.

Format of grammar rules

The grammar rules will be presented in top-down order, from most general, to most particular, since this seems to be the best way to understand the overall structure of the grammar, and to ensure that it describes current BibTeX usage, plus our suggested extensions and clarifications.

The colon in a grammar rule should be read "is" or "produces", because the rule is also known as a *production*. A vertical bar separates alternatives, and can be read "or". A semicolon terminates the rule.

Lower-case letters are used for *non-terminals*, which are names of rules in the parser grammar. Upper-case letters are used for *terminals*, which are names of tokens recognized by the lexer.

The spacing shown is arbitrary, but conventional for `yacc` grammars: each rule starts a new line, with the right-hand side indented from the margin, and the semicolon occupies a separate line.

Token declarations

The `%token` declarations merely provide symbolic names for the integer token types returned by the lexer. The values are arbitrary, except that they must exceed 257, and must agree with the definitions in the lexer code. We simply increment the token types output from `bibclean` by 1000, matching the offset added in the `RETURN` macro in the lexer.

```
%token TOKEN_ABBREV      1001
%token TOKEN_AT          1002
%token TOKEN_COMMA       1003
%token TOKEN_COMMENT     1004
%token TOKEN_ENTRY       1005
%token TOKEN_EQUALS      1006
%token TOKEN_FIELD       1007
%token TOKEN_INCLUDE     1008
%token TOKEN_INLINE      1009
%token TOKEN_KEY         1010
%token TOKEN_LBRACE      1011
%token TOKEN_LITERAL     1012
%token TOKEN_NEWLINE     1013
%token TOKEN_PREAMBLE    1014
%token TOKEN_RBRACE      1015
%token TOKEN_SHARP       1016
%token TOKEN_SPACE       1017
%token TOKEN_STRING      1018
%token TOKEN_VALUE       1019
```

Precedence declarations

The `%nonassoc` declaration makes the assignment operator non-associative, so input of the form `a = b = c` is illegal.

```
%nonassoc TOKEN_EQUALS
```

The first `%left` declaration makes space, in-line comment, and newline tokens left associative, and of equal precedence.

```
%left TOKEN_SPACE TOKEN_INLINE \
      TOKEN_NEWLINE
```

The second `%left` declaration makes the BibTeX string concatenation character, `#`, left associative, and of higher precedence than space, in-line comment, and newline.

```
%left TOKEN_SHARP
```

These precedence settings are crucial for resolving conflicts in this grammar which arise in assignments when the parser has seen an assignment operator and a value. Without the operator precedences, it cannot decide whether to complete the assignment, or to read ahead looking for a concatenation operator.

BibTeX file

The beginning of the grammar rules is indicated by a pair of percent characters.

```
%%
```

The first rule defines what we are going to parse, namely, a BibTeX *file*. The left-hand side of the first rule is known as the grammar's *start symbol*.

```
bibtex_file:
        opt_space
      | opt_space object_list opt_space
      ;
```

This rule says that a BibTeX file contains either optional space, or optional space followed by a list of objects followed by optional space. This definition permits a file to be empty, or contain only space tokens, or have leading and trailing space.

Object lists

A *list of objects* is either a single object, or a list of such objects, separated by optional space from another object.

```
object_list:
        object
      | object_list opt_space object
      ;
```

For LL(1) parsers, usually implemented by hand-coded recursive descent programs, this kind of left-recursive rule must be rewritten by standard methods [2, pp. 47–48, 176–178] to avoid an infinite loop in the parser. In this rule, we would instead define a list as an object, separated by optional space from another list. However, for LALR(1) parsers, left-recursive definitions are preferable, because they avoid parser stack overflow with long lists.

Objects

An *object* is one of the BibTeX `@name{...}` constructs. Notice that we allow optional space between the `@` and the `name`.

```
object:
        TOKEN_AT opt_space at_object
      ;
```

In this grammar, we will consistently allow optional space between *any* pair of BibTeX tokens; space is described more precisely below. This convention is easy to remember, and easy to implement in the grammar rules.

While it would be possible to include the `@` as part of the `name`, making `@name` a single lexical token, both BibTeX and Scribe permit intervening space, so we cannot collapse the two into a single token.

Entry types and error recovery

Here are the possibilities for the `name` following an `@`, which we call an `at_object`.

```
at_object:
        comment
      | entry
      | include
      | preamble
      | string
      | error TOKEN_RBRACE
      ;
```

`Comment`, `Include`, `Preamble`, and `String` must be handled separately from other types of entries, like `Article` and `Book`, because their braced arguments have a different syntax.

The rule with `error` is a special one supported by `yacc` and `bison`. It says that if an `at_object` cannot be recognized at the current state of the parse, then the input should be discarded until a right brace is found. An error message will be issued when this happens, but recovery will be attempted following that right brace. Without this error handling, any input error will immediately terminate the parser, hardly a user-friendly thing to do.

This is the only place where we will attempt error repair, although we could certainly do so in other rules, such as in the assignment rule below. The goal here is to present a rigorous complete grammar, without additional embellishments that would complicate understanding.

Comment entry

A BibTeX `@Comment{...}` is special in that the only requirement on the argument is that delimiters be balanced. The lexer returns the delimited argument as a single literal string, including the delimiters, and standardizes the delimiters to braces.

```
comment:
        TOKEN_COMMENT opt_space
            TOKEN_LITERAL
        ;
```

Bibliography entry

A BibTeX *bibliography entry* is braced text containing a citation key, a comma, and a list of assignments. The rules provide for an optional assignment list, and for an optional trailing comma. To shorten the rules, we introduce a subsidiary rule, `entry_head`, to represent their common prefix.

```
entry:  entry_head
              assignment_list
              TOKEN_RBRACE
      | entry_head
              assignment_list
              TOKEN_COMMA opt_space
              TOKEN_RBRACE
      | entry_head TOKEN_RBRACE
      ;

entry_head:
        TOKEN_ENTRY opt_space
              TOKEN_LBRACE opt_space
              key_name opt_space
              TOKEN_COMMA opt_space
        ;
```

There is no `opt_space` item following `assignment_list` because it is included in the definition of the latter. This infelicity seems to be necessary to obtain a grammar that conforms to the LALR(1) requirements of `yacc` and `bison`.

Key name

Because of intervening newlines and in-line comments, the lexical analyzer cannot always correctly recognize a *citation key* from trailing context. It might instead erroneously identify the token as an abbreviation. We therefore need to account for both possibilities:

```
key_name:
        TOKEN_KEY
      | TOKEN_ABBREV
      ;
```

`Include` entry

The `Include` entry is followed by a file name enclosed in balanced braces.

```
include:
        TOKEN_INCLUDE opt_space
            TOKEN_LITERAL
        ;
```

Because file names are operating-system dependent, the only restrictions that are placed on the file name are that it cannot contain unbalanced braces, and that it cannot contain leading or trailing space. However, the file name can have embedded space if the operating system permits.

BibTeX should discard the delimiting braces and surrounding space in the `TOKEN_LITERAL` to isolate the file name. It should search for this file in its standard input path, so that the file name need not contain an absolute directory path. This feature is not supported in BibTeX 0.99c, but `bibclean` and the lexer and parser recognize it in anticipation of its eventual incorporation.

`Preamble` entry

The `Preamble` entry argument is a braced BibTeX string value. BibTeX outputs the argument verbatim, minus the outer delimiters, to the `.bbl` file for TeX to process.

```
preamble:
        TOKEN_PREAMBLE opt_space
            TOKEN_LBRACE opt_space
            value opt_space
            TOKEN_RBRACE
        ;
```

`String` entry

The `String` entry argument is a braced single assignment.

```
string:
        TOKEN_STRING opt_space
            TOKEN_LBRACE opt_space
            assignment opt_space
            TOKEN_RBRACE
        ;
```

Value string

A BibTeX *value* is a string, which may be a simple value, or a list of strings separated by the string concatenation operator.

```
value:  simple_value
      | value opt_space
            TOKEN_SHARP opt_space
            simple_value
      ;
```

Simple values

A *simple value* is either a delimited string, returned by the lexer as a `TOKEN_VALUE`, or a string abbreviation, returned as a `TOKEN_ABBREV`.

```
simple_value:
        TOKEN_VALUE
      | TOKEN_ABBREV
      ;
```

The lexer can distinguish between these two because of the string delimiters. It is up to the parser support code to verify that an abbreviation is actually defined before it is used.

Assignment list

The body of most BIBTEX entries consists of a list of one or more assignments, separated by commas. Notice that this definition does not provide for an optional trailing comma after the last assignment. We handled that above in the rules for `entry`.

```
assignment_list:
        assignment
      | assignment_list
            TOKEN_COMMA opt_space
            assignment
      ;
```

Assignment

An *assignment* has a left-hand side separated from a value by the assignment operator, `=`.

```
assignment:
        assignment_lhs opt_space
            TOKEN_EQUALS opt_space value
            opt_space
      ;
```

Trailing optional space is included here, and omitted before the comma in `assignment_list`, in order to allow the LALR(1) parser to successfully distinguish between space between a value and a comma, and space between a value and a string concatenation operator.

My initial version of this grammar did not have this optional space item, and the resulting parser proved unable to recognize input in which a space separated a value from a comma or closing brace; it took quite a bit of experimentation to determine how to rewrite the grammar to remove this problem.

The left-hand side of an assignment is either a field name, like `author` or `title`, or a string abbreviation name. The lexer must distinguish between the two by remembering the last entry type seen, because they are made up of exactly the same set of possible characters.

```
assignment_lhs:
        TOKEN_FIELD
      | TOKEN_ABBREV
      ;
```

Optional space

Optional space is either an empty string, here indicated by the `/*...*/` comment, or space.

```
opt_space:
        /* empty */
      | space
      ;
```

Space

Space is an important part of the grammar. It is one or more single spaces.

```
space:  single_space
      | space single_space
      ;
```

We include space handling to support tools that process BIBTEX files and wish to preserve the input form. In normal compiler design, space is recognized by the lexer, and discarded, so the parser never has to deal with it, and the grammar can be considerably simpler.

Single space

The final rule of the grammar defines a *single space* as a literal space character, or an in-line comment, or a literal newline character.

```
single_space:
        TOKEN_SPACE
      | TOKEN_INLINE
      | TOKEN_NEWLINE
      ;
```

Although we could arrange for the lexer to merge `TOKEN_SPACE` and `TOKEN_NEWLINE` into a single token, this would interfere with heuristics used by a prettyprinter to detect empty lines inside string values, which are possibly indicative of missing delimiters.

Parsing grammar summary

We have now completed a `yacc` grammar for BIBTEX that provides a rigorous grammatical analysis of a stream of tokens recognized by the lexers in Sections 6 and 10.

Notice that there is no character-string processing whatever in the parser, because it has all been done in the lexer. Parsing operations just manipulate small integer values.

In this version, no actions have been supplied as C code fragments in the `yacc` grammar. The only output of the parser will be the token stream from the lexer, interspersed by error messages when the input fails to match a grammar rule.

Error recovery has been kept simple: input is flushed to the next closing brace, which is presumably the end of an entry. Braces of type `TOKEN_LBRACE` and `TOKEN_RBRACE` do not occur except around apparent entries in the lexer output; other braces are returned as tokens of type `TOKEN_LITERAL`.

No more than one token of lookahead is required by this grammar, although the lexer often looked several characters ahead to examine trailing context in order to distinguish between otherwise similar tokens.

BibTeX users should be able to read this grammar and decide whether a questionable BibTeX construct is legal or not, without having to resort to software experiments as I did to clarify fuzzy grammatical points.

12 Software availability

The source code and documentation for `bibclean` are in the *public domain*, in the interests of the widest availability and greatest benefit to the TeX community. Commercial vendors of TeXware are encouraged to include `bibclean` with their distributions.

The distribution also includes the separate complete lexer and parser grammar and code, which can be processed by `lex` or `flex`, and `yacc` or `bison`, respectively. The output C code from these tools is included so that recipients need not have them installed to actually compile and run the lexer and parser.

If you have Internet anonymous `ftp` access, you can retrieve the distribution in a variety of archive formats from the machine `ftp.math.utah.edu` in the directory `pub/tex/bib`. Major TeX Internet archive hosts around the world will also have `bibclean`, but the author's site will always have the most up-to-date version. If you lack `ftp` capability but have electronic mail access, a message to `tuglib@math.utah.edu` with the text

```
help
send index from tex/bib
```

will get you started.

The `bibclean` distribution includes a substantial collection of torture tests that should be run at installation time to verify correctness. As with the TeX `trip` and METAFONT `trap` tests, this testing has proved valuable in uncovering problems before the code is installed for general use.

References

[1] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, Reading, MA, USA, 1988. ISBN 0-201-07981-X.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986. ISBN 0-201-10088-6.

[3] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language C, ANSI X3.159-1989*, December 14 1989.

[4] Nelson H. F. Beebe. Publications about TeX and typography. *TUGBoat*, Supplement to 12 (2):176–183, May 1991.

[5] Nelson H. F. Beebe. Publications prepared with TeX. *TUGBoat*, Supplement to 12(2):183–194, May 1991.

[6] Martin Bryan. *SGML—An Author's Guide to the Standard Generalized Markup Language*. Addison-Wesley, Reading, MA, USA, 1988. ISBN 0-201-17535-5.

[7] Debra Cameron and Bill Rosenblatt. *Learning GNU Emacs*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1991. ISBN 0-937175-84-6.

[8] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, USA, 1990. ISBN 0-201-51459-1.

[9] Alan R. Feuer. *The C Puzzle Book*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1989. ISBN 0-13-115502-4.

[10] Samuel P. Harbison and Guy L. Steele Jr. *C—A Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, third edition, 1991. ISBN 0-13-110933-2.

[11] Rex Jaeschke. *Portability and the C Language*. Hayden Books, 4300 West 62nd Street, Indianapolis, IN 46268, USA, 1989. ISBN 0-672-48428-5.

[12] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Reinhart, and Winston, New York, NY, USA, 1979. AT&T Bell Laboratories Technical Report, July 31, 1978.

[13] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1978. ISBN 0-13-110163-3.

[14] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1988. ISBN 0-13-110362-8.

[15] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965. This is the original paper on the theory of LR(k) parsing.

[16] Donald E. Knuth. *The TeXbook*, volume A of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986. ISBN 0-201-13447-0.

[17] Andrew Koenig. *C Traps and Pitfalls*. Addison-Wesley, Reading, MA, USA, 1989. ISBN 0-201-17928-8.

[18] Leslie Lamport. *LaTeX—A Document Preparation System—User's Guide and Reference Manual*. Addison-Wesley, Reading, MA, USA, 1985. ISBN 0-201-15790-X.

[19] J. E. Lapin. *Portable C and UNIX Programming*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1987. ISBN 0-13-686494-5.

[20] Michael E. Lesk and Eric Schmidt. Lex—a lexical analyzer generator. In *UNIX Programmer's Manual*, volume 2, pages 388–400. Holt, Reinhart, and Winston, New York, NY, USA, 1979. AT&T Bell Laboratories Technical Report in 1975.

[21] John R. Levine, Tony Mason, and Doug Brown. `lex` *&* `yacc`. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, second edition, 1992. ISBN 1-56592-000-7. 400 pp. US$29.95.

[22] Tony Mason and Doug Brown. `lex` *&* `yacc`. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1990. ISBN 0-937175-49-8.

[23] P. J. Plauger. *The Standard C Library*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1992. ISBN 0-13-838012-0.

[24] Henry Rabinowitz and Chaim Schaap. *Portable C*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1990. ISBN 0-13-685967-4.

[25] Brian Reid. *Scribe User's Manual*. Carnegie-Mellon University, Pittsburgh, PA, USA, third edition, 1980.

[26] Marc J. Rochkind. *Advanced UNIX Programming*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1985. ISBN 0-13-011818-4 (hardback), 0-13-011800-1 (paperback).

[27] Michael A. Schoonover, John S. Bowie, and William R. Arnold. *GNU Emacs: UNIX Text Editing and Programming*. Addison-Wesley, Reading, MA, USA, 1992. ISBN 0-201-56345-2. 610 pp. LCCN QA76.76.T49S36.

[28] Axel T. Schreiner and H. George Friedman, Jr. *Introduction to Compiler Construction Under UNIX*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1985. ISBN 0-13-474396-2. 224 pp.

[29] W. Richard Stevens. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1990. ISBN 0-13-949876-1.

[30] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, USA, second edition, 1991. ISBN 0-201-53992-6.

[31] Eric van Herwijnen. *Practical SGML*. Kluwer Academic Publishers Group, Norwell, MA, USA, 1990. ISBN 0-7923-0635-X. xviii + 307 pp. £24.90 (1990).

[32] X/Open Company, Ltd. *X/Open Portability Guide, XSI Commands and Utilities*, volume 1. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1989. ISBN 0-13-685835-X.

Index

◇ Nelson H. F. Beebe
Center for Scientific Computing
Department of Mathematics
University of Utah
Salt Lake City, UT 84112
USA
Tel: +1 801 581 5254
FAX: +1 801 581 4148
Internet: beebe@math.utah.edu