A Replacement for BIBTEX (Version 0.9.18)

Norman Ramsey

July 19, 2019

Contents

1 Overview

The code herein comprises the "nbib" package, which is a collection of tools to help authors take better advantage of BibTeX data, especially when working in collaboration. The driving technology is that instead of using BibTeX "keys," which are chosen arbitrarily and idiosyncratically, nbib builds a bibliography by searching the contents of citations.

- `nbibtex` is a drop-in replacement for `bibtex`. Authors' `\cite{...}` commands are interpreted either as classic BibTeX keys (for backward compatibility) or as search commands. Thus, if your bibliography contains the classic paper on type inference, `nbibtex` should find it using a citation like `\cite{damas-milner:1978}`, or `\cite{damas-milner:polymorphism}`, or perhaps even simply `\cite{damas-milner}`—*regardless* of the BibTeX key you may have chosen. The same citations should also work with your coauthors' bibliographies, even if they are keyed differently.

- `nbibfind` uses the nbib search engine on the command line. If you know you are looking for a paper by Harper and Moggi, you can just type

  ```
  nbibfind harper-moggi
  ```

  and see what comes out.

- To help you work with coauthors who don't have the nbib package, `nbibmake`[1] examines a LaTeX document and builds a custom `.bib` file just for that document.

The package is written in a combination of C and Lua:

- Because I want nbib to be able to handle bibliographies with thousands or tens of thousands of entries, the code to parse a `.bib` "database" is written in C. A computer bought in 2003 can parse over 15,000 entries per second.

- Because the search for BibTeX entries requires string searching on every entry, the string search is also written in C (and uses Boyer-Moore).

- Because string manipulation is much more easily done in Lua, all the code that converts a BibTeX entry into printed matter is written in Lua, as is all the "driver" code that implements various programs.

The net result is that `nbibtex` is about five times slower than classic `bibtex`. This slowdown is easy to observe when printing a bibliography of several thousand entries, but on a typical paper with fewer than fifty citations and a personal bibliography with a thousand entries, the pause is imperceptible.

1.1 Compatibility

I've made every effort to make NbibTeX compatible with BibTeX, so that NbibTeX can be used on existing papers and should produce the same output as BibTeX. Regrettably, compatibility means avoiding modern treatment of non-ASCII characters, such as are found in the ISO Latin-1 character set: classic BibTeX simply treats every non-ASCII character as a letter.

- It would be pleasant to try instead to set NbibTeX to use an ISO 8859-1 locale, but this leads to incompatible output: NbibTeX forces characters to lower case that BibTeX leaves alone.

  ⟨*pleasant code that results in incompatible output*⟩≡

  ```
  do
    local locales =
      { "en_US", "en_AU", "en_CA", "en_GB", "fr_CA", "fr_CH", "fr_FR", }
    for _, l in pairs(locales) do
      if os.setlocale(l .. '.iso88591', 'ctype') then break end
    end
  end
  ```

[1] Not yet implemented.

- A much less pleasant alternative would be to abandon the support that Lua provides for distinguishing letters from nonletters and instead to try to do some sort of system-dependent character classification, as is done in BibTeX. I don't have the stomach for it.

- The most principled solution I can imagine would be to define a special "BibTeX locale," whose sole purpose would be to guarantee compatibility with BibTeX. But this potential solution looks like a nightmare for software distribution.

- What I've done is proceed blithely with the user's current locale, throwing in a hack here or there as needed to guarantee compatibility with the test cases I have in the default locale I happen to use. The most notable case is `bst.purify`, which is used to generate keys for sorting.

Expedience carries the day. Feh.

2 Parsing `.bib` files

This section reads the `.bib` file(s).

⟨*nbib.c*⟩≡

```
#include <stdio.h>
#include <assert.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>

#include <lua.h>
#include <lauxlib.h>

⟨type definitions⟩
⟨function prototypes⟩
⟨initialized and uninitialized data⟩
⟨macro definitions⟩
⟨Procedures and functions for input scanning⟩
⟨function definitions⟩
```

2.1 Internal interfaces

2.1.1 Data structures

For convenience in keeping function prototypes uncluttered, all state associated with reading a particular BibTeX file is stored in a single `Bibreader` abstraction. That state is divided into three groups:

- Fields that say what file we are reading and what is our position within that file
- A buffer that holds one line of the `.bib` file currently being scanned
- State accessible from Lua: an interpreter; a list of strings from the `.bib` preamble, which is exposed to the client; a warning function provided by the client; and a macro table provided by the client and updated by `@string` commands

In the buffer, the meaningful characters are in the half-open interval [`buf`, `lim`), and we reserve space for a sentinel at `lim`. The invariant is that $\texttt{buf} \le \texttt{cur} < \texttt{lim}$ and $\texttt{buf} + \texttt{bufsize} \ge \texttt{lim} + 1$.

⟨*type definitions*⟩≡

```
typedef struct bibreader {
  const char *filename;            /* name of the .bib file */
  FILE *file;                      /* .bib file open for read */
  int line_num;                    /* line number of the .bib file */
  int entry_line;                  /* line number of last seen entry start */

  unsigned char *buf, *cur, *lim;   /* input buffer */
  unsigned bufsize;                /* size of buffer */
  char entry_close;   /* character expected to close current entry */

  lua_State *L;
  int preamble;       /* reference to preamble list of strings */
  int warning;        /* reference to universal warning function */
  int macros;         /* reference to macro table */
} *Bibreader;
```

The `is_id_char` array is used to define a predicate that says whether a character is considered part of an identifier.

⟨*initialized and uninitialized data*⟩≡

```
bool is_id_char[256];    /* needs initialization */
#define concat_char '#'  /* used to concatenate parts of a field defn */
```

2.1.2 Scanning

Most internal functions are devoted to some form of scanning. The model is a bit like Icon: scanning may succeed or fail, and it has a side effect on the state of the reader—in particular the value of the `cur` pointer, and possibly also the contents of the buffer. (Unlike Icon, there is no backtracking.) Success or failure is nonzero or zero but is represented using type `bool`.

⟨*function prototypes*⟩≡

```
typedef int bool;
```

Function `get_line` refills the buffer with a new line (and updates `line_num`), returning failure on end of file.

⟨*function prototypes*⟩+≡

```
static bool get_line(Bibreader rdr);
```

Several scanning functions come in two flavors, which depend on what happends at the end of a line: the `_get_line` flavor refills the buffer and keeps scanning; the normal flavor fails. Here are some functions that scan for combinations of particular characters, whitespace, and nonwhite characters.

⟨*function prototypes*⟩+≡

```
static bool upto1(Bibreader rdr, char c);
static bool upto1_get_line(Bibreader rdr, char c);
static void upto_white_or_1(Bibreader rdr, char c);
static void upto_white_or_2(Bibreader rdr, char c1, char c2);
static void upto_white_or_3(Bibreader rdr, char c1, char c2, char c3);
static bool upto_nonwhite(Bibreader rdr);
static bool upto_nonwhite_get_line(Bibreader rdr);
```

Because there is always whitespace at the end of a line, the `upto_white_*` flavor cannot fail.

Some more sophisticated scanning functions. None attempts to return a value; instead each functions scans past the token in question, which the client can then find between the old and new values of the `cur` pointer.

⟨*function prototypes*⟩+≡

```
static bool scan_identifier (Bibreader rdr, char c1, char c2, char c3);
static bool scan_nonneg_integer (Bibreader rdr, unsigned *np);
```

Continuing from low to high level, here are functions used to scan fields, about which more below:

⟨*function prototypes*⟩+≡

```
static bool scan_and_buffer_a_field_token (Bibreader rdr, int key, luaL_Buffer *b);
static bool scan_balanced_braces(Bibreader rdr, char close, luaL_Buffer *b);
static bool scan_and_push_the_field_value (Bibreader rdr, int key);
```

Two utility functions used after scanning: The `lower_case` function overwrites buffer characters with their lowercase equivalents. The `strip_leading_and_trailing_space` functions removes leading and trailing space characters from a string on top of the Lua stack.

⟨*function prototypes*⟩+≡

```
static void lower_case(unsigned char *p, unsigned char *lim);
static void strip_leading_and_trailing_space(lua_State *L);
```

2.1.3 Other functions

⟨*function prototypes*⟩+≡

```
static int get_bib_command_or_entry_and_process(Bibreader rdr);
int luaopen_bibtex (lua_State *L);
```

2.1.4 Commands

In addition to database entries, a `.bib` file may contain the `comment`, `preamble`, and `string` commands. Each is implemented by a function of type `Command`, which is associated with the name by `find_command`.

⟨*function prototypes*⟩+≡

```
typedef bool (*Command)(Bibreader);
static Command find_command(unsigned char *p, unsigned char *lim);
static bool do_comment (Bibreader rdr);
static bool do_preamble(Bibreader rdr);
static bool do_string  (Bibreader rdr);
```

2.1.5 Error handling

The warnv function is used to call the warning function supplied by the Lua client. In addition to the reader, it takes as arguments the number of results expected and the signature of the arguments. (The warning function may receive any combination of string (s), floating-point (f), and integer (d) arguments; the fmt string gives the sequence of the arguments that follow.)

⟨*function prototypes*⟩+≡

```
static void warnv(Bibreader rdr, int nres, const char *fmt, ...);
```

There's a lot of crap here to do with reporting errors. An error in a function called direct from Lua pushes false and a message and returns 2; an error in a boolean function pushes the same but returns failure to its caller. I hope to replace this code with native Lua error handling (lua_error).

⟨*macro definitions*⟩≡

```
#define LERRPUSH(S) do { \
  if (!lua_checkstack(rdr->L, 10)) assert(0); \
  lua_pushboolean(rdr->L, 0); \
  lua_pushfstring(rdr->L, "%s, line %d: ", rdr->filename, rdr->line_num); \
  lua_pushstring(rdr->L, S); \
  lua_concat(rdr->L, 2); \
  } while(0)
#define LERRFPUSH(S,A) do { \
  if (!lua_checkstack(rdr->L, 10)) assert(0); \
  lua_pushboolean(rdr->L, 0); \
  lua_pushfstring(rdr->L, "%s, line %d: ", rdr->filename, rdr->line_num); \
  lua_pushfstring(rdr->L, S, A); \
  lua_concat(rdr->L, 2); \
  } while(0)
#define LERR(S)   do { LERRPUSH(S);   return 2; } while(0)
#define LERRF(S,A) do { LERRFPUSH(S,A); return 2; } while(0)
  /* next: cases for Boolean functions */
#define LERRB(S)   do { LERRPUSH(S);   return 0; } while(0)
#define LERRFB(S,A) do { LERRFPUSH(S,A); return 0; } while(0)
```

2.2 Reading a database entry

Syntactically, a `.bib` file is a sequence of entries, perhaps with a few `.bib` commands thrown in. Each entry consists of an at sign, an entry type, and, between braces or parentheses and separated by commas, a database key and a list of fields. Each field consists of a field name, an equals sign, and nonempty list of field tokens separated by `concat_chars`. Each field token is either a nonnegative number, a macro name (like 'jan'), or a brace-balanced string delimited by either double quotes or braces. Finally, case differences are ignored for all but delimited strings and database keys, and whitespace characters and ends-of-line may appear in all reasonable places (i.e., anywhere except within entry types, database keys, field names, and macro names); furthermore, comments may appear anywhere between entries (or before the first or after the last) as long as they contain no at signs.

This function reads a database entry and pushes it on the Lua stack. Any commands encountered before the database entry are executed. If no entry remains, the function returns 0.

⟨*function definitions*⟩≡

```
#undef ready_tok
#define ready_tok(RDR) do { \
  if (!upto_nonwhite_get_line(RDR)) \
    LERR("Unexpected end of file"); \
  } while(0)

static int get_bib_command_or_entry_and_process(Bibreader rdr) {
  unsigned char *id, *key;
  int keyindex;
  bool (*command)(Bibreader);
 getnext:
  ⟨scan rdr up to and past the next @ sign and skip white space (or return 0)⟩

  id = rdr->cur;
  if (!scan_identifier (rdr, '{', '(', '('))
    LERR("Expected an entry type");
  lower_case (id, rdr->cur);       /* ignore case differences */
  ⟨if [id, rdr->cur) points to a command, execute it and go to getnext⟩

  lua_pushlstring(rdr->L, (char *) id, rdr->cur - id);    /* push entry type */
  rdr->entry_line = rdr->line_num;
  ready_tok(rdr);
  ⟨scan past opening delimiter and set rdr->entry_close⟩
  ready_tok(rdr);
  key = rdr->cur;
  ⟨set rdr->cur to next whitespace, comma, or possibly }⟩
  lua_pushlstring(rdr->L, (char *) key, rdr->cur - key);  /* push database key */
  keyindex = lua_gettop(rdr->L);
  lua_newtable(rdr->L);                                   /* push table of fields */
  ready_tok(rdr);
  for (; *rdr->cur != rdr->entry_close; ) {
    ⟨absorb comma (breaking if followed by rdr->entry_close)⟩
    ⟨read a field-value pair and set it in the field table, which is on top of the Lua stack⟩
    ready_tok(rdr);
  }
  rdr->cur++;  /* skip past close of entry */
  return 3; /* entry type, key, table of fields */
}
```

⟨*scan* `rdr` *up to and past the next* `@` *sign and skip white space (or return 0)*⟩≡

```
if (!upto1_get_line(rdr, '@'))
  return 0;  /* no more entries; return nil */
assert(*rdr->cur == '@');
rdr->cur++;   /* skip the @ sign */
ready_tok(rdr);
```

⟨*if* [`id`, `rdr->cur`) *points to a command, execute it and go to* `getnext`⟩≡

```
command = find_command(id, rdr->cur);
if (command) {
  if (!command(rdr))
    return 2; /* command put (false, message) on Lua stack; we're done */
  goto getnext;
}
```

An entry is delimited either by braces or by brackets; in order to recognize the correct closing delimiter, we put it in `rdr->entry_close`.

⟨*scan past opening delimiter and set* `rdr->entry_close`⟩≡

```
if (*rdr->cur == '{')
    rdr->entry_close = '}';
else if (*rdr->cur == '(')
    rdr->entry_close = ')';
else
    LERR("Expected entry to open with { or (");
rdr->cur++;
```

I'm not quite sure why stopping at `}` is conditional on the closing delimiter in this way.

⟨*set* `rdr->cur` *to next whitespace, comma, or possibly* `}`⟩≡

```
if (rdr->entry_close == '}') {
  upto_white_or_1(rdr, ',');
} else {
  upto_white_or_2(rdr, ',', '}');
}
```

At this point we're at a nonwhite token that is not the closing delimiter. If it's not a comma, there's big trouble—but even if it is, the database may be using comma as a terminator, in which case a closing delimiter signals the end of the entry.

⟨*absorb comma (breaking if followed by* `rdr->entry_close`*)*⟩≡

```
if (*rdr->cur == ',') {
  rdr->cur++;
  ready_tok(rdr);
  if (*rdr->cur == rdr->entry_close) {
    break;
  }
} else {
  LERR("Expected comma or end of entry");
}
```

The syntax for a field is *identifier*=*value*. The field name is forced to lower case.

⟨*read a field-value pair and set it in the field table, which is on top of the Lua stack*⟩≡

```
if (id = rdr->cur, !scan_identifier (rdr, '=', '=', '='))
  LERR("Expected a field name");
lower_case(id, rdr->cur);
lua_pushlstring(rdr->L, (char *) id, rdr->cur - id);  /* push field name */
ready_tok(rdr);
if (*rdr->cur != '=')
  LERR("Expected '=' to follow field name");
rdr->cur++;                   /* skip over the [['=']] */
ready_tok(rdr);
if (!scan_and_push_the_field_value(rdr, keyindex))
  return 2;
strip_leading_and_trailing_space(rdr->L);
⟨if field is not already set, set it; otherwise warn⟩
```

Official BibTeX does not permit duplicate entries for a single field. But in entries on the net, you see lots of such duplicates in such unofficial fields as `reffrom`. Because classic BibTeX doesn't report errors on fields that aren't advertised by the `.bst` file, we don't want to just blat out a whole bunch of warning messages. So instead we dump the problem on the warning function provided by the Lua client.

We therefore can't simply set the field in the field table: we first look it up, and if it is nil, we set it; otherwise we warn.

⟨*if field is not already set, set it; otherwise warn*⟩≡

```
lua_pushvalue(rdr->L, -2);  /* push key */
lua_gettable(rdr->L, -4);
if (lua_isnil(rdr->L, -1)) {
  lua_pop(rdr->L, 1);
  lua_settable(rdr->L, -3);
} else {
  lua_pop(rdr->L, 1);  /* off comes old value  */
  warnv(rdr, 0, "ssdsss", /* tag, file, line, cite-key, field, newvalue */
        "extra field", rdr->filename, rdr->line_num,
        lua_tostring(rdr->L, keyindex),
        lua_tostring(rdr->L, -2), lua_tostring(rdr->L, -1));
  lua_pop(rdr->L, 2);  /* off come key and new value */
}
```

2.3 Scanning functions

2.3.1 Scanning functions for fields

While scanning fields, we are not operating in a toplevel function, so the error handling for `ready_tok` needs to be a bit different.

⟨*Procedures and functions for input scanning*⟩≡

```
#undef ready_tok
#define ready_tok(RDR) do { \
  if (!upto_nonwhite_get_line(RDR)) \
    LERRB("Unexpected end of file"); \
  } while(0)
```

Each field value is accumulated into a `luaL_Buffer` from the Lua auxiliary library. The buffer is always called `b`; for conciseness, we use the macro `copy_char` to add a character to it.

⟨*Procedures and functions for input scanning*⟩+≡

```
#define copy_char(C) luaL_putchar(b, (C))
```

A field value is a sequence of one or more tokens separated by a `concat_char` (`#` mark). A precondition for calling `scan_and_push_the_field_value` is that `rdr` is pointing at a nonwhite character.

⟨*Procedures and functions for input scanning*⟩+≡

```
  static bool scan_and_push_the_field_value (Bibreader rdr, int key) {
    luaL_Buffer field;

    luaL_checkstack(rdr->L, 10, "Not enough Lua stack to parse bibtex database");
    luaL_buffinit(rdr->L, &field);
    for (;;) {
      if (!scan_and_buffer_a_field_token(rdr, key, &field))
        return 0;
      ready_tok(rdr); /* cur now points to [[concat_char]] or end of field */
      if (*rdr->cur != concat_char) break;
      else { rdr->cur++; ready_tok(rdr); }
    }
    luaL_pushresult(&field);
    return 1;
  }
```

Because `ready_tok` can `return` in case of error, we can't write

```
for(; *rdr->cur == concat_char; rdr->cur++, ready_tok(rdr)).
```

A field token is either a nonnegative number, a macro name (like 'jan'), or a brace-balanced string delimited by either double quotes or braces. Thus there are four possibilities for the first character of the field token: If it's a left brace or a double quote, the token (with balanced braces, up to the matchin closing delimiter) is a string; if it's a digit, the token is a number; if it's anything else, the token is a macro name (and should thus have been defined by either the `.bst`-file's `macro` command or the `.bib`-file's `string` command). This function returns `false` if there was a serious syntax error.

⟨*Procedures and functions for input scanning*⟩+≡

```
static bool scan_and_buffer_a_field_token (Bibreader rdr, int key, luaL_Buffer *b) {
  unsigned char *p;
  unsigned number;
  *rdr->lim = ' ';
  switch (*rdr->cur) {
    case '{': case '"':
      return scan_balanced_braces(rdr, *rdr->cur == '{' ? '}' : '"', b);
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
      p = rdr->cur;
      scan_nonneg_integer(rdr, &number);
      luaL_addlstring(b, (char *)p, rdr->cur - p);
      return 1;
   default:
      /* named macro */
      p = rdr->cur;
      if (!scan_identifier(rdr, ',', rdr->entry_close, concat_char))
        LERRB("Expected a field part");
      lower_case (p, rdr->cur);   /* ignore case differences */
        /* missing warning of macro name used in its own definition */
      lua_pushlstring(rdr->L, (char *) p, rdr->cur - p); /* stack: name */
      lua_getref(rdr->L, rdr->macros);                   /* stack: name macros */
      lua_insert(rdr->L, -2);                            /* stack: name macros name */
      lua_gettable(rdr->L, -2);                          /* stack: name defn */
      lua_remove(rdr->L, -2);                            /* stack: defn */
      ⟨if top of stack is nil, pop it and warn of undefined macro; else buffer it⟩
      return 1;
  }
}
```

Here's another warning that's kicked out to the client. Reason: standard BibTeX complains only if it intends to use the entry in question.

⟨*if top of stack is nil, pop it and warn of undefined macro; else buffer it*⟩≡

```
{ int t = lua_gettop(rdr->L);
  if (lua_isnil(rdr->L, -1)) {
    lua_pop(rdr->L, 1);
    lua_pushlstring(rdr->L, (char *) p, rdr->cur - p);
    warnv(rdr, 1, "ssdss", /* tag, file, line, key, macro */
          "undefined macro", rdr->filename, rdr->line_num,
          key ? lua_tostring(rdr->L, key) : NULL, lua_tostring(rdr->L, -1));
    if (lua_isstring(rdr->L, -1))
      luaL_addvalue(b);
    else
      lua_pop(rdr->L, 1);
    lua_pop(rdr->L, 1);
  } else {
    luaL_addvalue(b);
  }
  assert(lua_gettop(rdr->L) == t-1);
}
```

This `.bib`-specific function scans and buffers a string with balanced braces, stopping just past the matching `close`. The original BibTeX tries to optimize the common case of a field with no internal braces; I don't. A precondition for calling this function is that `rdr->cur` point at the opening delimiter. Whitespace is compressed to a single space character.

⟨*Procedures and functions for input scanning*⟩+≡

```
static int scan_balanced_braces(Bibreader rdr, char close, luaL_Buffer *b) {
  unsigned char *p, *cur, c;
  int braces = 0;  /* number of currently open braces *inside* string */

  rdr->cur++;   /* scan past left delimiter */
  *rdr->lim = ’ ’;
  if (isspace(*rdr->cur)) {
    copy_char(’ ’);
    ready_tok(rdr);
  }
  for (;;) {
    p = rdr->cur;
    upto_white_or_3(rdr, ’}’, ’{’, close);
    cur = rdr->cur;
    for ( ; p < cur; p++) /* copy nonwhite, nonbrace characters */
      copy_char(*p);
    *rdr->lim = ’ ’;
    c = *cur;  /* will be whitespace if at end of line */
    ⟨depending on c, return or adjust braces and continue⟩
  }
}
```

Beastly complicated:

- Space is compressed and scanned past.
- A closing delimiter ends the scan at brace level 0 and otherwise is buffered.
- Braces adjust the `braces` count.

⟨*depending on* `c`*, return or adjust* `braces` *and continue*⟩≡

```
if (isspace(c)) {
  copy_char(' ');
  ready_tok(rdr);
} else {
  rdr->cur++;
  if (c == close) {
    if (braces == 0) {
      luaL_pushresult(b);
      return 1;
    } else {
      copy_char(c);
      if (c == '}')
        braces--;
    }
  } else if (c == '{') {
    braces++;
    copy_char(c);
  } else {
    assert(c == '}');
    if (braces > 0) {
      braces--;
      copy_char(c);
    } else {
      luaL_pushresult(b); /* restore invariant */
      LERRB("Unexpected '}'");
    }
  }
}
```

2.3.2 Low-level scanning functions

Scan the reader up to the character requested or end of line; fails if not found.

⟨*function definitions*⟩+≡

```
static bool upto1(Bibreader rdr, char c) {
  unsigned char *p = rdr->cur;
  unsigned char *lim = rdr->lim;
  *lim = c;
  while (*p != c)
    p++;
  rdr->cur = p;
  return p < lim;
}
```

Scan the reader up to the character requested or end of file; fails if not found.

⟨*function definitions*⟩+≡
```
static int upto1_get_line(Bibreader rdr, char c) {
  while (!upto1(rdr, c))
    if (!get_line(rdr))
      return 0;
  return 1;
}
```

Scan the reader up to the next whitespace or the one character requested. Always succeeds, because the end of the line is whitespace.

⟨*function definitions*⟩+≡
```
static void upto_white_or_1(Bibreader rdr, char c) {
  unsigned char *p = rdr->cur;
  unsigned char *lim = rdr->lim;
  *lim = c;
  while (*p != c && !isspace(*p))
    p++;
  rdr->cur = p;
}
```

Scan the reader up to the next whitespace or either of two characters requested.

⟨*function definitions*⟩+≡
```
static void upto_white_or_2(Bibreader rdr, char c1, char c2) {
  unsigned char *p = rdr->cur;
  unsigned char *lim = rdr->lim;
  *lim = c1;
  while (*p != c1 && *p != c2 && !isspace(*p))
    p++;
  rdr->cur = p;
}
```

Scan the reader up to the next whitespace or any of three characters requested.

⟨*function definitions*⟩+≡
```
static void upto_white_or_3(Bibreader rdr, char c1, char c2, char c3) {
  unsigned char *p = rdr->cur;
  unsigned char *lim = rdr->lim;
  *lim = c1;
  while (!isspace(*p) && *p != c1 && *p != c2 && *p != c3)
    p++;
  rdr->cur = p;
}
```

This function scans over whitespace characters, stopping either at the first nonwhite character or the end of the line, respectively returning `true` or `false`.

⟨*function definitions*⟩+≡
```
static bool upto_nonwhite(Bibreader rdr) {
  unsigned char *p = rdr->cur;
  unsigned char *lim = rdr->lim;
  *lim = 'x';
  while (isspace(*p))
    p++;
  rdr->cur = p;
  return p < lim;
}
```

Scan past whitespace up to end of file if needed; returns true iff nonwhite character found.

⟨*function definitions*⟩+≡

```
static int upto_nonwhite_get_line(Bibreader rdr) {
  while (!upto_nonwhite(rdr))
    if (!get_line(rdr))
      return 0;
  return 1;
}
```

2.3.3 Actual input

⟨*function definitions*⟩+≡

```
static bool get_line(Bibreader rdr) {
  char *result;
  unsigned char *buf = rdr->buf;
  int n;
  result = fgets((char *)buf, rdr->bufsize, rdr->file);
  if (result == NULL)
    return 0;
  rdr->line_num++;
  for (n = strlen((char *)buf); buf[n-1] != ’\n’; n = strlen((char *)buf)) {
    /* failed to get whole line */
    rdr->bufsize *= 2;
    buf = rdr->buf = realloc(rdr->buf, rdr->bufsize);
    assert(buf);
    if (fgets((char *)buf+n,rdr->bufsize-n,rdr->file)==NULL) {
      n = strlen((char *)buf) + 1;  /* -1 below is incorrect without newline */
      break; /* file ended without a newline */
    }
  }
  rdr->cur = buf;
  rdr->lim = buf+n-1;  /* trailing newline not in string */
  return 1;
}
```

2.3.4 Medium-level scanning functions

This procedure scans for an identifier, stopping at the first `illegal_id_char`, or stopping at the first character if it's `numeric`. It sets the global variable `scan_result` to `id_null` if the identifier is null, else to `white_adjacent` if it ended at a whitespace character or an end-of-line, else to `specified_char_adjacent` if it ended at one of `char1` or `char2` or `char3`, else to `other_char_adjacent` if it ended at a nonspecified, nonwhitespace `illegal_id_char`. By convention, when some calling code really wants just one or two "specified" characters, it merely repeats one of the characters.

⟨*Procedures and functions for input scanning*⟩+≡

```
static int scan_identifier (Bibreader rdr, char c1, char c2, char c3) {
  unsigned char *p, *orig, c;

  orig = p = rdr->cur;
  if (!isdigit(*p)) {
    /* scan until end-of-line or an [[illegal_id_char]] */
    *rdr->lim = ’ ’;  /* an illegal id character and also white space */
    while (is_id_char[*p])
      p++;
  }
  c = *p;
  if (p > rdr->cur && (isspace(c) || c == c1 || c == c2 || c == c3)) {
    rdr->cur = p;
    return 1;
  } else {
    return 0;
  }
}
```

This function scans for a nonnegative integer, stopping at the first nondigit; it writes the resulting integer through `np`. It returns `true` if the token was a legal nonnegative integer (i.e., consisted of one or more digits).

⟨*Procedures and functions for input scanning*⟩+≡

```
static bool scan_nonneg_integer (Bibreader rdr, unsigned *np) {
  unsigned char *p = rdr->cur;
  unsigned n = 0;
  *rdr->lim = ’ ’;   /* sentinel */
  while (isdigit(*p)) {
    n = n * 10 + (*p - ’0’);
    p++;
  }
  if (p == rdr->cur)
    return 0; /* no digits */
  else {
    rdr->cur = p;
    *np = n;
    return 1;
  }
}
```

This procedure scans for an integer, stopping at the first nondigit; it sets the value of `token_value` accordingly. It returns `true` if the token was a legal integer (i.e., consisted of an optional `minus_sign` followed by one or more digits).

⟨*unused Procedures and functions for input scanning*⟩≡
```
static bool scan_integer (Bibreader rdr) {
  unsigned char *p = rdr->cur;
  int n = 0;
  int sign = 0;      /* number of characters of sign */
  *rdr->lim = ' ';   /* sentinel */
  if (*p == '-') {
    sign = 1;
    p++;
  }
  while (isdigit(*p)) {
    n = n * 10 + (*p - '0');
    p++;
  }
  if (p == rdr->cur)
    return 0; /* no digits */
  else {
    rdr->cur = p;
    return 1;
  }
}
```

2.4 C utility functions

⟨*function definitions*⟩+≡
```
static void lower_case(unsigned char *p, unsigned char *lim) {
  for (; p < lim; p++)
    *p = tolower(*p);
}
```

⟨*function definitions*⟩+≡
```
static void strip_leading_and_trailing_space(lua_State *L) {
  const char *p;
  int n;
  assert(lua_isstring(L, -1));
  p = lua_tostring(L, -1);
  n = lua_strlen(L, -1);
  if (n > 0 && (isspace(*p) || isspace(p[n-1]))) {
    while(n > 0 && isspace(*p))
      p++, n--;
    while(n > 0 && isspace(p[n-1]))
      n--;
    lua_pushlstring(L, p, n);
    lua_remove(L, -2);
  }
}
```

2.5 Implementations of the BibTeX commands

On encountering an `@`*identifier*, we ask if the *identifier* stands for a command and if so, return that command.

⟨*function definitions*⟩+≡
```
static Command find_command(unsigned char *p, unsigned char *lim) {
  int n = lim - p;
  assert(lim > p);
#define match(S) (!strncmp(S, (char *)p, n) && (S)[n] == '\0')
  switch(*p) {
    case 'c' : if (match("comment"))  return do_comment;  else break;
    case 'p' : if (match("preamble")) return do_preamble; else break;
    case 's' : if (match("string"))   return do_string;   else break;
  }
  return (Command)0;
}
```

The `comment` command is implemented for SCRIBE compatibility. It's not really needed because BibTeX treats (flushes) everything not within an entry as a comment anyway.

⟨*function definitions*⟩+≡
```
static bool do_comment(Bibreader rdr) {
  return 1;
}
```

The `preamble` command lets a user have TeX stuff inserted (by the standard styles, at least) directly into the `.bbl` file. It is intended primarily for allowing TeX macro definitions used within the bibliography entries (for better sorting, for example). One `preamble` command per `.bib` file should suffice.

A `preamble` command has either braces or parentheses as outer delimiters. Inside is the preamble string, which has the same syntax as a field value: a nonempty list of field tokens separated by `concat_chars`. There are three types of field tokens—nonnegative numbers, macro names, and delimited strings.

This module does all the scanning (that's not subcontracted), but the `.bib`-specific scanning function `scan_and_push_the_field_value_and_eat_white` actually stores the value.

⟨*function definitions*⟩+≡
```
static bool do_preamble(Bibreader rdr) {
  ready_tok(rdr);
  ⟨scan past opening delimiter and set rdr->entry_close⟩
  ready_tok(rdr);
  lua_rawgeti(rdr->L, LUA_REGISTRYINDEX, rdr->preamble);
  lua_pushnumber(rdr->L, lua_objlen(rdr->L, -1) + 1);
  if (!scan_and_push_the_field_value(rdr, 0))
    return 0;
  ready_tok(rdr);
  if (*rdr->cur != rdr->entry_close)
    LERRFB("Missing '%c' in preamble command", rdr->entry_close);
  rdr->cur++;
  lua_settable(rdr->L, -3);
  lua_pop(rdr->L, 1); /* remove preamble */
  return 1;
}
```

The `string` command is implemented both for SCRIBE compatibility and for allowing a user: to override a `.bst`-file `macro` command, to define one that the `.bst` file doesn't, or to engage in good, wholesome, typing laziness.

The `string` command does mostly the same thing as the `.bst`-file's `macro` command (but the syntax is different and the `string` command compresses white space). In fact, later in this program, the term "macro" refers to either a `.bst` "macro" or a `.bib` "string" (when it's clear from the context that it's not a `WEB` macro).

A `string` command has either braces or parentheses as outer delimiters. Inside is the string's name (it must be a legal identifier, and case differences are ignored—all upper-case letters are converted to lower case), then an equals sign, and the string's definition, which has the same syntax as a field value: a nonempty list of field tokens separated by `concat_chars`. There are three types of field tokens—nonnegative numbers, macro names, and delimited strings.

⟨*function definitions*⟩+≡

```
  static bool do_string(Bibreader rdr) {
    unsigned char *id;
    int keyindex;
    ready_tok(rdr);
    ⟨scan past opening delimiter and set rdr->entry_close⟩
    ready_tok(rdr);
    id = rdr->cur;
    if (!scan_identifier(rdr, ’=’, ’=’, ’=’))
      LERRB("Expected a string name followed by ’=’");
    lower_case(id, rdr->cur);
    lua_pushlstring(rdr->L, (char *)id, rdr->cur - id);
    keyindex = lua_gettop(rdr->L);
    ready_tok(rdr);
    if (*rdr->cur != ’=’)
      LERRB("Expected a string name followed by ’=’");
    rdr->cur++;
    ready_tok(rdr);
    if (!scan_and_push_the_field_value(rdr, keyindex))
      return 0;
    ready_tok(rdr);
    if (*rdr->cur != rdr->entry_close)
      LERRFB("Missing ’%c’ in macro definition", rdr->entry_close);
    rdr->cur++;
    lua_getref(rdr->L, rdr->macros);
    lua_insert(rdr->L, -3);
    lua_settable(rdr->L, -3);
    lua_pop(rdr->L, 1);
    return 1;
  }
```

2.6 Interface to Lua

First, we define Lua access to a reader.

⟨*function definitions*⟩+≡

```
  static Bibreader checkreader(lua_State *L, int index) {
    return luaL_checkudata(L, index, "bibtex.reader");
  }
```

The reader's `__index` metamethod provides access to the `entry_line` and `preamble` values as if they were fields of the Lua table. It also provides access to the `next` and `close` methods of the reader object.

⟨*function definitions*⟩+≡

```
static int reader_meta_index(lua_State *L) {
  Bibreader rdr = checkreader(L, 1);
  const char *key;
  if (!lua_isstring(L, 2))
    return 0;
  key = lua_tostring(L, 2);
  if (!strcmp(key, "next"))
    lua_pushcfunction(L, next_entry);
  else if (!strcmp(key, "entry_line"))
    lua_pushnumber(L, rdr->entry_line);
  else if (!strcmp(key, "preamble"))
    lua_rawgeti(L, LUA_REGISTRYINDEX, rdr->preamble);
  else if (!strcmp(key, "close"))
    lua_pushcfunction(L, closereader);
  else
    lua_pushnil(L);
  return 1;
}
```

Here are the functions exported in the `bibtex` module:

⟨*function prototypes*⟩+≡

```
static int openreader(lua_State *L);
static int next_entry(lua_State *L);
static int closereader(lua_State *L);
```

⟨*initialized and uninitialized data*⟩+≡

```
static const struct luaL_reg bibtexlib [] = {
  {"open", openreader},
  {"close", closereader},
  {"next", next_entry},
  {NULL, NULL}
};
```

To create a reader, we call

`openreader(`*filename*, [*macro-table*, [*warn-function*]])

The warning function will be called in one of the following ways:

- warn(`"extra field"`, *file*, *line*, *citation-key*, *field-name*, *field-value*)
  Duplicate definition of a field in a single entry.
- warn(`"undefined macro"`, *file*, *line*, *citation-key*, *macro-name*)
  Use of an undefined macro.

⟨*function definitions*⟩+≡

```
#define INBUF 128  /* initial size of input buffer */
/* filename * macro table * warning function -> reader */
static int openreader(lua_State *L) {
  const char *filename = luaL_checkstring(L, 1);
  FILE *f = fopen(filename, "r");
  Bibreader rdr;
  if (!f) {
    lua_pushnil(L);
    lua_pushfstring(L, "Could not open file ’%s’", filename);
    return 2;
  }

  ⟨set items 2 and 3 on stack to hold macro table and optional warning function⟩

  rdr = lua_newuserdata(L, sizeof(*rdr));
  luaL_getmetatable(L, "bibtex.reader");
  lua_setmetatable(L, -2);

  rdr->line_num = 0;
  rdr->buf = rdr->cur = rdr->lim = malloc(INBUF);
  rdr->bufsize = INBUF;
  rdr->file = f;
  rdr->filename = malloc(lua_strlen(L, 1)+1);
  assert(rdr->filename);
  strncpy((char *)rdr->filename, filename, lua_strlen(L, 1)+1);
  rdr->L = L;
  lua_newtable(L);
  rdr->preamble = luaL_ref(L, LUA_REGISTRYINDEX);
  lua_pushvalue(L, 2);
  rdr->macros = luaL_ref(L, LUA_REGISTRYINDEX);
  lua_pushvalue(L, 3);
  rdr->warning = luaL_ref(L, LUA_REGISTRYINDEX);
  return 1;
}
```

⟨*set items 2 and 3 on stack to hold macro table and optional warning function*⟩≡

```
if (lua_type(L, 2) == LUA_TNONE)
  lua_newtable(L);

if (lua_type(L, 3) == LUA_TNONE)
  lua_pushnil(L);
else if (!lua_isfunction(L, 3))
  luaL_error(L, "Warning value to bibtex.open is not a function");
```

Reader method `next_entry` takes no parameters. On success it returns a triple (*type*, *key*, *field-table*). On error it returns (`false`, *message*). On end of file it returns nothing.

⟨*function definitions*⟩+≡
```
static int next_entry(lua_State *L) {
  Bibreader rdr = checkreader(L, 1);
  if (!rdr->file)
    luaL_error(L, "Tried to read from closed bibtex.reader");
  return get_bib_command_or_entry_and_process(rdr);
}
```

Closing a reader recovers its resources; the `file` field of a closed reader is `NULL`.

⟨*function definitions*⟩+≡
```
static int closereader(lua_State *L) {
  Bibreader rdr = checkreader(L, 1);
  if (!rdr->file)
    luaL_error(L, "Tried to close closed bibtex.reader");
  fclose(rdr->file);
  rdr->file = NULL;
  free(rdr->buf);
  rdr->buf = rdr->cur = rdr->lim = NULL;
  rdr->bufsize = 0;
  free((void*)rdr->filename);
  rdr->filename = NULL;
  rdr->L = NULL;
  luaL_unref(L, LUA_REGISTRYINDEX, rdr->preamble);
  rdr->preamble = 0;
  luaL_unref(L, LUA_REGISTRYINDEX, rdr->warning);
  rdr->warning = 0;
  luaL_unref(L, LUA_REGISTRYINDEX, rdr->macros);
  rdr->macros = 0;
  return 0;
}
```

To help implement the call to the warning function, we have `warnv`. If there is no warning function, we return the nubmer of nils specified by `nres`.

⟨*function definitions*⟩+≡

```
static void warnv(Bibreader rdr, int nres, const char *fmt, ...) {
  const char *p;
  va_list vl;

  lua_rawgeti(rdr->L, LUA_REGISTRYINDEX, rdr->warning);
  if (lua_isnil(rdr->L, -1)) {
    lua_pop(rdr->L, 1);
    while (nres-- > 0)
      lua_pushnil(rdr->L);
  } else {
    va_start(vl, fmt);
    for (p = fmt; *p; p++)
      switch (*p) {
        case 'f': lua_pushnumber(rdr->L, va_arg(vl, double)); break;
        case 'd': lua_pushnumber(rdr->L, va_arg(vl, int)); break;
        case 's': {
          const char *s = va_arg(vl, char *);
          if (s == NULL) lua_pushnil(rdr->L);
          else lua_pushstring(rdr->L, s);
          break;
        }
        default: luaL_error(rdr->L, "invalid parameter type %c", *p);
      }
    lua_call(rdr->L, p - fmt, nres);
    va_end(vl);
  }
}
```

Here's where the library is initialized. This is the only exported function in the whole file.

⟨*function definitions*⟩+≡

```
int luaopen_bibtex (lua_State *L) {
  luaL_newmetatable(L, "bibtex.reader");
  lua_pushstring(L, "__index");
  lua_pushcfunction(L, reader_meta_index);  /* pushes the index method */
  lua_settable(L, -3);  /* metatable.__index = metatable */

  luaL_register(L, "bibtex", bibtexlib);
  ⟨initialize the is_id_char table⟩
  return 1;
}
```

In an identifier, we can accept any printing character except the ones listed in the `nonids` string.

⟨*initialize the* `is_id_char` *table*⟩≡

```
{
  unsigned c;
  static unsigned char *nonids = (unsigned char *)"\"#%'(),={} \t\n\f";
  unsigned char *p;

  for (c = 0; c <= 0377; c++)
    is_id_char[c] = 1;
  for (c = 0; c <= 037; c++)
    is_id_char[c] = 0;
  for (p = nonids; *p; p++)
    is_id_char[*p] = 0;
}
```

2.7 Main function for the nbib commands

This code is the standalone main function for all the nbib commands.

⟨*nbibtex.c*⟩≡

```
#include <stdlib.h>
#include <stdio.h>

#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>

extern int luaopen_bibtex(lua_State *L);
extern int luaopen_boyer_moore (lua_State *L);

int main (int argc, char *argv[]) {
  int i, rc;
  lua_State *L = luaL_newstate();
  static const char* files[] = { SHARE "/bibtex.lua",  SHARE "/natbib.nbs" };

  #define OPEN(N) lua_pushcfunction(L, luaopen_ ## N); lua_call(L, 0, 0)
  OPEN(base); OPEN(io); OPEN(os); OPEN(package); OPEN(string); OPEN(table);
  OPEN(bibtex); OPEN(boyer_moore);

  for (i = 0; i < sizeof(files)/sizeof(files[0]); i++) {
    if (luaL_dofile(L, files[i])) {
      fprintf(stderr, "%s: error loading configuration file %s\n",
              argv[0], files[i]);
      exit(2);
    }
  }
  lua_pushstring(L, "bibtex");
  lua_gettable(L, LUA_GLOBALSINDEX);
  lua_pushstring(L, "main");
  lua_gettable(L, -2);
  lua_newtable(L);
  for (i = 0; i < argc; i++) {
    lua_pushnumber(L, i);
    lua_pushstring(L, argv[i]);
    lua_settable(L, -3);
  }
  rc = lua_pcall(L, 1, 0, 0);
  if (rc) {
    fprintf(stderr, "Call failed: %s\n", lua_tostring(L, -1));
    lua_pop(L, 1);
  }
  lua_close(L);
  return rc;
}
```

3 Implementation of nbibtex

From here out, everything is written in Lua (http://www.lua.org). The main module is `bibtex`, and style-file support is in the submodule `bibtex.bst`. Each has a `doc` submodule, which is intended as machine-readable documentation.

```
⟨bibtex.lua⟩≡
  ⟨if not already present, load the C code for the bibtex module⟩

  local config = config or { } --- may be defined by config process

  local workaround = {
    badbibs = true,  --- don’t look at bad .bib files that come with teTeX
  }
  local bst = { }
  bibtex.bst = bst

  bibtex.doc = { }
  bibtex.bst.doc = { }

  bibtex.doc.bst = ’# table of functions used to write style files’
```

Not much code is executed during startup, so the main issue is to manage declaration before use. I have a few forward declarations in ⟨*declarations of internal functions*⟩; otherwise, count only on “utility” functions being declared before “exported” ones.

```
⟨bibtex.lua⟩+≡
  local find = string.find
  ⟨declarations of internal functions⟩
  ⟨Lua utility functions⟩
  ⟨exported Lua functions⟩
  ⟨check constant values for consistency⟩

  return bibtex
```

The Lua code relies on the C code. How we get the C code depends on how `bibtex.lua` is used; there are two alternatives:

- In the distribution, `bibtex.lua` is loaded by the C code in Section 2.7, which defines the `bibtex` module.
- For standalone testing purposes, `bibtex.lua` can be loaded directly into an interactive Lua interpreter, in which case it loads the `bibtex` module as a shared library.

```
⟨if not already present, load the C code for the bibtex module⟩≡
  if not bibtex then
    require ’nrlib’
    nrlib.load ’bibtex’
  end
```

3.1 Error handling, warning messages, and logging

```
⟨Lua utility functions⟩≡
  local function printf (...) return io.stdout:write(string.format(...)) end
  local function eprintf(...) return io.stderr:write(string.format(...)) end
```

I have to figure out what to do about errors — the current code is bogus. Among other things, I should be setting error levels.

⟨*Lua utility functions*⟩+≡
```
local function bibwarnf (...) eprintf(...); eprintf('\n') end
local function biberrorf(...) eprintf(...); eprintf('\n') end
local function bibfatalf(...) eprintf(...); eprintf('\n'); os.exit(2) end
```

Logging? What logging?

⟨*Lua utility functions*⟩+≡
```
local function logf() end
```

3.1.1 Support for delayed warnings

Like classic BibTeX, NbibTeX typically warns only about entries that are actually used. This functionality is implemented by function hold_warning, which keeps warnings on ice until they are either returned by held_warnings or thrown away by drop_warning. The function emit_warning emits a warning message eagerly when called; it is used to issue warnings about entries we actually use, or if the -strict option is given, to issue every warning.

⟨*Lua utility functions*⟩+≡

```
local hold_warning  -- function suitable to pass to bibtex.open; holds
local emit_warning  -- function suitable to pass to bibtex.open; prints
local held_warnings -- returns nil or list of warnings since last call
local drop_warnings -- drops warnings

local extra_ok = { reffrom = true }
   -- set of fields about which we should not warn of duplicates

do
  local warnfuns = { }
  warnfuns["extra field"] =
    function(file, line, cite, field, newvalue)
      if not extra_ok[field] then
        bibwarnf("Warning--I’m ignoring %s’s extra \"%s\" field\n--line %d of file %s\n",
                 cite, field, line, file)
      end
    end

  warnfuns["undefined macro"] =
    function(file, line, cite, macro)
      bibwarnf("Warning--string name \"%s\" is undefined\n--line %d of file %s\n",
               macro, line, file)
    end

  function emit_warning(tag, ...)
    return assert(warnfuns[tag])(...)
  end

  local held
  function hold_warning(...)
    held = held or { }
    table.insert(held, { ... })
  end
  function held_warnings()
    local h = held
    held = nil
    return h
  end
  function drop_warnings()
    held = nil
  end
end
```

3.2 Miscellany

All this stuff is dubious.

⟨*Lua utility functions*⟩+≡

```
function table.copy(t)
  local u = { }
  for k, v in pairs(t) do u[k] = v end
  return u
end
```

⟨*Lua utility functions*⟩+≡

```
local function open(f, m, what)
  local f, msg = io.open(f, m)
  if f then
    return f
  else
    (what or bibfatalf)('Could not open file %s: %s', f, msg)
  end
end
```

⟨*exported Lua functions*⟩≡

```
local function entries(rdr, empty)
  assert(not empty)
  return function() return rdr:next() end
end

bibtex.entries = entries
bibtex.doc.entries = 'reader -> iterator   # generate entries'
```

3.3 Internal documentation

We attempt to document everything!

⟨*exported Lua functions*⟩+≡

```
function bibtex:show_doc(title)
  local out = bst.writer(io.stdout, 5)
  local function outf(...) return out:write(string.format(...)) end
  local allkeys, dkeys = { }, { }
  for k, _ in pairs(self)     do table.insert(allkeys, k) end
  for k, _ in pairs(self.doc) do table.insert(dkeys,   k) end
  table.sort(allkeys)
  table.sort(dkeys)
  for i = 1, table.getn(dkeys) do
    outf("%s.%-12s : %s\n", title, dkeys[i], self.doc[dkeys[i]])
  end
  local header
  for i = 1, table.getn(allkeys) do
    local k = allkeys[i]
    if k ~= "doc" and k ~= "show_doc" and not self.doc[k] then
      if not header then
        outf('Undocumented keys in table %s:', title)
        header = true
      end
      outf(' %s', k)
    end
  end
  if header then outf('\n') end
end
bibtex.bst.show_doc = bibtex.show_doc
```

Here is the documentation for what's defined in C code:

⟨*exported Lua functions*⟩+≡

```
bibtex.doc.open  = 'filename -> reader # open a reader for a .bib file'
bibtex.doc.close = 'reader -> unit     # close open reader'
bibtex.doc.next  = 'reader -> type * key * field table # read an entry'
```

3.4 Main function for nbibtex

Actually, the same main function does for both `nbibtex` and `nbibfind`; depending on how the program is called, it delegates to `bibtex.bibtex` or `bibtex.run_find`.

⟨*exported Lua functions*⟩+≡

```
bibtex.doc.main = 'string list -> unit # main program that dispatches on argv[0]'
function bibtex.main(argv)
  if argv[1] == '-doc' then -- undocumented internal doco
    bibtex:show_doc('bibtex')
    bibtex.bst:show_doc('bst')
  elseif find(argv[0], 'bibfind$') then
    return bibtex.run_find(argv)
  elseif find(argv[0], 'bibtex$') then
    return bibtex.bibtex(argv)
  else
    error("Call me something ending in 'bibtex' or 'bibfind'; when called\n  "..
          argv[0]..", I don't know what to do")
  end
end
```

⟨*exported Lua functions*⟩+≡

```
local permissive    = false -- nbibtex extension (ignore missing .bib files, etc.)
local strict        = false -- complain eagerly about errors in .bib files
local min_crossrefs = 2     -- how many crossref’s required to add an entry?
local output_name   = nil   -- output file if not default
local bib_out       = false -- output .bib format

bibtex.doc.bibtex = ’string list -> unit # main program for nbibtex’
function bibtex.bibtex(argv)
  ⟨set bibtex options from argv⟩
  if table.getn(argv) < 1 then
    bibfatalf(’Usage: %s [-permissive|-strict|...] filename[.aux] [bibfile...]’,
              argv[0])
  end
  local auxname = table.remove(argv, 1)
  local basename = string.gsub(string.gsub(auxname, ’%.aux$’, ’’), ’%.$’, ’’)
  auxname = basename .. ’.aux’
  local bblname = output_name or (basename .. ’.bbl’)
  local blgname = basename .. (output_name and ’.nlg’ or ’.blg’)
  local blg = open(blgname, ’w’)

  -- Here’s what we accumulate by reading .aux files:
  local bibstyle           -- the bibliography style
  local bibfiles = { }     -- list of files named in order of file
  local citekeys = { }     -- list of citation keys from .aux
                           -- (in order seen, mixed case, no duplicates)
  local cited_star = false -- .tex contains \cite{*} or \nocite{*}

  ⟨using file auxname, set bibstyle, citekeys, and bibfiles⟩

  if table.getn(argv) > 0 then -- override the bibfiles listed in the .aux file
    bibfiles = argv
  end
  ⟨validate contents of bibstyle, citekeys, and bibfiles⟩
  ⟨from bibstyle, citekeys, and bibfiles, compute and emit the list of entries⟩
  blg:close()
end
```

Options are straightforward.

⟨*set bibtex options from* `argv`⟩≡

```
while table.getn(argv) > 0 and find(argv[1], '^%-') do
  if argv[1] == '-terse' then
    -- do nothing
  elseif argv[1] == '-permissive' then
    permissive = true
  elseif argv[1] == '-strict' then
    strict = true
  elseif argv[1] == '-min-crossrefs' and find(argv[2], '^%d+$') then
    min_crossrefs = assert(tonumber(argv[2]))
    table.remove(argv, 1)
  elseif string.find(argv[1], '^%-min%-crossrefs=(%d+)$') then
    local _, _, n = string.find(argv[1], '^%-min%-crossrefs=(%d+)$')
    min_crossrefs = assert(tonumber(n))
  elseif string.find(argv[1], '^%-min%-crossrefs') then
    biberrorf("Ill-formed option %s", argv[1])
  elseif argv[1] == '-o' then
    output_name = assert(argv[2])
    table.remove(argv, 1)
  elseif argv[1] == '-bib' then
    bib_out = true
  elseif argv[1] == '-help' then
    help()
  elseif argv[1] == '-version' then
    printf("nbibtex version 0.9.18\n")
    os.exit(0)
  else
    biberrorf('Unknown option %s', argv[1])
    help(2)
  end
  table.remove(argv, 1)
end
```

⟨*Lua utility functions*⟩+≡

```
local function help(code)
  printf([[
Usage: nbibtex [OPTION]... AUXFILE[.aux] [BIBFILE...]
  Write bibliography for entries in AUXFILE to AUXFILE.bbl.

Options:
  -bib                   write output as BibTeX source
  -help                  display this help and exit
  -o FILE                write output to FILE (- for stdout)
  -min-crossrefs=NUMBER  include item after NUMBER cross-refs; default 2
  -permissive            allow missing bibfiles and (some) duplicate entries
  -strict                complain about any ill-formed entry we see
  -version               output version information and exit

Home page at http://www.eecs.harvard.edu/~nr/nbibtex.
Email bug reports to nr@eecs.harvard.edu.
]])
  os.exit(code or 0)
end
```

3.5 Reading all the aux files and validating the inputs

We pay attention to four commands: `\@input`, `\bibdata`, `\bibstyle`, and `\citation`.

⟨*using file* `auxname`*, set* `bibstyle`*,* `citekeys`*, and* `bibfiles`⟩≡

```
do
  local commands = { } -- table of commands we recognize in .aux files
  local function do_nothing() end -- default for unrecognized commands
  setmetatable(commands, { __index = function() return do_nothing end })
  ⟨functions for commands found in .aux files⟩
  commands['@input'](auxname)  -- reads all the variables
end
```

⟨*functions for commands found in .aux files*⟩≡

```
do
  local auxopened = { }  --- map filename to true/false

  commands['@input'] = function (auxname)
    if not find(auxname, '%.aux$') then
      bibwarnf('Name of auxfile "%s" does not end in .aux\n', auxname)
    end
    ⟨mark auxname as opened (but fail if opened already)⟩
    local aux = open(auxname, 'r')
    logf('Top-level aux file: %s\n', auxname)
    for line in aux:lines() do
      local _, _, cmd, arg = find(line, '^\\([%a%@]+)%s*{([^%}]+)}%s*$')
      if cmd then commands[cmd](arg) end
    end
    aux:close()
  end
end
```

⟨*mark* `auxname` *as opened (but fail if opened already)*⟩≡

```
if auxopened[auxname] then
  error("File " .. auxname .. " cyclically \\@input's itself")
else
  auxopened[auxname] = true
end
```

BibTeX expects `.bib` files to be separated by commas. They are forced to lower case, should have no spaces in them, and the `\bibdata` command should appear exactly once.

⟨*functions for commands found in .aux files*⟩+≡

```
do
  local bibdata_seen = false

  function commands.bibdata(arg)
    assert(not bibdata_seen, [[LaTeX provides multiple \bibdata commands]])
    bibdata_seen = true
    for bib in string.gmatch(arg, '[^,]+') do
      assert(not find(bib, '%s'), 'bibname from LaTeX contains whitespace')
      table.insert(bibfiles, string.lower(bib))
    end
  end
end
```

The style should be unique, and it should be known to us.

⟨*functions for commands found in .aux files*⟩+≡

```
function commands.bibstyle(stylename)
  if bibstyle then
    biberrorf(’Illegal, another \\bibstyle command’)
  else
    bibstyle = bibtex.style(string.lower(stylename))
    if not bibstyle then
      bibfatalf(’There is no nbibtex style called "%s"’)
    end
  end
end
```

We accumulated cited keys in `citekeys`. Keys may be duplicated, but the input should not contain two keys that differ only in case.

⟨*functions for commands found in .aux files*⟩+≡

```
do
  local keys_seen, lower_seen = { }, { } -- which keys have been seen already

  function commands.citation(arg)
    for key in string.gmatch(arg, ’[^,]+’) do
      assert(not find(key, ’%s’),
             ’Citation key {’ .. key .. ’} from LaTeX contains whitespace’)
      if key == ’*’ then
        cited_star = true
      elseif not keys_seen[key] then --- duplicates are OK
        keys_seen[key] = true
        local low = string.lower(key)
        ⟨if another key with same lowercase, complain bitterly⟩
        if not cited_star then -- no more insertions after the star
          table.insert(citekeys, key) -- must be key, not low,
                                      -- so that keys in .bbl match .aux
        end
      end
    end
  end
end
```

⟨*if another key with same lowercase, complain bitterly*⟩≡

```
if lower_seen[low] then
  biberrorf("Citation key ’%s’ inconsistent with earlier key ’%s’",
            key, lower_seen[low])
else
  lower_seen[low] = key
end
```

After reading the variables, we do a little validation. I can't seem to make up my mind what should be done incrementally while things are being read.

⟨*validate contents of* `bibstyle`*,* `citekeys`*, and* `bibfiles`⟩≡

```
if not bibstyle then
  bibfatalf('No \\bibliographystyle in original LaTeX')
end

if table.getn(bibfiles) == 0 then
  bibfatalf('No .bib files specified --- no \\bibliography in original LaTeX?')
end

if table.getn(citekeys) == 0 and not cited_star then
  biberrorf('No citations in document --- empty bibliography')
end

do --- check for duplicate bib entries
  local i = 1
  local seen = { }
  while i <= table.getn(bibfiles) do
    local bib = bibfiles[i]
    if seen[bib] then
      bibwarnf('Multiple references to bibfile "%s"', bib)
      table.remove(bibfiles, i)
    else
      i = i + 1
    end
  end
end
```

3.6 Reading the entries from all the BibTeX files

These are diagnostics that might be written to a log.

⟨*from* `bibstyle`*,* `citekeys`*, and* `bibfiles`*, compute and emit the list of entries*⟩≡

```
logf("bibstyle == %q\n", bibstyle.name)
logf("consult these bibfiles:")
for _, bib in ipairs(bibfiles) do logf(" %s", bib) end
logf("\ncite these papers:\n")
for _, key in ipairs(citekeys) do logf("  %s\n", key) end
if cited_star then logf("  and everything else in the database\n") end
```

Each bibliography file is opened with `openbib`. Unlike classic BibTeX, we can't simply select the first entry matching a citation key. Instead, we read all entries into `bibentries` and do searches later.

The easy case is when we're not permissive: we put all the entries into one list, just as if they had come from a single `.bib` file. But if we're permissive, duplicates in different bibfiles are OK: we will search one bibfile after another and stop after the first successful search—thus instead of a single list, we have a list of lists.

⟨*from* `bibstyle`, `citekeys`, *and* `bibfiles`, *compute and emit the list of entries*⟩+≡

```
  local bibentries = { } -- if permissive, list of lists, else list of entries
  local dupcheck = { } -- maps lower key to entry
  local preamble = { } -- accumulates preambles from all .bib files
  local got_one_bib = false -- did we open even one .bib file?
  ⟨definition of function openbib, which sets get_one_bib if successful⟩

  local warnings = { }  -- table of held warnings for each entry
  local macros = bibstyle.macros() -- must accumulate macros across .bib files
  for _, bib in ipairs(bibfiles) do
    local bibfilename, rdr = openbib(bib, macros)
    if rdr then
      local t -- list that will receive entries from this reader
      if permissive then
        t = { }
        table.insert(bibentries, t)
      else
        t = bibentries
      end
      local localdupcheck = { } -- lower key to entry; finds duplicates within this file
      for type, key, fields, file, line in entries(rdr) do
        if type == nil then
          break
        elseif type then -- got something without error
          local e = { type = type, key = key, fields = fields,
                      file = bibfilename, line = rdr.entry_line }
          warnings[e] = held_warnings()
          ⟨definition of local function not_dup⟩
          local ok1, ok2 = not_dup(localdupcheck), not_dup(dupcheck) -- evaluate both
          if ok1 and ok2 then
            table.insert(t, e)
          end
        end
      end
      for _, l in ipairs(rdr.preamble) do table.insert(preamble, l) end
      rdr:close()
    end
  end

  if not got_one_bib then
    bibfatalf("Could not open any of the following .bib files: %s",
              table.concat(bibfiles, ' '))
  end
```

Because the preamble is accumulated as the `.bib` file is read, it must be copied at the end.

Here we open files. If we're not being permissive, we must open each file successfully. If we're permissive, it's enough to get at least one.

To find the pathname for a bib file, we use `bibtex.bibpath`.

⟨*definition of function* `openbib`*, which sets* `get_one_bib` *if successful*⟩≡

```
local function openbib(bib, macros)
  macros = macros or bibstyle.macros()
  local filename, msg = bibtex.bibpath(bib)
  if not filename then
    if not permissive then biberrorf("Cannot find file %s.bib", bib) end
    return
  end
  local rdr = bibtex.open(filename, macros, strict and emit_warning or hold_warning)
  if not rdr and not permissive then
    biberrorf("Cannot open file %s.bib", bib)
    return
  end
  got_one_bib = true
  return filename, rdr
end
```

3.6.1 Duplication checks

There’s a great deal of nuisance to checking the integrity of a `.bib` file.

⟨*definition of local function* `not_dup`⟩≡

```
⟨abstraction exporting savecomplaint and issuecomplaints⟩

local k = string.lower(key)
local function not_dup(dup)
  local e1, e2 = dup[k], e
  if e1 then
    -- do return false end --- avoid extra msgs for now
    local diff = entries_differ(e1, e2)
    if diff then
      local verybad = not permissive or e1.file == e2.file
      local complain = verybad and biberrorf or bibwarnf
      if e1.key == e2.key then
        if verybad then
          savecomplaint(e1, e2, complain,
                        "Ignoring second entry with key ’%s’ on file %s, line %d\n" ..
                        "  (first entry occurred on file %s, line %d;\n"..
                        "   entries differ in %s)\n",
                        e2.key, e2.file, e2.line, e1.file, e1.line, diff)
        end
      else
        savecomplaint(e1, e2, complain,
           "Entries ’%s’ on file %s, line %d and\n  ’%s’ on file %s, line %d" ..
           " have keys that differ only in case\n",
           e1.key, e1.file, e1.line, e2.key, e2.file, e2.line)
      end
    elseif e1.file == e2.file then
      savecomplaint(e1, e2, bibwarnf,
        "Entry ’%s’ is duplicated in file ’%s’ at both line %d and line %d\n",
        e1.key, e1.file, e1.line, e2.line)
    elseif not permissive then
      savecomplaint(e1, e2, bibwarnf,
        "Entry ’%s’ appears both on file ’%s’, line %d and file ’%s’, line %d"..
        "\n  (entries are exact duplicates)\n",
        e1.key, e1.file, e1.line, e2.file, e2.line)
    end
    return false
  else
    dup[k] = e
    return true
  end
end
```

Calling `savecomplaint(e1, e2, complain, ...)` takes the complaint `complain(...)` and associates it with entries `e1` and `e2`. If we are operating in “strict” mode, the complaint is issued right away; otherwise calling `issuecomplaints(e)` issues the complaint lazily. In non-strict, lazy mode, the outside world arranges to issue only complaints with entries that are actually used.

⟨*abstraction exporting* `savecomplaint` *and* `issuecomplaints`⟩≡

```
local savecomplained, issuecomplaints
if strict then
  function savecomplaint(e1, e2, complain, ...)
    return complain(...)
  end
  function issuecomplaints(e) end
else
  local complaints = { }
  local function save(e, t)
    complaints[e] = complaints[e] or { }
    table.insert(complaints[e], t)
  end
  function savecomplaint(e1, e2, ...)
    save(e1, { ... })
    save(e2, { ... })
  end
  local function call(c, ...)
    return c(...)
  end
  function issuecomplaints(e)
    for _, c in ipairs(complaints[e] or { }) do
      call(unpack(c))
    end
  end
end
```

⟨*Lua utility functions*⟩+≡

```
-- return ’key’ or ’type’ or ’field <name>’ at which entries differ,
-- or nil if entries are the same
local function entries_differ(e1, e2, notkey)
  if e1.key  ~= e2.key  and not notkey then return ’key’  end
  if e1.type ~= e2.type                then return ’type’ end
  for k, v in pairs(e1.fields) do
    if e2.fields[k] ~= v then return ’field ’ .. k end
  end
  for k, v in pairs(e2.fields) do
    if e1.fields[k] ~= v then return ’field ’ .. k end
  end
end
```

I’ve seen at least one bibliography with identical entries listed under multiple keys. (Thanks, Andrew.)

⟨*Lua utility functions*⟩+≡

```
-- every entry is identical to every other
local function all_entries_identical(es, notkey)
  if table.getn(es) == 0 then return true end
  for i = 2, table.getn(es) do
    if entries_differ(es[1], es[i], notkey) then
      return false
    end
  end
  return true
end
```

3.7 Computing and emitting the list of citations

A significant complexity added in NbibTeX is that a single entry may be cited using more than one citation key. For example, `\cite{milner:type-polymorphism}` and `\cite{milner:theory-polymorphism}` may well specify the same paper. Thus, in addition to a list of citations, I also keep track of the set of keys with which each entry is cited, as well as the first such key. The function `cite` manages all these data structures.

⟨*from* `bibstyle`*,* `citekeys`*, and* `bibfiles`*, compute and emit the list of entries*⟩+≡

```
local citations = { } -- list of citations
local cited = { } -- (entry -> key set) table
local first_cited = { } -- (entry -> key) table
local function cite(c, e) -- cite entry e with key c
  local seen = cited[e]
  cited[e] = seen or { }
  cited[e][c] = true
  if not seen then
    first_cited[e] = c
    table.insert(citations, e)
  end
end
```

When the dust settles, we adjust members of each citation record: the first key actually used becomes `key`, the original key becomes `orig_key`, and other keys go into `also_cited_as`.

⟨*using* `cited` *and* `first_cited`*, adjust fields* `key` *and* `also_cited_as`⟩≡

```
for i = 1, table.getn(citations) do
  local c = citations[i]
  local key = assert(first_cited[c], "citation is not cited?!")
  c.orig_key, c.key = c.key, key
  local also = { }
  for k in pairs(cited[c]) do
    if k ~= key then table.insert(also, k) end
  end
  c.also_cited_as = also
end
```

For each actual `\cite` command in the original LaTeX file, we call `find_entry` to find an appropriate BibTeX entry. Because a `\cite` command might match more than one paper, the results may be ambiguous. We therefore produce a list of all *candidates* matching the `\cite` command. If we're permissive, we search one list of entries after another, stopping as soon as we get some candidates. If we're not permissive, we have just one list of entries overall, so we search it and we're done. If permissive, we search entry lists in turn until we

⟨*from* `bibstyle`*,* `citekeys`*, and* `bibfiles`*, compute and emit the list of entries*⟩+≡

```
local find_entry -- function from key to citation
do
  local cache = { } -- (citation-key -> entry) table

  function find_entry(c)
    local function remember(e) cache[c] = e; return e end -- cache e and return it

    if cache[c] or dupcheck[c] then
      return cache[c] or dupcheck[c]
    else
      local candidates
      if permissive then
        for _, entries in ipairs(bibentries) do
          candidates = query(c, entries)
          if table.getn(candidates) > 0 then break end
        end
      else
        candidates = query(c, bibentries)
      end
      assert(candidates)
      ⟨from the available candidates, choose one and remember it⟩
    end
  end
end
```

If we have no candidates, we're hosed. Otherwise, if all the candidates are identical (most likely when there is a unique candidate, but still possible otherwise),[2] we take the first. Finally, if there are multiple, distinct candidates to choose from, we take the first and issue a warning message. To avoid surprising the unwary coauthor, we put a warning message into the entry as well, from which it will go into the printed bibliography.

⟨*from the available* `candidates`*, choose one and* `remember` *it*⟩≡

```
if table.getn(candidates) == 0 then
  biberrorf('No .bib entry matches \\cite{%s}', c)
elseif all_entries_identical(candidates, 'notkey') then
  logf("Query '%s' produced unique candidate %s from %s\n",
          c, candidates[1].key, candidates[1].file)
  return remember(candidates[1])
else
  local e = table.copy(candidates[1])
  ⟨warn of multiple candidates for query c⟩
  e.warningmsg = string.format('[This entry is the first match for query ' ..
                               '\\texttt{%s}, which produced %d matches.]',
                               c, table.getn(candidates))
  return remember(e)
end
```

[2] Andrew Appel has a bibliography in which the *Definition of Standard ML* appears as two different entries that are identical except for keys.

I can do better later…

⟨*warn of multiple candidates for query* `c`⟩≡

```
bibwarnf("Query '%s' produced %d candidates\n    (using %s from %s)\n",
         c, table.getn(candidates), e.key, e.file)
bibwarnf("First two differ in %s\n", entries_differ(candidates[1], candidates[2], true))
```

The `query` function uses the engine described in Section 3.9.

⟨*definition of* `query`*, used to search a list of entries*⟩≡

```
function query(c, entries)
  local p = matchq(c)
  local t = { }
  for _, e in ipairs(entries) do
    if p(e.type, e.fields) then
      table.insert(t, e)
    end
  end
  return t
end
bibtex.query = query
bibtex.doc.query = 'query: string -> entry list -> entry list'
```

⟨*declarations of internal functions*⟩≡

```
local query
local matchq
bibtex.doc.matchq = 'matchq: string -> predicate --- compile query string'
bibtex.matchq = matchq
```

Finally we can compute the list of entries: search on each citation key, and if we had `\cite{*}` or `\nocite{*}`, add all the other entries as well. The `cite` command takes care of avoiding duplicates.

⟨*from* `bibstyle`*,* `citekeys`*, and* `bibfiles`*, compute and emit the list of entries*⟩+≡

```
for _, c in ipairs(citekeys) do
  local e = find_entry(c)
  if e then cite(c, e) end
end
if cited_star then
  for _, es in ipairs(permissive and bibentries or {bibentries}) do
    logf('Adding all entries in list of %d\n', table.getn(es))
    for _, e in ipairs(es) do
      cite(e.key, e)
    end
  end
end
⟨using cited and first_cited, adjust fields key and also_cited_as⟩
```

I've always hated BibTeX's cross-reference feature, but I believe I've implemented it faithfully.

⟨*from* `bibstyle`*,* `citekeys`*, and* `bibfiles`*, compute and emit the list of entries*⟩+≡

```
bibtex.do_crossrefs(citations, find_entry)
```

With the entries computed, there are two ways to emit: as another BibTeX file or as required by the style file. So that we can read from `bblname` before writing to it, the opening of `bbl` is carefully delayed to this point.

⟨*from* `bibstyle`*,* `citekeys`*, and* `bibfiles`*, compute and emit the list of entries*⟩+≡

```
  ⟨emit warnings for entries in citations⟩
  local bbl = bblname == '-' and io.stdout or open(bblname, 'w')
  if bib_out then
    bibtex.emit(bbl, preamble, citations)
  else
    bibstyle.emit(bbl, preamble, citations)
  end
  if bblname ~= '-' then bbl:close() end
```

Here's a function to emit a list of citations as BibTeX source.

⟨*exported Lua functions*⟩+≡

```
  bibtex.doc.emit =
    'outfile * string list * entry list -> unit -- write citations in .bib format'
  function bibtex.emit(bbl, preamble, citations)
    local warned = false
    if preamble[1] then
      bbl:write('@preamble{\n')
      for i = 1, table.getn(preamble) do
        bbl:write(string.format('  %s "%s"\n', i > 1 and '#' or ' ', preamble[i]))
      end
      bbl:write('}\n\n')
    end
    for _, e in ipairs(citations) do
      local also = e.also_cited_as
      if also and table.getn(also) > 0 then
        for _, k in ipairs(e.also_cited_as or { }) do
          bbl:write(string.format('@%s{%s, crossref={%s}}\n', e.type, k, e.key))
        end
        if not warned then
          warned = true
          bibwarnf("Warning: some entries (such as %s) are cited with multiple keys;\n"..
            "  in the emitted .bib file, these entries are duplicated (using crossref)\n",
                   e.key)
        end
      end
      emit_tkf.bib(bbl, e.type, e.key, e.fields)
    end
  end
```

⟨*emit warnings for entries in* `citations`⟩≡

```
  for _, e in ipairs(citations) do
    if warnings[e] then
      for _, w in ipairs(warnings[e]) do emit_warning(unpack(w)) end
    end
  end
```

3.8 Cross-reference

If an entry contains a `crossref` field, that field is used as a key to find the parent, and the entry inherits missing fields from the parent.

If the parent is cross-referenced sufficiently often (i.e., more than `min_crossref` times), it may be added to the citation list, in which case the style file knows what to do with the `crossref` field. But if the parent is not cited sufficiently often, it disappears, and do does the `crossref` field.

⟨*exported Lua functions*⟩+≡

```
bibtex.doc.do_crossrefs = "citation list -> unit # add crossref'ed fields in place"
function bibtex.do_crossrefs(citations, find_entry)
  local map  = { } --- key to entry (on citation list)
  local xmap = { } --- key to entry (xref'd only)
  local xref_count = { } -- entry -> number of times xref'd
  ⟨make map map lower-case keys in citations to their entries⟩
  for i = 1, table.getn(citations) do
    local c = citations[i]
    if c.fields.crossref then
      local lowref = string.lower(c.fields.crossref)
      local parent = map[lowref] or xmap[lowref]
      if not parent and find_entry then
        parent = find_entry(lowref)
        xmap[lowref] = parent
      end
      if not parent then
        biberrorf("Entry %s cross-references to %s, but I can't find %s",
                  c.key, c.fields.crossref, c.fields.crossref)
        c.fields.crossref = nil
      else
        xref_count[parent] = (xref_count[parent] or 0) + 1
        local fields = c.fields
        fields.crossref = parent.key -- force a case match!
        for k, v in pairs(parent.fields) do -- inherit field if missing
          fields[k] = fields[k] or v
        end
      end
    end
  end
  ⟨add oft-crossref'd entries from xmap to the list in citations⟩
  ⟨remove crossref fields for entries with seldom-crossref'd parents⟩
end
```

⟨*make* `map` *map lower-case keys in* `citations` *to their entries*⟩≡

```
for i = 1, table.getn(citations) do
  local c = citations[i]
  local key = string.lower(c.key)
  map[key] = map[key] or c
end
```

⟨*add oft-crossref'd entries from* `xmap` *to the list in* `citations`⟩≡

```
for _, e in pairs(xmap) do -- includes only missing entries
  if xref_count[e] >= min_crossrefs then
    table.insert(citations, e)
  end
end
```

⟨*remove* `crossref` *fields for entries with seldom-crossref'd parents*⟩≡

```
for i = 1, table.getn(citations) do
  local c = citations[i]
  if c.fields.crossref then
    local parent = xmap[string.lower(c.fields.crossref)]
    if parent and xref_count[parent] < min_crossrefs then
      c.fields.crossref = nil
    end
  end
end
```

3.9 The query engine (i.e., the point of it all)

The query language is described in the man page for `nbibtex`. Its implementation is divided into two parts: the internal predicates which are composed to form a query predicate, and the parser that takes a string and produces a query predicate. Function `matchq` is declared `local` above and is the only function visible outside this block.

⟨*exported Lua functions*⟩+≡

```
do
  if not boyer_moore then
    require ’boyer-moore’
  end
  local bm = boyer_moore
  local compile = bm.compilenc
  local search  = bm.matchnc

  -- type predicate = type * field table -> bool
  -- val match   : field * string -> predicate
  -- val author  : string -> predicate
  -- val matchty : string -> predicate
  -- val andp    : predicate option * predicate option -> predicate option
  -- val orp     : predicate option * predicate option -> predicate option
  -- val matchq  : string -> predicate --- compile query string

  ⟨definitions of query-predicate functions⟩

  ⟨definition of matchq, the query compiler⟩
  ⟨definition of query, used to search a list of entries⟩
end
```

3.9.1 Query predicates

The common case is a predicate for a named field. We also have some special syntax for "all fields" and the BibTeX "type," which is not a field.

⟨*definitions of query-predicate functions*⟩≡

```
local matchty
local function match(field, string)
  if string == '' then return nil end
  local pat = compile(string)
  if field == '*' then
    return function (t, fields)
             for _, v in pairs(fields) do if search(pat, v) then return true end end
           end
  elseif field == '[type]' then
    return matchty(string)
  else
    return function (t, fields) return search(pat, fields[field] or '') end
  end
end
```

Here's a type matcher.

⟨*definitions of query-predicate functions*⟩+≡

```
function matchty(string)
  if string == '' then return nil end
  local pat = compile(string)
  return function (t, fields) return search(pat, t) end
end
```

We make a special case of `author` because it really means "author or editor."

⟨*definitions of query-predicate functions*⟩+≡

```
local function author(string)
  if string == '' then return nil end
  local pat = compile(string)
  return function (t, fields)
           return search(pat, fields.author or fields.editor or '')
         end
end
```

We conjoin and disjoin predicates, being careful to use tail calls (not `and` and `or`) in order to save stack space.

⟨*definitions of query-predicate functions*⟩+≡

```
local function andp(p, q)
  -- associate to right for constant stack space
  if not p then
    return q
  elseif not q then
    return p
  else
    return function (t,f) if p(t,f) then return q(t,f) end end
  end
end
```

⟨*definitions of query-predicate functions*⟩+≡

```
local function orp(p, q)
  -- associate to right for constant stack space
  if not p then
    return q
  elseif not q then
    return p
  else
    return function (t,f) if p(t,f) then return true else return q(t,f) end end
  end
end
```

3.9.2 The query compiler

The function `matchq` takes the syntax explained in the man page and produces a predicate.

⟨*definition of* `matchq`*, the query compiler*⟩≡

```
function matchq(query)
  local find = string.find
  local parts = split(query, ’%:’)
  local p = nil
  if parts[1] and not find(parts[1], ’=’) then
    ⟨add to p a match for parts[1] as author⟩
    table.remove(parts, 1)
    if parts[1] and not find(parts[1], ’=’) then
      ⟨add to p a match for parts[1] as title or year⟩
      table.remove(parts, 1)
      if parts[1] and not find(parts[1], ’=’) then
        ⟨add to p a match for parts[1] as type or year⟩
        table.remove(parts, 1)
      end
    end
  end
  for _, part in ipairs(parts) do
    if not find(part, ’=’) then
      biberrorf(’bad query %q --- late specs need = sign’, query)
    else
      local _, _, field, words = find(part, ’ˆ(.*)=(.*)$’)
      assert(field and words, ’bug in query parsing’)
      ⟨add to p a match for words as field⟩
    end
  end
  if not p then
    bibwarnf(’empty query---matches everything\n’)
    return function() return true end
  else
    return p
  end
end
```

Here’s where an unnamed key defaults to author or editor.

⟨*add to* `p` *a match for* `parts[1]` *as author*⟩≡

```
for _, word in ipairs(split(parts[1], ’-’)) do
  p = andp(author(word), p)
end
```

⟨*add to* `p` *a match for* `parts[1]` *as title or year*⟩≡

```
local field, words = find(parts[1], ’%D’) and ’title’ or ’year’, parts[1]
⟨add to p a match for words as field⟩
```

⟨*add to* `p` *a match for* `parts[1]` *as type or year*⟩≡

```
if find(parts[1], ’%D’) then
  local ty = nil
  for _, word in ipairs(split(parts[1], ’-’)) do
    ty = orp(matchty(word), ty)
  end
  p = andp(p, ty) --- check type last for efficiency
else
  for _, word in ipairs(split(parts[1], ’-’)) do
    p = andp(p, match(’year’, word)) -- check year last for efficiency
  end
end
```

There could be lots of matches on a year, so we check years last.

⟨*add to* `p` *a match for* `words` *as* `field`⟩≡

```
for _, word in ipairs(split(words, ’-’)) do
  if field == ’year’ then
    p = andp(p, match(field, word))
  else
    p = andp(match(field, word), p)
  end
end
```

3.10 Path search and other system-dependent stuff

To find a bib file, I rely on the `kpsewhich` program, which is typically found on Unix TEX installations, and which should guarantee to find the same bib files as normal bibtex.

⟨*Lua utility functions*⟩+≡

```
assert(io.popen)
local function capture(cmd, raw)
  local f = assert(io.popen(cmd, ’r’))
  local s = assert(f:read(’*a’))
  assert(f:close())  --- can’t get an exit code
  if raw then return s end
  s = string.gsub(s, ’^%s+’, ’’)
  s = string.gsub(s, ’%s+$’, ’’)
  s = string.gsub(s, ’[\n\r]+’, ’ ’)
  return s
end
```

Function `bibpath` is normally called on a bibname in a LaTeX file, but because a bibname may also be given on the command line, we add `.bib` only if not already present. Also, because we can

⟨*exported Lua functions*⟩+≡

```
bibtex.doc.bibpath = 'string -> string # from \\bibliography name, find pathname of file'
function bibtex.bibpath(bib)
  if find(bib, '/') then
    local f, msg = io.open(bib)
    if not f then
      return nil, msg
    else
      f:close()
      return bib
    end
  else
    if not find(bib, '%.bib$') then
      bib = bib .. '.bib'
    end
    local pathname = capture('kpsewhich ' .. bib)
    if string.len(pathname) > 1 then
      return pathname
    else
      return nil, 'kpsewhich cannot find ' .. bib
    end
  end
end
```

4 Implementation of `nbibfind`

4.1 Output formats for BibTeX entries

We can emit a BibTeX entry in any of three formats: `bib`, `terse`, and `full`. An emitter takes as arguments the type, key, and fields of the entry, and optionally the name of the file the entry came from.

⟨*Lua utility functions*⟩+≡

```
local emit_tkf = { }
```

The simplest entry is legitimate BibTeX source:

⟨*exported Lua functions*⟩+≡

```
function emit_tkf.bib(outfile, type, key, fields)
  outfile:write('@', type, '{', key, ',\n')
  for k, v in pairs(fields) do
    outfile:write('  ', k, ' = {', v, '},\n')
  end
  outfile:write('}\n\n')
end
```

For the other two entries, we devise a string format. In principle, we could go with an ASCII form of a full-blown style, but since the purpose is to identify the entry in relatively few characters, it seems sufficient to spit out the author, year, title, and possibly the source. “Full” output shows the whole string; “terse” is just the first line.

⟨*exported Lua functions*⟩+≡

```
do
  local function bibstring(type, key, fields, bib)
    ⟨define local format_lab_names as for a bibliography label⟩
    local names = format_lab_names(fields.author) or
                  format_lab_names(fields.editor) or
                  fields.key or fields.organization or '????'
    local year = fields.year
    local lbl = names .. (year and ' ' .. year or '')
    local title = fields.title or '????'
    if bib then
      key = string.gsub(bib, '.*/', '') .. ': ' .. key
    end
    local answer =
      bib and
      string.format('%-25s = %s: %s', key, lbl, title) or
      string.format('%-21s = %s: %s', key, lbl, title)
    local where = fields.booktitle or fields.journal
    if where then answer = answer .. ', in ' .. where end
    answer = string.gsub(answer, '%~', ' ')
    for _, cs in ipairs { 'texttt', 'emph', 'textrm', 'textup' } do
      answer = string.gsub(answer, '\\' .. cs .. '%A', '')
    end
    answer = string.gsub(answer, '[%{%}]', '')
    return answer
  end

  function emit_tkf.terse(outfile, type, key, fields, bib)
    outfile:write(truncate(bibstring(type, key, fields, bib), 80), '\n')
  end

  function emit_tkf.full(outfile, type, key, fields, bib)
    local w = bst.writer(outfile)
    w:write(bibstring(type, key, fields, bib), '\n')
  end
end
```

⟨*define local* `format_lab_names` *as for a bibliography label*⟩≡

```
local format_lab_names
do
  local fmt = '{vv }{ll}'
  local function format_names(s)
    local s = bst.commafy(bst.format_names(fmt, bst.namesplit(s)))
    return (string.gsub(s, ' and others$', ' et al.'))
  end
  function format_lab_names(s)
    if not s then return s end
    local t = bst.namesplit(s)
    if table.getn(t) > 3 then
      return bst.format_name(fmt, t[1]) .. ' et al.'
    else
      return format_names(s)
    end
  end
end
```

Function `truncate` returns enough of a string to fit in `n` columns, with ellipses as needed.

⟨*Lua utility functions*⟩+≡

```
local function truncate(s, n)
  local l = string.len(s)
  if l <= n then
    return s
  else
    return string.sub(s, 1, n-3) .. '...'
  end
end
```

4.2 Main functions for nbibfind

⟨*exported Lua functions*⟩+≡

```
bibtex.doc.run_find = 'string list -> unit # main program for nbibfind'
bibtex.doc.find = 'string * string list -> entry list'

function bibtex.find(pattern, bibs)
  local es = { }
  local p = matchq(pattern)
  for _, bib in ipairs(bibs) do
    local rdr = bibtex.open(bib, bst.months(), hold_warning)
    for type, key, fields in entries(rdr) do
      if type == nil then
        break
      elseif not type then
        io.stderr:write('Something disastrous happened with entry ', key, '\n')
      elseif key == pattern or p(type, fields) then
        ⟨emit held warnings, if any⟩
        table.insert(es, { type = type, key = key, fields = fields,
                           bib = table.getn(bibs) > 1 and bib })
      else
        drop_warnings()
      end
    end
    rdr:close()
  end
  return es
end

function bibtex.run_find(argv)
  local emit = emit_tkf.terse
  while argv[1] and find(argv[1], '^-') do
    if emit_tkf[string.sub(argv[1], 2)] then
      emit = emit_tkf[string.sub(argv[1], 2)]
    else
      biberrorf('Unrecognized option %s', argv[1])
    end
    table.remove(argv, 1)
  end
  if table.getn(argv) == 0 then
    io.stderr:write(string.format('Usage: %s [-bib|-terse|-full] pattern [bibs]\n',
                                  string.gsub(argv[0], '.*/', '')))
    os.exit(1)
  end
  local pattern = table.remove(argv, 1)
  local bibs = { }
  ⟨make bibs the list of pathnames implied by argv⟩

  local entries = bibtex.find(pattern, bibs)
  for _, e in ipairs(entries) do
    emit(io.stdout, e.type, e.key, e.fields, e.bib)
  end
end
```

If we have no arguments, search all available bibfiles. Otherwise, an argument with a `/` is a pathname, and an argument without `/` is a name as it would appear in `\bibliography`.

⟨*make* `bibs` *the list of pathnames implied by* `argv`⟩≡

```
  if table.getn(argv) == 0 then
    bibs = all_bibs()
  else
    for _, a in ipairs(argv) do
      if find(a, ’/’) then
        table.insert(bibs, a)
      else
        table.insert(bibs, assert(bibtex.bibpath(a)))
      end
    end
  end
```

⟨*emit held warnings, if any*⟩≡

```
  local ws = held_warnings()
  if ws then
    for _, w in ipairs(ws) do
      emit_warning(unpack(w))
    end
  end
```

To search all bib files, we lean heavily on `kpsewhich`, which is distributed with the Web2C version of TeX, and which knows exactly which directories to search.

⟨*post-split Lua utility functions*⟩≡

```
  local function all_bibs()
    local pre_path = assert(capture(’kpsewhich -show-path bib’))
    local path = assert(capture(’kpsewhich -expand-path ’ .. pre_path))
    local bibs = { } -- list of results
    local inserted = { } -- set of inserted bibs, to avoid duplicates
    for _, dir in ipairs(split(path, ’:’)) do
      local files = assert(capture(’echo ’ .. dir .. ’/*.bib’))
      for _, file in ipairs(split(files, ’%s’)) do
        if readable(file) then
          if not (workaround.badbibs and (find(file, ’amsxport%-options’) or
                                          find(file, ’/plbib%.bib$’)))
          then
            if not inserted[file] then
              table.insert(bibs, file)
              inserted[file] = true
            end
          end
        end
      end
    end
    return bibs
  end
  bibtex.all_bibs = all_bibs
```

Notice the `workaround.badbibs`, which prevents us from searching some bogus bibfiles that come with Thomas Esser's teTeX.

It's a pity there's no more efficient way to see if a file is readable than to try to read it, but that's portability for you.

⟨*Lua utility functions*⟩+≡

```
  local function readable(file)
    local f, msg = io.open(file, 'r')
    if f then
      f:close()
      return true
    else
      return false, msg
    end
  end
```

5 Support for style files

A BibTeX style file is used to turn a BibTeX entry into TeX or LaTeX code suitable for inclusion in a bibliography. It can also be used for many other wondrous purposes, such as generating HTML for web pages. In classes BibTeX, each style file is written in a rudimentary, unnamed, stack-based language, which is described in a document called "Designing BibTeX Styles," which is often called `btxhak.dvi`. One of the benefits of NbibTeX is that styles can instead be written in Lua, which is a much more powerful language—and perhaps even easier to read.

But while Lua has amply powerful string-processing primitives, it lacks some of the primitives that are specific to BibTeX. Most notable among these primitives is the machinery for parsing and formatting names (of authors, editors and so on). That machinery is re-implemented here. If documentation seems scanty, consult the original `btxhak`.

In classic BibTeX, each style is its own separate file. Here, we share code by allowing a single file to register multiple styles.

⟨*exported Lua functions*⟩+≡

```
bibtex.doc.register_style =
  [[string * style -> unit # remember style with given name
type style = { emit   : outfile * string list * citation list -> unit
             , style  : table of formatting functions # defined document types
             , macros : unit -> macro table
             }]]
bibtex.doc.style = 'name -> style # return style with given name, loading on demand'

do
  local styles = { }

  function bibtex.register_style(name, s)
    assert(not styles[name], "Duplicate registration of style " .. name)
    styles[name] = s
    s.name = s.name or name
  end

  function bibtex.style(name)
    if not styles[name] then
      local loaded
      if config.nbs then
        local loaded = loadfile(config.nbs .. '/' .. name .. '.nbs')
        if loaded then loaded() end
      end
      if not loaded then
        require ('nbib-' .. name)
      end
      if not styles[name] then
        bibfatalf('Tried to load a file, but it did not register style %s\n', name)
      end
    end
    return styles[name]
  end
end
```

5.1 Special string-processing support

A great deal of BibTeX's processing depends on giving a special status to substrings inside braces; indeed, when such a substring begins with a backslash, it is called a "special character." Accordingly, we provide a function to search for a pattern *outside* balanced braces.

⟨*Lua utility functions*⟩+≡

```
local function find_outside_braces(s, pat, i)
  local len = string.len(s)
  local j, k = string.find(s, pat, i)
  if not j then return j, k end
  local jb, kb = string.find(s, '%b{}', i)
  while jb and jb < j do --- scan past braces
    --- braces come first, so we search again after close brace
    local i2 = kb + 1
    j, k = string.find(s, pat, i2)
    if not j then return j, k end
    jb, kb = string.find(s, '%b{}', i2)
  end
  -- either pat precedes braces or there are no braces
  return string.find(s, pat, j) --- 2nd call needed to get captures
end
```

5.1.1 String splitting

Another common theme in BibTeX is the list represented as string. A list of names is represented as a string with individual names separated by "and." A name itself is a list of parts separated by whitespace. So here are some functions to do general splitting. When we don't care about the separators, we use `split`; when we care only about the separators, we use `splitters`; and when we care about both, we use `odd_even_split`.

⟨*Lua utility functions*⟩+≡

```
local function split(s, pat, find) --- return list of substrings separated by pat
  find = find or string.find -- could be find_outside_braces
  local len = string.len(s)
  local t = { }
  local insert = table.insert
  local i, j, k = 1, true
  while j and i <= len + 1 do
    j, k = find(s, pat, i)
    if j then
      insert(t, string.sub(s, i, j-1))
      i = k + 1
    else
      insert(t, string.sub(s, i))
    end
  end
  return t
end
```

Function `splitters` returns a table that, when interleaved with the result of `split`, reconstructs the original string.

⟨*Lua utility functions*⟩+≡

```
local function splitters(s, pat, find) --- return list of separators
  find = find or string.find -- could be find_outside_braces
  local t = { }
  local insert = table.insert
  local j, k = find(s, pat, 1)
  while j do
    insert(t, string.sub(s, j, k))
    j, k = find(s, pat, k+1)
  end
  return t
end
```

Function `odd_even_split` makes odd entries strings between the sought-for pattern and even entries the strings that match the pattern.

⟨*Lua utility functions*⟩+≡

```
local function odd_even_split(s, pat)
  local len = string.len(s)
  local t = { }
  local insert = table.insert
  local i, j, k = 1, true
  while j and i <= len + 1 do
    j, k = find(s, pat, i)
    if j then
      insert(t, string.sub(s, i, j-1))
      insert(t, string.sub(s, j, k))
      i = k + 1
    else
      insert(t, string.sub(s, i))
    end
  end
  return t
end
```

As a special case, we may want to pull out brace-delimited substrings:

⟨*Lua utility functions*⟩+≡

```
local function brace_split(s) return odd_even_split(s, '%b{}') end
```

Some things need splits.

⟨*Lua utility functions*⟩+≡

  ⟨*post-split Lua utility functions*⟩

5.1.2 String lengths and widths

Function `text_char_count` counts characters, but a special counts as one character. It is based on BibTeX's `text.length$` function.

⟨*Lua utility functions*⟩+≡

```
local function text_char_count(s)
  local n = 0
  local i, last = 1, string.len(s)
  while i <= last do
    local special, splast, sp = find(s, '(%b{})', i)
    if not special then
      return n + (last - i + 1)
    elseif find(sp, '^{\\') then
      n = n + (special - i + 1) -- by statute, it's a single character
      i = splast + 1
    else
      n = n + (splast - i + 1) - 2  -- don't count braces
      i = splast + 1
    end
  end
  return n
end
bst.text_length = text_char_count
bst.doc.text_length = "string -> int # length (with 'special' char == 1)"
```

Sometimes we want to know not how many characters are in a string, but how much space we expect it to take when typeset. (Or rather, we want to compare such widths to find the widest.) This is original BibTeX's `width$` function.

The code should use the `char_width` array, for which `space` is the only whitespace character given a nonzero printing width. The widths here are taken from Stanford's June '87 *cmr*10 font and represent hundredths of a point (rounded), but since they're used only for relative comparisons, the units have no meaning.

⟨*exported Lua functions*⟩+≡

```
do
  local char_width = { }
  local special_widths = { ss = 500, ae = 722, oe = 778, AE = 903, oe = 1014 }
  for i = 0, 255 do char_width[i] = 0 end
  local char_width_from_32 = {
     278, 278, 500, 833, 500, 833, 778, 278, 389, 389, 500, 778, 278, 333,
     278, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 278, 278,
     278, 778, 472, 472, 778, 750, 708, 722, 764, 681, 653, 785, 750, 361,
     514, 778, 625, 917, 750, 778, 681, 778, 736, 556, 722, 750, 750,
     1028, 750, 750, 611, 278, 500, 278, 500, 278, 278, 500, 556, 444,
     556, 444, 306, 500, 556, 278, 306, 528, 278, 833, 556, 500, 556, 528,
     392, 394, 389, 556, 528, 722, 528, 528, 444, 500, 1000, 500, 500,
  }
  for i = 1, table.getn(char_width_from_32) do
    char_width[32+i-1] = char_width_from_32[i]
  end

  bst.doc.width = "string -> faux_points # width of string in 1987 cmr10"
  function bst.width(s)
    assert(false, 'have not implemented width yet')
  end
end
```

5.2 Parsing names and lists of names

Names in a string are separated by `and` surrounded by nonnull whitespace. Case is not significant.

⟨*exported Lua functions*⟩+≡

```
local function namesplit(s)
  local t = split(s, '%s+[aA][nN][dD]%s+', find_outside_braces)
  local i = 2
  while i <= table.getn(t) do
    while find(t[i], '^[aA][nN][dD]%s+') do
      t[i] = string.gsub(t[i], '^[aA][nN][dD]%s+', '')
      table.insert(t, i, '')
      i = i + 1
    end
    i = i + 1
  end
  return t
end
bst.namesplit = namesplit
bst.doc.namesplit = 'string -> list of names # split names on "and"'
```

⟨*exported Lua functions*⟩+≡

```
local sep_and_not_tie = '%-'
local sep_chars = sep_and_not_tie .. '%~'
```

To parse an individual name, we want to count commas. We first remove leading white space (and `sep_chars`), and trailing white space (and `sep_chars`) and commas, complaining for each trailing comma.

We then represent the name as two sequences: `tokens` and `trailers`. The `tokens` are the names themselves, and the `trailers` are the separator characters between tokens. (A separator is white space, a dash, or a tie, and multiple separators in sequence are frowned upon.) The `commas` table becomes an array mapping the comma number to the index of the token that follows it.

⟨*exported Lua functions*⟩+≡

```
local parse_name
do
  local white_sep         = ’[’ .. sep_chars .. ’%s]+’
  local white_comma_sep   = ’[’ .. sep_chars .. ’%s%,]+’
  local trailing_commas   = ’(,[’ .. sep_chars .. ’%s%,]*)$’
  local sep_char          = ’[’ .. sep_chars .. ’]’
  local leading_white_sep = ’^’ .. white_sep

  ⟨name-parsing utilities⟩

  function parse_name(s, inter_token)
    if string.find(s, trailing_commas) then
      biberrorf("Name ’%s’ has one or more commas at the end", s)
    end
    s = string.gsub(s, trailing_commas, ’’)
    s = string.gsub(s, leading_white_sep, ’’)
    local tokens = split(s, white_comma_sep, find_outside_braces)
    local trailers = splitters(s, white_comma_sep, find_outside_braces)
    ⟨rewrite trailers to hold a single separator character each⟩
    local commas = { } --- maps each comma to index of token the follows it
    for i, t in ipairs(trailers) do
      string.gsub(t, ’,’, function() table.insert(commas, i+1) end)
    end
    local name = { }
    ⟨parse the name tokens and set fields of name⟩
    return name
  end
end
bst.parse_name = parse_name
bst.doc.parse_name = ’string * string option -> name table’
```

A name has up to four parts: the most general form is either "First von Last, Junior" or "von Last, First, Junior", but various vons and Juniors can be omitted. The name-parsing algorithm is baroque and is transliterated from the original BibTeX source, but the principle is clear: assign the full version of each part to the four fields `ff`, `vv`, `ll`, and `jj`; and assign an abbreviated version of each part to the fields `f`, `v`, `l`, and `j`.

⟨*parse the name tokens and set fields of* `name`⟩≡

```
local first_start, first_lim, last_lim, von_start, von_lim, jr_lim
  -- variables mark subsequences; if start == lim, sequence is empty
local n = table.getn(tokens)
⟨local parsing functions⟩

local commacount = table.getn(commas)
if commacount == 0 then -- first von last jr
  von_start, first_start, last_lim, jr_lim = 1, 1, n+1, n+1
  ⟨parse first von last jr⟩
elseif commacount == 1 then -- von last jr, first
  von_start, last_lim, jr_lim, first_start, first_lim =
    1, commas[1], commas[1], commas[1], n+1
  divide_von_from_last()
elseif commacount == 2 then -- von last, jr, first
  von_start, last_lim, jr_lim, first_start, first_lim =
    1, commas[1], commas[2], commas[2], n+1
  divide_von_from_last()
else
  biberrorf("Too many commas in name ’%s’")
end
⟨set fields of name based on first_start and friends⟩
```

The von name, if any, goes from the first von token to the last von token, except the last name is entitled to at least one token. So to find the limit of the von name, we start just before the last token and wind down until we find a von token or we hit the von start (in which latter case there is no von name).

⟨*local parsing functions*⟩≡

```
function divide_von_from_last()
  von_lim = last_lim - 1;
  while von_lim > von_start and not isVon(tokens[von_lim-1]) do
    von_lim = von_lim - 1
  end
end
```

OK, here's one form.

⟨*parse first von last jr*⟩≡
```
local got_von = false
while von_start < last_lim-1 do
  if isVon(tokens[von_start]) then
    divide_von_from_last()
    got_von = true
    break
  else
    von_start = von_start + 1
  end
end
if not got_von then -- there is no von name
  while von_start > 1 and find(trailers[von_start - 1], sep_and_not_tie) do
    von_start = von_start - 1
  end
  von_lim = von_start
end
first_lim = von_start
```

The last name starts just past the last token, before the first comma (if there is no comma, there is deemed to be one at the end of the string), for which there exists a first brace-level-0 letter (or brace-level-1 special character), and it's in lower case, unless this last token is also the last token before the comma, in which case the last name starts with this token (unless this last token is connected by a `sep_char` other than a `tie` to the previous token, in which case the last name starts with as many tokens earlier as are connected by non`tie`s to this last one (except on Tuesdays . . .), although this module never sees such a case). Note that if there are any tokens in either the von or last names, then the last name has at least one, even if it starts with a lower-case letter.

The string separating tokens is reduced to a single "separator character." A comma always trumps other separator characters. Otherwise, if there's no comma, we take the first character, be it a separator or a space. (Patashnik considers that multiple such characters constitute "silliness" on the user's part.)

⟨*rewrite* `trailers` *to hold a single separator character each*⟩≡
```
for i = 1, table.getn(trailers) do
  local s = trailers[i]
  assert(string.len(s) > 0)
  if find(s, ',') then
    trailers[i] = ','
  else
    trailers[i] = string.sub(s, 1, 1)
  end
end
```

⟨*set fields of name based on* `first_start` *and friends*⟩≡
```
⟨definition of function set_name⟩
set_name(first_start, first_lim, 'ff', 'f')
set_name(von_start,   von_lim,   'vv', 'v')
set_name(von_lim,     last_lim,  'll', 'l')
set_name(last_lim,    jr_lim,    'jj', 'j')
```

We set long and short forms together; `ss` is the long form and `s` is the short form.

⟨*definition of function* `set_name`⟩≡

```
local function set_name(start, lim, long, short)
  if start < lim then
    -- string concatenation is quadratic, but names are short
    ⟨definition of abbrev, for shortening a token⟩
    local ss = tokens[start]
    local s  = abbrev(tokens[start])
    for i = start + 1, lim - 1 do
      if inter_token then
        ss = ss .. inter_token .. tokens[i]
        s  = s  .. inter_token .. abbrev(tokens[i])
      else
        local ssep, nnext = trailers[i-1], tokens[i]
        local sep,  next  = ssep,          abbrev(nnext)
        ⟨possibly adjust sep and ssep according to token position and size⟩
        ss = ss ..        ssep .. nnext
        s  = s  .. '.' .. sep  .. next
      end
    end
    name[long] = ss
    name[short] = s
  end
end
```

Here is the default for a character between tokens: a tie is the default space character between the last two tokens of the name part, and between the first two tokens if the first token is short enough; otherwise, a space is the default.

⟨*possibly adjust* `sep` *and* `ssep` *according to token position and size*⟩≡

```
if find(sep, sep_char) then
  -- do nothing; sep is OK
elseif i == lim-1 then
  sep, ssep = '~', '~'
elseif i == start + 1 then
  sep  = text_char_count(s)  < 3 and '~' or ' '
  ssep = text_char_count(ss) < 3 and '~' or ' '
else
  sep, ssep = ' ', ' '
end
```

The von name starts with the first token satisfying `isVon`, unless that is the last token. A “von token” is simply one that begins with a lower-case letter—but those damn specials complicate everything.

⟨*Lua utility functions*⟩+≡

```
local upper_specials = { OE = true, AE = true, AA = true, O = true, L = true }
local lower_specials = { i = true, j = true, oe = true, ae = true, aa = true,
                         o = true, l = true, ss = true }
```

⟨*name-parsing utilities*⟩≡

```
function isVon(s)
  local lower  = find_outside_braces(s, ’%l’) -- first nonbrace lowercase
  local letter = find_outside_braces(s, ’%a’) -- first nonbrace letter
  local bs, ebs, command = find_outside_braces(s, ’%{%\\(%a+)’) -- \xxx
  if lower and lower <= letter and lower <= (bs or lower) then
    return true
  elseif letter and letter <= (bs or letter) then
    return false
  elseif bs then
    if upper_specials[command] then
      return false
    elseif lower_specials[command] then
      return true
    else
      local close_brace = find_outside_braces(s, ’%}’, ebs+1)
      lower  = find(s, ’%l’) -- first nonbrace lowercase
      letter = find(s, ’%a’) -- first nonbrace letter
      return lower and lower <= letter
    end
  else
    return false
  end
end
```

An abbreviated token is the first letter of a token, except again we have to deal with the damned specials.

⟨*definition of* `abbrev`*, for shortening a token*⟩≡

```
local function abbrev(token)
  local first_alpha, _, alpha = find(token, ’(%a)’)
  local first_brace           = find(token, ’%{%\\’)
  if first_alpha and first_alpha <= (first_brace or first_alpha) then
    return alpha
  elseif first_brace then
    local i, j, special = find(token, ’(%b{})’, first_brace)
    if i then
      return special
    else -- unbalanced braces
      return string.sub(token, first_brace)
    end
  else
    return ’’
  end
end
```

5.3 Formatting names

Lacking Lua's string-processing utilities, classic BIBTEX defines a way of converting a "format string" and a name into a formatted name. I find this formatting technique painful, but I also wanted to preserve compatibility with existing bibliography styles, so I've implemented it as accurately as I can.

The interface is not quite identical to classic BIBTEX; a style can use `namesplit` to split names and then `format_name` to format a single one, or it can throw caution to the winds and call `format_names` to format a whole list of names.

⟨*exported Lua functions*⟩+≡

```
bst.doc.format_names = "format * name list -> string list # format each name in list"
function bst.format_names(fmt, t)
  local u = { }
  for i = 1, table.getn(t) do
    u[i] = bst.format_name(fmt, t[i])
  end
  return u
end
```

A BIBTEX format string contains its variable elements inside braces. Thus, we format a name by replacing each braced substring of the format string.

⟨*exported Lua functions*⟩+≡

```
do
  local good_keys = { ff = true, vv = true, ll = true, jj = true,
                      f  = true, v  = true, l  = true, j  = true, }

  bst.doc.format_name = "format * name -> string # format 1 name as in bibtex"
  function bst.format_name(fmt, name)
    local t = type(name) == 'table' and name or parse_name(name)
    -- at most one of the important letters, perhaps doubled, may appear
    local function replace_braced(s)
      local i, j, alpha = find_outside_braces(s, '(%a+)', 2)
      if not i then
        return '' --- can never be printed, but who are we to complain?
      elseif not good_keys[alpha] then
        biberrorf ('The format string %q has an illegal brace-level-1 letter', s)
      elseif find_outside_braces(s, '%a+', j+1) then
        biberrorf ('The format string %q has two sets of brace-level-1 letters', s)
      elseif t[alpha] then
        local k = j + 1
        local t = t
        ⟨make k follow inter-token string, if any, rebuilding t as needed⟩
        local head, tail = string.sub(s, 2, i-1) .. t[alpha], string.sub(s, k, -2)
        ⟨adjust tail to account for discretionality of ties, if any⟩
        return head .. tail
      else
        return ''
      end
    end
    return (string.gsub(fmt, '%b{}', replace_braced))
  end
end
```

⟨*make* `k` *follow inter-token string, if any, rebuilding* `t` *as needed*⟩≡

```
local kk, jj = find(s, '%b{}', k)
if kk and kk == k then
  k = jj + 1
  if type(name) == 'string' then
    t = parse_name(name, string.sub(s, kk+1, jj-1))
  else
    error('Style error -- used a pre-parsed name with non-standard inter-token format string')
  end
end
```

⟨*adjust* `tail` *to account for discretionality of ties, if any*⟩≡

```
if find(tail, '%~%~$') then
  tail = string.sub(tail, 1, -2) -- denotes hard tie
elseif find(tail, '%~$') then
  if text_char_count(head) + text_char_count(tail) - 1 >= 3 then
    tail = string.gsub(tail, '%~$', ' ')
  end
end
```

5.4 Line-wrapping output

EXPLAIN THIS INTERFACE!!!

My `max_print_line` appears to be off by one from Oren Patashnik's.

⟨*exported Lua functions*⟩+≡

```
local min_print_line, max_print_line = 3, 79
bibtex.hard_max = max_print_line
bibtex.doc.hard_max = 'int # largest line that avoids a forced line break (for wizards)'
bst.doc.writer = "io-handle * int option -> object # result:write(s) buffers and breaks lines"
function bst.writer(out, indent)
  indent = indent or 2
  assert(indent + 10 < max_print_line)
  indent = string.rep(' ', indent)
  local gsub = string.gsub
  local buf = ''
  local function write(self, ...)
    local s = table.concat { ... }
    local lines = split(s, '\n')
    lines[1] = buf .. lines[1]
    buf = table.remove(lines)
    for i = 1, table.getn(lines) do
      local line = lines[i]
      if not find(line, '^%s+$') then -- no line of just whitespace
        line = gsub(line, '%s+$', '')
        while string.len(line) > max_print_line do
          ⟨emit initial part of line and reassign⟩
        end
        out:write(line, '\n')
      end
    end
  end
  assert(out.write, "object passed to bst.writer does not have a write method")
  return { write = write }
end
```

⟨*emit initial part of line and reassign*⟩≡

```
local last_pre_white, post_white
local i, j, n = 1, 1, string.len(line)
while i and i <= n and i <= max_print_line do
  i, j = find(line, '%s+', i)
  if i and i <= max_print_line + 1 then
    if i > min_print_line then last_pre_white, post_white = i - 1, j + 1 end
    i = j + 1
  end
end
if last_pre_white then
  out:write(string.sub(line, 1, last_pre_white), '\n')
  if post_white > max_print_line + 2 then
    post_white = max_print_line + 2 -- bug-for-bug compatibility with bibtex
  end
  line = indent .. string.sub(line, post_white)
elseif n < bibtex.hard_max then
  out:write(line, '\n')
  line = ''
else -- ‘‘unbreakable’’
  out:write(string.sub(line, 1, bibtex.hard_max-1), '%\n')
  line = string.sub(line, bibtex.hard_max)
end
```

⟨*check constant values for consistency*⟩≡

```
assert(min_print_line >= 3)
assert(max_print_line > min_print_line)
```

5.5 Functions copied from classic BibTeX

Adding a period Find the last non-} character, and if it is not a sentence terminator, add a period.

⟨*exported Lua functions*⟩+≡

```
do
  local terminates_sentence = { ["."] = true, ["?"] = true, ["!"] = true }

  bst.doc.add_period = "string -> string # add period unless already .?!"
  function bst.add_period(s)
    local _, _, last = find(s, '([^%}])%}*$')
    if last and not terminates_sentence[last] then
      return s .. '.'
    else
      return s
    end
  end
end
```

Case-changing Classic BibTeX has a `change.case$` function, which takes an argument telling whether to change to lower case, upper case, or "title" case (which has initial letters capitalized). Because Lua supports first-class functions, it makes more sense just to export three functions: `lower`, `title`, and `upper`.

⟨*exported Lua functions*⟩+≡

```
do
  bst.doc.lower = "string -> string  # lower case according to bibtex rules"
  bst.doc.upper = "string -> string  # upper case according to bibtex rules"
  bst.doc.title = "string -> string  # title case according to bibtex rules"

  ⟨utilities for case conversion⟩

  ⟨definitions of case-conversion functions⟩
end
```

Case conversion is complicated by the presence of brace-delimited sequences, especially since there is one set of conventions for a "special character" (brace-delimited sequence beginning with TeX control sequence) and another set of conventions for other brace-delimited sequences. To deal with them, we typically do an "odd-even split" on balanced braces, then apply a "normal" conversion function to the odd elements and a "special" conversion function to the even elements. The application is done by `oeapp`.

⟨*utilities for case conversion*⟩≡

```
local function oeapp(f, g, t)
  for i = 1, table.getn(t), 2 do
    t[i] = f(t[i])
  end
  for i = 2, table.getn(t), 2 do
    t[i] = g(t[i])
  end
  return t
end
```

Upper- and lower-case conversion are easiest. Non-specials are hit directly with `string.lower` or `string.upper`; for special characters, we use utility called `convert_special`.

⟨*definitions of case-conversion functions*⟩≡

```
local lower_special = convert_special(string.lower)
local upper_special = convert_special(string.upper)

function bst.lower(s)
  return table.concat(oeapp(string.lower, lower_special, brace_split(s)))
end

function bst.upper(s)
  return table.concat(oeapp(string.upper, upper_special, brace_split(s)))
end
```

Here is `convert_special`. If a special begins with an alphabetic control sequence, we convert only elements between control sequences. If a special begins with a nonalphabetic control sequence, we convert the whole special as usual. Finally, if a special does not begin with a control sequence, we leave it the hell alone. (This is the convention that allows us to put `{FORTRAN}` in a BibTeX entry and be assured that capitalization is not lost.)

⟨*utilities for case conversion*⟩+≡

```
function convert_special(cvt)
  return function(s)
           if find(s, '^{\\(%a+)') then
             local t = odd_even_split(s, '\\%a+')
             for i = 1, table.getn(t), 2 do
               t[i] = cvt(t[i])
             end
             return table.concat(t)
           elseif find(s, '^{\\') then
             return cvt(s)
           else
             return s
           end
         end
end
```

Title conversion doesn't fit so nicely into the framework.

Function `lower_later` lowers all but the first letter of a string.

⟨*utilities for case conversion*⟩+≡

```
local function lower_later(s)
  return string.sub(s, 1, 1) .. string.lower(string.sub(s, 2))
end
```

For title conversion, we don't mess with a token that follows a colon. Hence, we must maintain `prev` and can't use `convert_special`.

⟨*definitions of case-conversion functions*⟩+≡

```
local function title_special(s, prev)
  if find(prev, ':%s+$') then
    return s
  else
    if find(s, '^{\\(%a+)') then
      local t = odd_even_split(s, '\\%a+')
      for i = 1, table.getn(t), 2 do
        local prev = t[i-1] or prev
        if find(prev, ':%s+$') then
          assert(false, 'bugrit')
        else
          t[i] = string.lower(t[i])
        end
      end
      return table.concat(t)
    elseif find(s, '^{\\') then
      return string.lower(s)
    else
      return s
    end
  end
end
```

Internal function `recap` deals with the damn colons.

⟨*definitions of case-conversion functions*⟩+≡

```
function bst.title(s)
  local function recap(s, first)
    local parts = odd_even_split(s, '%:%s+')
    parts[1] = first and lower_later(parts[1]) or string.lower(parts[1])
    for i = (first and 3 or 1), table.getn(parts), 2 do
      parts[i] = lower_later(parts[i])
    end
    return table.concat(parts)
  end
  local t = brace_split(s)
  for i = 1, table.getn(t), 2 do -- elements outside specials get recapped
    t[i] = recap(t[i], i == 1)
  end
  for i = 2, table.getn(t), 2 do -- specials are, well, special
    local prev = t[i-1]
    if i == 2 and not find(prev, '%S') then prev = ': ' end
    t[i] = title_special(t[i], prev)
  end
  return table.concat(t)
end
```

Purification Purification (classic `purify$`) involves removing non-alphanumeric characters. Each sequence of “separator” characters becomes a single space.

⟨*exported Lua functions*⟩+≡

```
do
  bst.doc.purify = "string -> string # remove nonalphanumeric, non-sep chars"
  local high_alpha = string.char(128) .. '-' .. string.char(255)
  local sep_white_char = '[' .. sep_chars .. '%s]'
  local disappears     = '[^' .. sep_chars .. high_alpha .. '%s%w]'
  local gsub = string.gsub
  local function purify(s)
    return gsub(gsub(s, sep_white_char, ' '), disappears, '')
  end
  -- special characters are purified by removing all non-alphanumerics,
  -- including white space and sep-chars
  local function spurify(s)
    return gsub(s, '[^%w' .. high_alpha .. ']+', '')
  end
  local purify_all_chars = { oe = true, OE = true, ae = true, AE = true, ss = true }

  function bst.purify(s)
    local t = brace_split(s)
    for i = 1, table.getn(t) do
      local _, k, cmd = find(t[i], '^{\\(%a+)%s*')
      if k then
        if lower_specials[cmd] or upper_specials[cmd] then
          if not purify_all_chars[cmd] then
            cmd = string.sub(cmd, 1, 1)
          end
          t[i] = cmd .. spurify(string.sub(t[i], k+1))
        else
          t[i] = spurify(string.sub(t[i], k+1))
        end
      elseif find(t[i], '^{\\') then
        t[i] = spurify(t[i])
      else
        t[i] = purify(t[i])
      end
    end
    return table.concat(t)
  end
end
```

Text prefix Function `text_prefix` (classic `text.prefix$`) takes an initial substring of a string, with the proviso that a BibTeX “special character” sequence counts as a single character.

⟨*exported Lua functions*⟩+≡

```
bst.doc.text_prefix = "string * int -> string  # take first n chars with special == 1"
function bst.text_prefix(s, n)
  local t = brace_split(s)
  local answer, rem = ’’, n
  for i = 1, table.getn(t), 2 do
    answer = answer .. string.sub(t[i], 1, rem)
    rem = rem - string.len(t[i])
    if rem <= 0 then return answer end
    if find(t[i+1], ’^{\\’) then
      answer = answer .. t[i+1]
      rem = rem - 1
    else
      ⟨take up to rem characters from t[i+1], not counting braces⟩
    end
  end
  return answer
end
```

⟨*take up to* `rem` *characters from* `t[i+1]`, *not counting braces*⟩≡

```
local s = t[i+1]
local braces = 0
local sub = string.sub
for i = 1, string.len(s) do
  local c = sub(s, i, i)
  if c == ’{’ then
    braces = braces + 1
  elseif c == ’}’ then
    braces = braces + 1
  else
    rem = rem - 1
    if rem == 0 then
      return answer .. string.sub(s, 1, i) .. string.rep(’}’, braces)
    end
  end
end
answer = answer .. s
```

Emptiness test Function `empty` (classic `empty$`) tells if a value is empty; i.e., it is missing (nil) or it is only white space.

⟨*exported Lua functions*⟩+≡

```
bst.doc.empty = "string option -> bool # is string there and holding nonspace?"
function bst.empty(s)
  return s == nil or not find(s, ’%S’)
end
```

5.6 Other utilities

A stable sort Function `bst.sort` is like `table.sort` only stable. It is needed because classic BibTeX uses a stable sort. Its interface is the same as `table.sort`.

⟨*exported Lua functions*⟩+≡

```
bst.doc.sort = ’value list * compare option # like table.sort, but stable’
function bst.sort(t, lt)
  lt = lt or function(x, y) return x < y end
  local pos = { } --- position of each element in original table
  for i = 1, table.getn(t) do pos[t[i]] = i end
  local function nlt(x, y)
    if lt(x, y) then
      return true
    elseif lt(y, x) then
      return false
    else -- elements look equal
      return pos[x] < pos[y]
    end
  end
  return table.sort(t, nlt)
end
bst.doc.sort = ’value list * compare option -> unit  # stable sort’
```

The standard months Every style is required to recognize the months, so we make it easy to create a fresh table with either full or abbreviated months.

⟨*exported Lua functions*⟩+≡

```
bst.doc.months = "string option -> table # macros table containing months"
function bst.months(what)
  local m = {
    jan = "January", feb = "February", mar = "March", apr = "April",
    may = "May", jun = "June", jul = "July", aug = "August",
    sep = "September", oct = "October", nov = "November", dec = "December" }
  if what == ’short’ or what == 3 then
    for k, v in pairs(m) do
      m[k] = string.sub(v, 1, 3)
    end
  end
  return m
end
```

Comma-separated lists The function `commafy` takes a list and inserts commas and `and` (or `or`) using American conventions. For example,

```
commafy { 'Graham', 'Knuth', 'Patashnik' }
```

returns `'Graham, Knuth, and Patashnik'`, but

```
commafy { 'Knuth', 'Plass' }
```

returns `'Knuth and Plass'`.

⟨*exported Lua functions*⟩+≡

```
bst.doc.commafy = "string list -> string # concat separated by commas, and"
function bst.commafy(t, andword)
  andword = andword or 'and'
  local n = table.getn(t)
  if n == 1 then
    return t[1]
  elseif n == 2 then
    return t[1] .. ' ' .. andword .. ' ' .. t[2]
  else
    local last = t[n]
    t[n] = andword .. ' ' .. t[n]
    local answer = table.concat(t, ', ')
    t[n] = last
    return answer
  end
end
```

6 Testing and so on

Here are a couple of test functions I used during development that I thought might be worth keeping around.

⟨*exported Lua functions*⟩+≡

```
bibtex.doc.cat = 'string -> unit # emit the named bib file in bib format'
function bibtex.cat(bib)
  local rdr = bibtex.open(bib, bst.months())
  if not rdr then
    rdr = assert(bibtex.open(assert(bibtex.bibpath(bib)), bst.months()))
  end
  for type, key, fields in entries(rdr) do
    if type == nil then
      break
    elseif not type then
      io.stderr:write('Error on key ', key, '\n')
    else
      emit_tkf.bib(io.stdout, type, key, fields)
    end
  end
  bibtex.close(rdr)
end
```

⟨*exported Lua functions*⟩+≡

```
bibtex.doc.count = 'string list -> unit # take list of bibs and print number of entries'
function bibtex.count(argv)
  local bibs = { }
  local macros = { }
  local n = 0
  ⟨make bibs the list of pathnames implied by argv⟩
  local function warn() end
  for _, bib in ipairs(bibs) do
    local rdr = bibtex.open(bib, macros)
    for type, key, fields in entries(rdr) do
      if type == nil then
        break
      elseif type then
        n = n + 1
      end
    end
    rdr:close()
  end
  printf("%d\n", n)
end
```

⟨*exported Lua functions*⟩+≡

```
bibtex.doc.all_entries = "bibname * macro-table -> preamble * citation list"
function bibtex.all_entries(bib, macros)
  macros = macros or bst.months()
  warn = warn or emit_warning
  local rdr = bibtex.open(bib, macros, warn)
  if not rdr then
    rdr = assert(bibtex.open(assert(bibtex.bibpath(bib)), macros, warn),
                 "could not open bib file " .. bib)
  end
  local cs = { }
  local seen = { }
  for type, key, fields in entries(rdr) do
    if type == nil then
      break
    elseif not type then
      io.stderr:write(key, '\n')
    elseif not seen[key] then
      seen[key] = true
      table.insert(cs, { type = type, key = key, fields = fields, file = bib,
                         line = rdr.entry_line })
    end
  end
  local p = assert(rdr.preamble)
  rdr:close()
  return p, cs
end
```

7 Laundry list

THINGS TO DO:

- TRANSITION THE C CODE TO LUA NATIVE ERROR HANDLING (`luaL_error` and `pcall`)
- NO WARNING FOR DUPLICATE FIELDS NOT DEFINED IN .BST?
- STANDARD WARNING FOR REPEATED ENTRY?
- NOT ENFORCED: An entry type must be defined in the `.bst` file if this entry is to be included in the reference list.
- THE WHOLE BST-SEARCH THING NEEDS MORE CARE.

  BibTeX searches the directories in the path defined by the BSTINPUTS environment variable for .bst files. If BSTINPUTS is not set, it uses the system default. For .bib files, it uses the BIBINPUTS environment variable if that is set, otherwise the default. See tex(1) for the details of the searching.

  If the environment variable TEXMFOUTPUT is set, BibTeX attempts to put its output files in it, if they cannot be put in the current directory. Again, see tex(1). No special searching is done for the .aux file.
- RATIONALIZE ERROR MACHINERY WITH WARNING, ERROR, AND FATAL CASES – AND COUNTS.
- Here are some things that BibTeX does that NbibTeX should do:

  1. Writes a log file
  2. Counts warnings, or if there is an error, counts errors instead