獣を馴らす準備

第 2 部：btxbst.doc を読むために

私立文系🔰初級ユーザー

2024 年 12 月 4 日

🎄 TEX & LATEX Advent Calendar 2024 🎄

構成についてとオリジナルについて

⚠ 2 部構成になっています ——

本ペーパーは、BIBTEX のドキュメントを読み解く助けとなることを目指すものですが、

🦁 第 1 部：btxhak.pdf を読むために

🐯 第 2 部：btxbst.doc を読むために

という 2 部構成になっていて、物理的にも、2 つの文書に分けてあります。こちらはその第 2 部のほうになります。

⚠ オリジナルのドキュメントの書誌は、以下の通りです(*) ——

[1a] Oren Patashnik, *BIBTEXing*, February 8, 1988. (btxdoc.tex|pdf)

[2a] Oren Patashnik, *Designing BIBTEX Styles*, February 8, 1988. (btxhak.tex|pdf)

[3a] Howard Trickey and Oren Patashnik, *BIBTEX 'plain' family*, 1984, 1985, 1988, 2010. (btxbst.doc)

はじめに

このペーパーは、BIBTEX の standard styles (plain.bst、unsrt.bst、alpha.bst、abbrv.bst) の documented source である *BIBTEX 'plain' family* (btxbst.doc) [3a] を、切り貼りしたものです。

btxbst.doc は、拡張子が“doc”となっていますが、これはもちろん Word のファイルではなくて、中味はテキストファイルです。LATEX2.09 の時代には、コードにコメント〔ドキュメンテーション〕を付した“doc ファイル”と、doc からコメントを削除してコードのみを残した“cls ファイル”や“sty ファイル”という二本建てで、配布がなされていたらしいです（今では、ひとつの dtx ファイルから、ドキュメントもコードも得られるようになっていますね）。

なお、dtx ファイルの場合には、docstrip プログラムで処理する際にオプションを指定して、コードの特定の部分を include するか exclude するかを指定できますが、btxbst.doc の場合にも同様に、CPP (C preprocessor) というプログラムを使って、コードを conditional に生成できるようになっています(1)。

btxbst.doc は、頭から順を追って読んでいけば理解できるように書かれていますが、章や節に分けたり、見出しを付けたりはされていないので、全体の構成や、function の定義順の制約等が予め分かっていないと、ちょっぴりだけ読みづらいです。

(*) 日本語訳もあります：[1b] 松井正一訳「BIBTEXing：BIBTEX の使い方」（1991 年 1 月 1 日）(jbtxdoc.tex|pdf)；[2b] 松井正一訳「BIBTEX スタイルの作り方」（1991 年 1 月 1 日）(jbtxhak.tex|pdf)；[3b] 松井 正一「JBibTeX 'jplain' family」（1994-10-25）(jbtxbst.doc)。私の手元の W32TEX [2020/07/19] にはいずれも含まれているのですが、TEX Live には入っていないのかも知れません。その場合には、CTAN に行って、[1b] と [2b] については pbibtex-manual パッケージ、[3b] については pbibtex-base パッケージを探してみてください。

(1) btxbst.doc のオリジナルのコメント部分を削除してから dtx に書き直した上で、多数のオプションを付したものが、custom-bib パッケージの merlin.mbs ですね。

どういうことかと言いますと、例えば、`calc.label` という function の内部では `author.key.label` が使われていて、その `author.key.label` の中では `format.lab.names` が使われているという場合に、定義やドキュメンテーションは、`format.lab.names` → `author.key.label` → `calc.label` という順番になされています。

そのため、`format.lab.names` や `author.key.label` の部分を読んでいるときには少々辛抱が必要で、`calc.label` に至るとようやく、「あぁ、そういうことね！」と分かる感じになります（私が物分かりが悪いせいもあるかも知れませんけれど）。

`btxbst.doc` の構成については、*TLC* でも簡潔に説明されていますが[2]、以下では、適宜見出しを補いつつドキュメンテーション部分をコピペして、`dtx` 風にしてみました。見出しと脚註は私が勝手に適当につけたものです。飽くまでドキュメント全体の流れを把握することが目的ですので、コード自体の多くは省略しています（なので、個々のコードについては、オリジナルのほうをご参照ください）。

BibTeX 'plain' family
(`btxbst.doc`)

Version 0.99b
(8-Dec-10 release)

[2] *The LaTeX Companion*, 1st (1993), "13.7.4 The Documentation Style `btxbst.doc`"; 2nd (2004), "13.6.2 The documentation style `btxbst.doc`".

1 C Preprocessor Constructs

This is file `btxbst.doc`; it helps document bibliography styles, and is also a template file that you can use to make several different style files, if you have access to a C preprocessor.

For example, the standard styles were made by doing something like

```
cpp -P -DPLAIN btxbst.doc plain.txt
cpp -P -DUNSRT btxbst.doc unsrt.txt
cpp -P -DALPHA btxbst.doc alpha.txt
cpp -P -DABBRV btxbst.doc abbrv.txt
```

and then renaming after removing unwanted comments and blank lines.

If you don't have access, you can edit this file by hand to imitate the preprocessor, with the following explanation of the C preprocessor constructs used here.

The output of the preprocessor is the same as the input, except that certain lines will be excluded (and some blank lines will be added). The sequence

```
#if VAR
    lines to be included when VAR is not zero
#else
    lines to be included when VAR is zero
#endif
```

(with the #-signs appearing in column 1) means that one set or the other of the lines are to be included depending on the value of `VAR`. The `#else` part is optional. Comments can be added after `#else` and `#endif`.

Variables can be set by

```
#define VAR value
```

and one can also use `#ifdef VAR` to see if `VAR` has any value, and `#ifndef` to see if it has none.

Another `#if` form used in this file is `#if !VAR`, which includes the lines after the `#if` only if `VAR` is zero.

Convention: Use all uppercase identifiers for these preprocessor variables so you can spot them easily

1.1 CPP Variables

The command line to the preprocessor should define one of `PLAIN`, `UNSRT`, `ALPHA` or `ABBRV` (though `PLAIN` will be used by default if none is given), and the following lines will set various boolean variables to control the various lines that are chosen from the rest of the file.

Each boolean variable should be set true (`1`) or false (`0`) in each style.

Here are the current variables, and their meanings:

- `LAB_ALPH`: an alphabetic label is used (if false then a numeric label is used)
- `SORTED`: the entries should be sorted by label (if nonnumeric) and other info, like authors (if false, then entries remain in order of occurrence)
- `NAME_FULL`: the authors, editors, etc., get the full names as given in the bibliography file (if false, the first names become initials)
- `ATIT_LOWER`: titles of non-"books" (e.g., articles) should be converted to lowercase, except the first letter or first letter after a colon (if false then they appear as in the database)
- `MONTH_FULL`: months are spelled out in full (if false, then they're abbreviated)
- `JOUR_FULL`: macro journal names are spelled out in full (if false then they are abbreviated, currently as they appear in ACM publications)

Selecting a Default Style

```
#ifndef UNSRT
#   ifndef ALPHA
#       ifndef ABBRV
#           define PLAIN 1
#       endif
#   endif
#endif
```

Style Set of Defined Values [1]

- plain style (sorted numbers)
- unsrt style (unsorted numbers)
- alpha style (sorted short alphabetics)
- abbrv style (sorted numbers, with abbreviations)

	LAB_ ALPH	SORTED	NAME_ FULL	ATIT_ LOWER	MONTH_ FULL	JOUR_ FULL
PLAIN	0	1	1	1	1	1
UNSRT	0	0	1	1	1	1
ALPHA	1	1	1	1	1	1
ABBRV	0	1	0	1	0	0

1.2 History [2]

12/16/84	(HWT)	Original ‘plain’ version, by Howard Trickey.
12/23/84	(LL)	Some comments made by Leslie Lamport.
2/16/85	(OP)	Changes based on LL’s comments, Oren Patashnik.
2/17/85	(HWT)	Template file and other standard styles made.
3/28/85	(OP)	First release, version 0.98b for BibTeX 0.98f.
5/ 9/85	(OP)	Version 0.98c for BibTeX 0.98i.
1/24/88	(OP)	Version 0.99a for BibTeX 0.99a.
3/23/88	(OP)	Version 0.99b for BibTeX 0.99c.
12/ 8/10	(OP)	Still version 0.99b, as the code itself was unchanged; this release clarified the license.

2 Entry Formatting: Overview

Similar to that recommended by Mary-Claire van Leunen in “A Handbook for Scholars” [3].

2.1 Title Format

Book-like titles are italicized (emphasized) and non-book titles are converted to sentence capitilization [*sic*] (and not enclosed in quotes).

---

[1] 実際の btxbst.doc では、各スタイルごとに define による設定が並んでいますが、ここでは表にまとめてみました。

[2] btxbst.doc では、CPP Variables の設定と “History” との間に “Entry Formatting” の説明が挟まっているのですが、その部分は **2** へと移動させました。また、オリジナルの “History” の部分では、バージョンごとの変更点が列挙されていますが、ここでは割愛しています。

[3] Mary-Claire van Leunen, *A Handbook for Scholars*, 1979. (from: btxdoc.bib)

2.2 Open/Closed Format

This file outputs a \newblock between major blocks of an entry (the name \newblock is analogous to the names \newline and \newpage) so that the user can obtain an "open" format, which has a line break before each block and lines after the first are indented within blocks, by giving the optional \documentstyle argument 'openbib'; The default is the "closed" format—blocks runs together.

2.3 Label Format

- Citation alphabetic label format:

  [Knu73] for single author (or editor or key)

  [AHU83] (first letters of last names) for multiple authors

- Citation label numberic [*sic*] format:

  [number]

2.4 Bibliography Ordering [4]

- Reference list ordering for sorted, alphabetic lables [*sic*]:

  alphabetical by citation label, then by author(s) or whatever passes for author in the absence of one, then by year, then title

- Reference list ordering for sorted, numeric lables [*sic*]:

  alphabetical by author(s) or whatever passes for author in the absence of one, then by year, then title

- Reference list ordering for unsorted:

  by the order cited in the text

3 ENTRY-Variables Declaration

Like Scribe's (according to pages 231–2 of the April '84 edition) [5], but no `full-author` or `editors` fields because BibTeX does name handling. The `annote` field is commented out here because this family doesn't include an annotated bibliography style. And in addition to the fields listed here, BibTeX has a built-in `crossref` field, explained later [6].

---

[4] Patashnik さんが "reference list" とおっしゃるものを、このペーパーでは "bibliography" に統一しています。

ここに示されていますように、ソートの有無による standard styles の Bibliography の種類は、"sorted, alphabetic labels"、"sorted, numeric labels"、"unsorted" の 3 種類です。**1.1** の設定でも、`LAB_ALPH 1` 且つ `SORTED 0` という組み合わせはないのですが、`btxbst.doc` の中では、"unsorted, alphabetic labels" の場合のコードも一応用意されています。しかし毎回「good idea だとは思えないが…」とコメントされています：

- It doesn't seem like a good idea to use an order-of-citation reference list when using alphabetic labels, but when this happens we do things a little differently (☜ p. 6)
- We need the `chop.word` stuff for the dubious unsorted-list-with-labels case. (☜ p. 24)
- It doesn't seem like a particularly good idea to use an order-of-citation reference list when using alphabetic labels, but we need to have a special pass to calculate labels when this happens. (☜ p. 28)
- It still doesn't seem like a good idea to use an order-of-citation reference list when using alphabetic labels, but when this happens we must compute the longest label (☜ p. 32)

[5] Unilogic, Ltd., *Scribe: Document Production System: User Manual*, 1984. (from: `btxdoc.bib`)

[6] 各エントリは、built-in string entry variable である "`sort.key$`" も、有しています。

```
ENTRY
  { address
    % annote
    author
    booktitle
    chapter
    edition
    editor
    howpublished
    institution
    journal
    key
    month
    note
    number
    organization
    pages
    publisher
    school
    series
    title
    type
    volume
    year
  }
```

There are no integer entry variables

```
  {}
```

These string entry variables are used to form the citation label. In a storage pinch, `sort.label` can be easily computed on the fly.

```
#if LAB_ALPH
#if SORTED
  { label extra.label sort.label }
```

It doesn't seem like a good idea to use an order-of-citation reference list when using alphabetic labels, but when this happens we do things a little differently

```
#else !SORTED
  { label }
#endif SORTED
```

```
#else !LAB_ALPH
  { label }
#endif LAB_ALPH
```

4 Output Routine Functions

4.1 Structure of Bibliography Entries

Each entry function[7] starts by calling `output.bibitem`, to write the `\bibitem` and its arguments to the `.BBL` file. Then the various fields are formatted and printed by `output` or `output.check`. Those functions handle the writing of separators (commas, periods, `\newblock`'s), taking care not to do so when they are passed a null string. Finally, `fin.entry` is called to add the final period and finish the entry.

[7] `article` や `book` 等々、`call.type$` に呼ばれて bibliography のエントリを整形する function のことです。7 の冒頭だと“type function”と書かれているのですが、この 4.1 や 7 の半ばあたりでは“entry function”となっています。本ペーパーでは“entry-type function”に統一しています。

A bibliographic reference is formatted into a number of ‘blocks’: in the open format, a block begins on a new line and subsequent lines of the block are indented. A block may contain more than one sentence (well, not a grammatical sentence, but something to be ended with a sentence ending period). The entry functions should call `new.block` whenever a block other than the first is about to be started. They should call `new.sentence` whenever a new sentence is to be started. The output functions will ensure that if two `new.sentence`’s occur without any nonnull string being output between them then there won’t be two periods output. Similarly for two successive `new.block`’s.

4.2 Separators between Fields

The output routines don’t write their argument immediately. Instead, by convention, that argument is saved on the stack to be output next time (when we’ll know what separator needs to come after it). Meanwhile, the output routine has to pop the pending output off the stack, append any needed separator, and write it [8].

To tell which separator is needed, we maintain an `output.state`. It will be one of these values:

- `before.all` – just after the `\bibitem`
- `mid.sentence` – in the middle of a sentence: comma needed if more sentence is output
- `after.sentence` – just after a sentence: period needed
- `after.block` – just after a block (and sentence): period and `\newblock` needed.

Note: These styles don’t use `after.sentence`

```
VAR: output.state : INTEGER           -- state variable for output

INTEGERS { output.state before.all mid.sentence after.sentence after.block }
```

init.state.consts

```
FUNCTION {init.state.consts}
{ #0 'before.all :=
  #1 'mid.sentence :=
  #2 'after.sentence :=
  #3 'after.block :=
}
```

4.3 Output Functions

[T]he variables `s` and `t` are temporary string holders

```
STRINGS { s t }
```

output.nonnull

The `output.nonnull` function saves its argument (assumed to be nonnull) on the stack, and writes the old saved value followed by any needed separator. The ordering of the tests is decreasing frequency of occurrence.

```
output.nonnull(s) ==
 BEGIN
      s := argument on stack
      if output.state = mid.sentence then
          write$(pop() * ", ")
                -- "pop" isn't a function: just use stack top
```

[8] Karl Berry and Oren Patashnik, *State secrets in bibliography-style hacking*, TUGboat 39-3 (2018), p. 275 において、この“国家機密”が解説されています。

```
      else
          if output.state = after.block then
              write$(add.period$(pop()))
              newline$
              write$("\newblock ")
          else
              if output.state = before.all then
                  write$(pop())
              else        -- output.state should be after.sentence
                  write$(add.period$(pop()) * " ")
              fi
          fi
          output.state := mid.sentence
      fi
      push s on stack
 END
```

```
FUNCTION {output.nonnull}
{. . . . .
```

output The output function calls output.nonnull if its argument is non-empty; its argument may be a missing field (thus, not necessarily a string)

```
output(s) ==
 BEGIN
      if not empty$(s) then output.nonnull(s)
      fi
 END
```

```
FUNCTION {output}
{. . . . .
```

output.check The output.check function is the same as the output function except that, if necessary, output.check warns the user that the t field shouldn't be empty (this is because it probably won't be a good reference without the field; the entry functions try to make the formatting look reasonable even when such fields are empty).

```
output.check(s,t) ==
 BEGIN
      if empty$(s) then
          warning$("empty " * t * " in " * cite$)
      else output.nonnull(s)
      fi
 END
```

```
FUNCTION {output.check}
{. . . . .
```

output.bibitem The output.bibitem function writes the \bibitem for the current entry (the label should already have been set up), and sets up the separator state for the output functions. And, it leaves a string on the stack as per the output convention.

```
output.bibitem ==
 BEGIN
      newline$
      write$("\bibitem[")      % for alphabetic labels,
      write$(label)            % these three lines
      write$("]{")             % are used
      write$("\bibitem{")              % this line for numeric labels
      write$(cite$)
      write$("}")
      push "" on stack
      output.state := before.all
```

```
 END
```

```
FUNCTION {output.bibitem}
{. . . . .
```

fin.entry

The fin.entry function finishes off an entry by adding a period to the string remaining on the stack. If the state is still before.all then nothing was produced for this entry, so the result will look bad, but the user deserves it. (We don't omit the whole entry because the entry was cited, and a bibitem is needed to define the citation label.)

```
fin.entry ==
 BEGIN
      write$(add.period$(pop()))
      newline$
 END
```

```
FUNCTION {fin.entry}
{. . . . .
```

new.block

new.sentence

The new.block function prepares for a new block to be output, and new.sentence prepares for a new sentence.

```
new.block ==
 BEGIN
      if output.state <> before.all then
          output.state := after.block
      fi
 END
```

```
new.sentence ==
 BEGIN
      if output.state <> after.block then
          if output.state <> before.all then
              output.state :=  after.sentence
          fi
      fi
 END
```

```
FUNCTION {new.block}
{. . . . .
```

```
FUNCTION {new.sentence}
{. . . . .
```

not

and

or

These three functions pop one or two (integer) arguments from the stack and push a single one, either 0 or 1.

The 'skip$ in the 'and' and 'or' functions are used because the corresponding if$ would be idempotent

```
FUNCTION {not}
{ { #0 }
    { #1 }
  if$
}
```

```
FUNCTION {and}
{ 'skip$
    { pop$ #0 }
```

```
  if$
}
```

```
FUNCTION {or}
{ { pop$ #1 }
    'skip$
  if$
}
```

new.block.checka
new.block.checkb

Sometimes we begin a new block only if the block will be big enough. The new.block.checka function issues a new.block if its argument is nonempty; new.block.checkb does the same if either of its TWO arguments is nonempty.

```
FUNCTION {new.block.checka}
{ empty$
    'skip$
    'new.block
  if$
}
```

```
FUNCTION {new.block.checkb}
{ empty$
  swap$ empty$
  and
    'skip$
    'new.block
  if$
}
```

new.sentence.checka
new.sentence.checkb

The new.sentence.check functions are analogous.

```
FUNCTION {new.sentence.checka}
{. . . . .
```

```
FUNCTION {new.sentence.checkb}
{. . . . .
```

5 Inner Functions for Formatting

Here are some functions for formatting chunks of an entry. By convention they either produce a string that can be followed by a comma or period (using add.period$, so it is OK to end in a period), or they produce the null string.

5.1 Utility/Helper Functions

field.or.null

A useful utility is the field.or.null function, which checks if the argument is the result of pushing a 'missing' field (one for which no assignment was made when the current entry was read in from the database) or the result of pushing a string having no non-white-space characters. It returns the null string if so, otherwise it returns the field string. Its main (but not only) purpose is to guarantee that what's left on the stack is a string rather than a missing field.

```
field.or.null(s) ==
 BEGIN
      if empty$(s) then return ""
      else return s
 END
```

The ‘pop$’ in this function gets rid of the duplicate ‘empty’ value and the ‘skip$’ returns the duplicate field value

```
FUNCTION {field.or.null}
{ duplicate$ empty$
    { pop$ "" }
    'skip$
  if$
}
```

emphasize

Another helper function is emphasize, which returns the argument emphazised [*sic*], if that is non-empty, otherwise it returns the null string. Italic corrections aren't used, so this function should be used when punctation will follow the result.

```
emphasize(s) ==
 BEGIN
      if empty$(s) then return ""
      else return "{\em " * s * "}"
 END
```

```
FUNCTION {emphasize}
{ duplicate$ empty$
    { pop$ "" }
    { "{\em " swap$ * "}" * }
  if$
}
```

5.2 Name Formatting Functions

format.names

The format.names function formats the argument (which should be in BibTeX name format) into“First Von Last, Junior”, separated by commas and with an “and” before the last (but ending with “et al.” if the last of multiple authors is “others”). This function's argument should always contain at least one name.

```
VAR: nameptr, namesleft, numnames: INTEGER
pseudoVAR: nameresult: STRING         (it's what's accumulated on the stack)
```

```
INTEGERS { nameptr namesleft numnames }
```

```
format.names(s) ==
 BEGIN
      nameptr := 1
      numnames := num.names$(s)
      namesleft := numnames
      while namesleft > 0
        do
                              % for full names:
          t := format.name$(s, nameptr, "{ff~}{vv~}{ll}{, jj}")
                              % for abbreviated first names:
          t := format.name$(s, nameptr, "{f.~}{vv~}{ll}{, jj}")
          if nameptr > 1 then
              if namesleft > 1 then nameresult := nameresult * ", " * t
              else if numnames > 2
                     then nameresult := nameresult * ","
                   fi
                   if t = "others"
                     then nameresult := nameresult * " et~al."
                     else nameresult := nameresult * " and " * t
                   fi
              fi
          else nameresult := t
          fi
          nameptr := nameptr + 1
```

```
          namesleft := namesleft - 1
        od
      return nameresult
 END
```

```
FUNCTION {format.names}
{. . . . .
```

format.authors The format.authors function returns the result of format.names(author) if the author is present, or else it returns the null string

```
format.authors ==
 BEGIN
      if empty$(author) then return ""
      else return format.names(author)
      fi
 END
```

```
FUNCTION {format.authors}
{. . . . .
```

format.editors Format.editors is like format.authors, but it uses the editor field, and appends ", editor" or ", editors"

```
format.editors ==
 BEGIN
      if empty$(editor) then return ""
      else
          if num.names$(editor) > 1 then
              return format.names(editor) * ", editors"
          else
              return format.names(editor) * ", editor"
          fi
      fi
 END
```

```
FUNCTION {format.editors}
{. . . . .
```

Other formatting functions are similar, so no "comment version" will be given for them.

5.3 Title Formatting Functions

format.title The format.title function is used for non-book-like titles. For most styles we convert to lowercase (except for the very first letter, and except for the first one after a colon (followed by whitespace)), and hope the user has brace-surrounded words that need to stay capitilized [*sic*]; for some styles, however, we leave it as it is in the database [9].

```
FUNCTION {format.title}
{ title empty$
    { "" }
#if ATIT_LOWER
    { title "t" change.case$ }
#else
    'title
#endif ATIT_LOWER
```

[9] 確かに #else branch も用意されてはいますが、standard styles では ATIT_LOWER はどれも "1" に設定されていますので、そちらが選択されることはありません。そのため、*TLC* では "how to eliminate the (sometimes unpleasant) standard BibTeX style feature that transforms titles to lowercase" が "to make slight changes" の一例として挙げられています。*TLC*, 1st, 13.8; 2nd, 13.6.3。

```
  if$
}
```

global.max$
entry.max$

By default, BibTeX sets the global integer variable global.max$ to the BibTeX constant glob_str_size, the maximum length of a global string variable. Analogously, BibTeX sets the global integer variable entry.max$ to ent_str_size, the maximum length of an entry string variable. The style designer may change these if necessary (but this is unlikely)

n.dashify

The n.dashify function makes each single '-' in a string a double '--' if it's not already

```
pseudoVAR: pageresult: STRING          (it's what's accumulated on the stack)
```

```
n.dashify(s) ==
 BEGIN
      t := s
      pageresult := ""
      while (not empty$(t))
        do
          if (first character of t = "-")
            then
              if (next character isn't)
                then
                  pageresult := pageresult * "--"
                  t := t with the "-" removed
                else
                  while (first character of t = "-")
                    do
                      pageresult := pageresult * "-"
                      t := t with the "-" removed
                    od
              fi
            else
              pageresult := pageresult * the first character
              t := t with the first character removed
          fi
        od
      return pageresult
 END
```

```
FUNCTION {n.dashify}
{. . . . .
```

format.date

The format.date function is for the month and year, but we give a warning if there's an empty year but the month is there, and we return the empty string if they're both empty.

```
FUNCTION {format.date}
{. . . . .
```

format.btitle

The format.btitle is for formatting the title field when it is a book-like entry—the style used here keeps it in uppers-and-lowers and emphasizes it.

```
FUNCTION {format.btitle}
{ title emphasize
}
```

5.4 Series Formatting Functions

tie.or.space.connect

For several functions we'll need to connect two strings with a tie (~) if the second one isn't very long (fewer than 3 characters). The tie.or.space.connect func-

tion does that. It concatenates the two strings on top of the stack, along with either a tie or space between them, and puts this concatenation back onto the stack:

```
tie.or.space.connect(str1,str2) ==
 BEGIN
      if text.length$(str2) < 3
        then return the concatenation of str1, "~", and str2
        else return the concatenation of str1, " ", and str2
 END
```

```
FUNCTION {tie.or.space.connect}
{. . . . .
```

either.or.check

The `either.or.check` function complains if both fields or an either-or pair are nonempty.

```
either.or.check(t,s) ==
 BEGIN
      if empty$(s) then
          warning$("can't use both " * t * " fields in " * cite$)
      fi
 END
```

```
FUNCTION {either.or.check}
{. . . . .
```

format.bvolume

The `format.bvolume` function is for formatting the volume and perhaps series name of a multivolume work. If both a `volume` and a `series` field are there, we assume the `series` field is the title of the whole multivolume work (the `title` field should be the title of the thing being referred to), and we add an "of ⟨`series`⟩". This function is called in mid-sentence.

```
FUNCTION {format.bvolume}
{ volume empty$
    { "" }
    { "volume" volume tie.or.space.connect
      series empty$
        'skip$
        { " of " * series emphasize * }
      if$
      "volume and number" number either.or.check
    }
  if$
}
```

format.number.series

The `format.number.series` function is for formatting the series name and perhaps number of a work in a series. This function is similar to `format.bvolume`, although for this one the series must exist (and the volume must not exist). If the `number` field is empty we output either the `series` field unchanged if it exists or else the null string. If both the `number` and `series` fields are there we assume the `series` field gives the name of the whole series (the `title` field should be the title of the work being one referred to), and we add an "in ⟨`series`⟩". We capitilize [*sic*] Number when this function is used at the beginning of a block.

```
FUNCTION {format.number.series}
{ volume empty$
    { number empty$
        { series field.or.null }
        { output.state mid.sentence =
            { "number" }
            { "Number" }
          if$
          number tie.or.space.connect
```

```
        series empty$
          { "there's a number but no series in " cite$ * warning$ }
          { " in " * series * }
        if$
      }
    if$
  }
  { "" }
 if$
}
```

format.edition

The format.edition function appends " edition" to the edition, if present. We lowercase the edition (it should be something like "Third"), because this doesn't start a sentence.

```
FUNCTION {format.edition}
{ edition empty$
  { "" }
  { output.state mid.sentence =
    { edition "l" change.case$ " edition" * }
    { edition "t" change.case$ " edition" * }
   if$
  }
 if$
}
```

5.5 Page-Range Formatting Functions

The format.pages function is used for formatting a page range in a book (and in rare circumstances, an article).

multi.page.check

The multi.page.check function examines the page field for a "-" or "," or "+" so that format.pages can use "page" instead of "pages" if none exists.

Note: global.max$ here means "take the rest of the string"

```
VAR: multiresult: INTEGER     (actually, a boolean)
```

```
INTEGERS { multiresult }
```

```
multi.page.check(s) ==
 BEGIN
      t := s
      multiresult := false
      while ((not multiresult) and (not empty$(t)))
        do
          if (first character of t = "-" or "," or "+")
            then multiresult := true
            else t := t with the first character removed
          fi
        od
      return multiresult
 END
```

```
FUNCTION {multi.page.check}
{ 't :=
  #0 'multiresult :=
    { multiresult not
      t empty$ not
      and
    }
    { t #1 #1 substring$
      duplicate$ "-" =
      swap$ duplicate$ "," =
```

```
      swap$ "+" =
      or or
        { #1 'multiresult := }
        { t #2 global.max$ substring$ 't := }
      if$
    }
  while$
  multiresult
}
```

format.pages

This function doesn't begin a sentence so "pages" isn't capitalized. Other functions that use this should keep that in mind.

```
FUNCTION {format.pages}
{ pages empty$
    { "" }
    { pages multi.page.check
        { "pages" pages n.dashify tie.or.space.connect }
        { "page" pages tie.or.space.connect }
      if$
    }
  if$
}
```

format.vol.num.pages

The format.vol.num.pages function is for the volume, number, and page range of a journal article. We use the format: vol(number):pages, with some variations for empty fields. This doesn't begin a sentence.

```
FUNCTION {format.vol.num.pages}
{. . . . .
```

format.chapter.pages

The format.chapter.pages, if the chapter is present, puts whatever is in the type field (or else "chapter" if type is empty) in front of a chapter number. It then appends the pages, if present. This doesn't begin a sentence.

```
FUNCTION {format.chapter.pages}
{. . . . .
```

5.6 Miscellaneous Functions

format.in.ed.booktitle

The format.in.ed.booktitle function is used for starting out a sentence that begins "In ⟨booktitle⟩", putting an editor before the title if one exists.

```
FUNCTION {format.in.ed.booktitle}
{. . . . .
```

empty.misc.check

The function empty.misc.check complains if all six fields are empty, and if there's been no sorting or alphabetic-label complaint.

```
FUNCTION {empty.misc.check}
{. . . . .
```

format.thesis.type

The function format.thesis.type returns either the (case-changed) type field, if it is defined, or else the default string already on the stack (like "Master's thesis" or "PhD thesis").

```
FUNCTION {format.thesis.type}
{. . . . .
```

format.tr.number

The function format.tr.number makes a string starting with "Technical Report" (or type, if that field is defined), followed by the number if there is one; it returns the starting part (with a case change) even if there is no number. This is used at the beginning of a sentence.

```
FUNCTION {format.tr.number}
{. . . . .
```

6 Cross-referencing Functions

Now come the cross-referencing functions (these are invoked because one entry in the database file(s) cross-references another, by giving the other entry's database key in a 'crossref' field). This feature allows one or more titled things that are part of a larger titled thing to cross-reference the larger thing. These styles allow for five posibilities [*sic*]: (1) an ARTICLE may cross-reference an ARTICLE; (2) a BOOK, (3) INBOOK, or (4) INCOLLECTION may cross-reference a BOOK; or (5) an INPROCEEDINGS may cross-reference a PROCEEDINGS. Each of these is explained in more detail later.

format.article.crossref

An ARTICLE entry type may cross reference another ARTICLE (this is intended for when an entire journal is devoted to a single topic—but since there is no JOURNAL entry type, the journal, too, should be classified as an ARTICLE but without the author and title fields). This will result in two warning messages for the journal's entry if it's included in the reference list, but such is life [10].

```
format.article.crossref ==
 BEGIN
      if empty$(key) then
          if empty$(journal) then
              warning$("need key or journal for " * cite$ *
                                                  " to crossref " * crossref)
              return(" \cite{" * crossref * "}")
          else
              return("In " * emphazise.correct (journal) *
                                                  " \cite{" * crossref * "}")
              fi
      else
          return("In " * key * " \cite{" * crossref * "}")
      fi
 END
```

```
FUNCTION {format.article.crossref}
{ key empty$
   { journal empty$
      { "need key or journal for " cite$ * " to crossref " * crossref *
        warning$
        ""
      }
      { "In {\em " journal * "\/}" * }
     if$
   }
   { "In " key * }
  if$
  " \cite{" * crossref * "}" *
}
```

The other cross-referencing functions are similar, so no "comment version" will be given for them.

format.crossref.editor

We use just the last names of editors for a cross reference: either "editor", or "editor1 and editor2", or "editor1 et~al." depending on whether there are one, or

[10] "\em ...\/" という処理を "emphazise.correct" と称してますね (typo ですが)。あと、もっと上のほうの pseudo code でも、"END" が抜けていたり、"" が抜けていたところがありましたが、それぞれ補いました。まぁ、but such is life ですから。

two, or more than two editors.

```
FUNCTION {format.crossref.editor}
{ editor #1 "{vv~}{ll}" format.name$
  editor num.names$ duplicate$
  #2 >
    { pop$ " et~al." * }
    { #2 <
        'skip$
        { editor #2 "{ff }{vv }{ll}{ jj}" format.name$ "others" =
            { " et~al." * }
            { " and " * editor #2 "{vv~}{ll}" format.name$ * }
          if$
        }
      if$
    }
  if$
}
```

format.book.crossref

A BOOK (or INBOOK) entry type (assumed to be for a single volume in a multivolume work) may cross reference another BOOK (the entire multivolume). Usually there will be an editor, in which case we use that to construct the cross reference; otherwise we use a nonempty `key` field or else the `series` field (since the series gives the title of the multivolume work).

```
FUNCTION {format.book.crossref}
{ volume empty$
    { "empty volume in " cite$ * "'s crossref of " * crossref * warning$
      "In "
    }
    { "Volume" volume tie.or.space.connect
      " of " *
    }
  if$
  editor empty$
  editor field.or.null author field.or.null =
  or
    { key empty$
        { series empty$
            { "need editor, key, or series for " cite$ * " to crossref " *
              crossref * warning$
              "" *
            }
            { "{\em " * series * "\/}" * }
          if$
        }
        { key * }
      if$
    }
    { format.crossref.editor * }
  if$
  " \cite{" * crossref * "}" *
}
```

at.incoll.inproc.crossref

An INCOLLECTION entry type may cross reference a BOOK (assumed to be the collection), or an INPROCEEDINGS may cross reference a PROCEEDINGS. Often there will be an `editor`, in which case we use that to construct the cross reference; otherwise we use a nonempty `key` field or else the `booktitle` field (which gives the cross-referenced work's title).

```
FUNCTION {format.incoll.inproc.crossref}
{ editor empty$
  editor field.or.null author field.or.null =
  or
```

```
    { key empty$
       { booktitle empty$
          { "need editor, key, or booktitle for " cite$ * " to crossref " *
            crossref * warning$
            ""
          }
          { "In {\em " booktitle * "\/}" * }
        if$
       }
       { "In " key * }
      if$
    }
    { "In " format.crossref.editor * }
  if$
  " \cite{" * crossref * "}" *
}
```

7 Entry-Type Functions

Now we define the type functions for all entry types that may appear in the .BIB file—e.g., functions like ‘article’ and ‘book’. These are the routines that actually generate the .BBL-file output for the entry. These must all precede the READ command. In addition, the style designer should have a function ‘default.type’ for unknown types.

Note: The fields (within each list) are listed in order of appearance, except as described for an ‘inbook’ or a ‘proceedings’.

article

The article function is for an article in a journal. An article may CROSSREF another article.

- Required fields: author, title, journal, year
- Optional fields: volume, number, pages, month, note

```
article ==
 BEGIN
      output.bibitem
      output.check(format.authors,"author")
      new.block
      output.check(format.title,"title")
      new.block
      if missing$(crossref) then
          output.check(emphasize(journal),"journal")
          output(format.vol.num.pages)
          output.check(format.date,"year")
      else
          output.nonnull(format.article.crossref)
          output(format.pages)
      fi
      new.block
      output(note)
      fin.entry
 END
```

```
FUNCTION {article}
{ output.bibitem
  format.authors "author" output.check
  new.block
  format.title "title" output.check
  new.block
  crossref missing$
    { journal emphasize "journal" output.check
```

```
      format.vol.num.pages output
      format.date "year" output.check
    }
    { format.article.crossref output.nonnull
      format.pages output
    }
  if$
  new.block
  note output
  fin.entry
}
```

book The book function is for a whole book. A book may CROSSREF another book.

- Required fields: author or editor, title, publisher, year
- Optional fields: volume or number, series, address, edition, month, note

```
book ==
 BEGIN
      if empty$(author) then output.check(format.editors,"author and editor")
      else    output.check(format.authors,"author")
              if missing$(crossref) then
                  either.or.check("author and editor",editor)
              fi
      fi
      new.block
      output.check(format.btitle,"title")
      if missing$(crossref) then
          output(format.bvolume)
          new.block
          output(format.number.series)
          new.sentence
          output.check(publisher,"publisher")
          output(address)
      else
          new.block
          output.nonnull(format.book.crossref)
      fi
      output(format.edition)
      output.check(format.date,"year")
      new.block
      output(note)
      fin.entry
 END
```

```
FUNCTION {book}
{ output.bibitem
  author empty$
    { format.editors "author and editor" output.check }
    { format.authors output.nonnull
      crossref missing$
        { "author and editor" editor either.or.check }
        'skip$
      if$
    }
  if$
  new.block
  format.btitle "title" output.check
  crossref missing$
    { format.bvolume output
      new.block
      format.number.series output
      new.sentence
      publisher "publisher" output.check
```

```
      address output
    }
    { new.block
      format.book.crossref output.nonnull
    }
  if$
  format.edition output
  format.date "year" output.check
  new.block
  note output
  fin.entry
}
```

The other entry functions are all quite similar, so no "comment version" will be given for them.

booklet

A booklet is a bound thing without a publisher or sponsoring institution.

- Required: title
- Optional: author, howpublished, address, month, year, note

```
FUNCTION {booklet}
{. . . . .
```

conference

For the conference entry type, see inproceedings.

inbook

An inbook is a piece of a book: either a chapter and/or a page range. It may CROSSREF a book. If there's no volume field, the type field will come before number and series.

- Required: author or editor, title, chapter and/or pages, publisher, year
- Optional: volume or number, series, type, address, edition, month, note

```
FUNCTION {inbook}
{. . . . .
```

incollection

An incollection is like inbook, but where there is a separate title for the referenced thing (and perhaps an editor for the whole). An incollection may CROSSREF a book.

- Required: author, title, booktitle, publisher, year
- Optional: editor, volume or number, series, type, chapter, pages, address, edition, month, note

```
FUNCTION {incollection}
{. . . . .
```

inproceedings

An inproceedings is an article in a conference proceedings, and it may CROSSREF a proceedings. If there's no address field, the month (& year) will appear just before note.

- Required: author, title, booktitle, year
- Optional: editor, volume or number, series, pages, address, month, organization, publisher, note

```
FUNCTION {inproceedings}
{. . . . .
```

conference

The conference function is included for Scribe compatibility.

```
FUNCTION {conference} { inproceedings }
```

manual

A manual is technical documentation.

- Required: title
- Optional: author, organization, address, edition, month, year, note

```
FUNCTION {manual}
{. . . . .
```

mastersthesis

A mastersthesis is a Master's thesis.

- Required: author, title, school, year
- Optional: type, address, month, note

```
FUNCTION {mastersthesis}
{. . . . .
```

misc

A misc is something that doesn't fit elsewhere.

- Required: at least one of the 'optional' fields
- Optional: author, title, howpublished, month, year, note

```
FUNCTION {misc}
{. . . . .
```

phdthesis

A phdthesis is like a mastersthesis.

- Required: author, title, school, year
- Optional: type, address, month, note

```
FUNCTION {phdthesis}
{. . . . .
```

proceedings

A proceedings is a conference proceedings.

If there is an organization but no editor field, the organization will appear as the first optional field (we try to make the first block nonempty); if there's no address field, the month (& year) will appear just before note.

- Required: title, year
- Optional: editor, volume or number, series, address, month, organization, publisher, note

```
FUNCTION {proceedings}
{. . . . .
```

techreport

A techreport is a technical report.

- Required: author, title, institution, year
- Optional: type, number, address, month, note

```
FUNCTION {techreport}
{. . . . .
```

unpublished

An unpublished is something that hasn't been published.

- Required: author, title, note
- Optional: month, year

```
FUNCTION {unpublished}
{. . . . .
```

default.type

We use entry type 'misc' for an unknown type; BibTeX gives a warning.

```
FUNCTION {default.type} { misc }
```

8 MACROs for Months and Journals

Here are macros for common things that may vary from style to style. Users are encouraged to use these macros.

8.1 Abbreviations for Months

Months are either written out in full or abbreviated

```
#if MONTH_FULL

MACRO {jan} {"January"}
. . . . .
MACRO {dec} {"December"}
```

```
#else !MONTH_FULL

MACRO {jan} {"Jan."}
. . . . .
MACRO {dec} {"Dec."}

#endif MONTH_FULL
```

8.2 Abbreviations for Journals

Journals are either written out in full or abbreviated; the abbreviations are like those found in ACM publications.

To get a completely different set of abbreviations, it may be best to make a separate `.bib` file with nothing but those abbreviations; users could then include that file name as the first argument to the `\bibliography` command

```
#if JOUR_FULL

MACRO {acmcs} {"ACM Computing Surveys"}
. . . . .
MACRO {tcs} {"Theoretical Computer Science"}
```

```
#else !JOUR_FULL

MACRO {acmcs} {"ACM Comput. Surv."}
. . . . .
MACRO {tcs} {"Theoretical Comput. Sci."}

#endif JOUR_FULL
```

9 READing-in the `.bib` File

Now we read in the `.BIB` entries.

```
READ
```

10 Preliminary Calculations for Labeling and Sorting

10.1 Utility Functions

sortify The `sortify` function converts to lower case after `purify$`ing; it's used in sorting and in computing alphabetic labels after sorting

```
#if SORTED

FUNCTION {sortify}
{ purify$
  "l" change.case$
}
```

chop.word

The `chop.word(w,len,s)` function returns either `s` or, if the first `len` letters of `s` equals `w` (this comparison is done in the third line of the function's definition), it returns that part of `s` after `w`.

```
INTEGERS { len }

FUNCTION {chop.word}
{ 's :=
  'len :=
  s #1 len substring$ =
    { s len #1 + global.max$ substring$ }
    's
  if$
}
```

We need the `chop.word` stuff for the dubious unsorted-list-with-labels case.

```
#else !SORTED
#if LAB_ALPH

INTEGERS { len }

FUNCTION {chop.word}
{ 's :=
  'len :=
  s #1 len substring$ =
    { s len #1 + global.max$ substring$ }
    's
  if$
}

#endif LAB_ALPH
#endif SORTED
```

10.2 For Alphabetic Labels: `calc.label`

This long comment applies only to alphabetic labels

10.2.1 Helper Function

format.lab.names

The `format.lab.names` function makes a short label by using the initials of the von and Last parts of the names (but if there are more than four names, (i.e., people) it truncates after three and adds a superscripted "+"; it also adds such a "+" if the last of multiple authors is "`others`"). If there is only one name, and its von and Last parts combined have just a single name-token ("Knuth" has a single token, "Brinch Hansen" has two), we take the first three letters of the last name. The boolean `et.al.char.used` tells whether we've used a superscripted "+", so that we know whether to include a LaTeX macro for it.

```
format.lab.names(s) ==
 BEGIN
      numnames := num.names$(s)
      if numnames > 1 then
          if numnames > 4 then
              namesleft := 3
```

```
          else
              namesleft := numnames
          nameptr := 1
          nameresult := ""
          while namesleft > 0
            do
              if (name_ptr = numnames) and
                   format.name$(s, nameptr, "{ff }{vv }{ll}{ jj}") = "others"
                 then nameresult := nameresult * "{\etalchar{+}}"
                      et.al.char.used := true
                 else nameresult := nameresult *
                              format.name$(s, nameptr, "{v{}}{l{}}")
              nameptr := nameptr + 1
              namesleft := namesleft - 1
            od
          if numnames > 4 then
              nameresult := nameresult * "{\etalchar{+}}"
              et.al.char.used := true
      else
          t := format.name$(s, 1, "{v{}}{l{}}")
          if text.length$(t) < 2 then % there's just one name-token
              nameresult := text.prefix$(format.name$(s,1,"{ll}"),3)
          else
              nameresult := t
          fi
      fi
      return nameresult
 END
```

nitialize.et.al.char.used

```
#if LAB_ALPH

INTEGERS { et.al.char.used }

FUNCTION {initialize.et.al.char.used}
{ #0 'et.al.char.used :=
}

EXECUTE {initialize.et.al.char.used}
```

```
FUNCTION {format.lab.names}
{ 's :=
  s num.names$ 'numnames :=
  numnames #1 >
    { numnames #4 >
        { #3 'namesleft := }
        { numnames 'namesleft := }
      if$
      #1 'nameptr :=
      ""
        { namesleft #0 > }
        { nameptr numnames =
            { s nameptr "{ff }{vv }{ll}{ jj}" format.name$ "others" =
                { "{\etalchar{+}}" *
                  #1 'et.al.char.used :=
                }
                { s nameptr "{v{}}{l{}}" format.name$ * }
              if$
            }
            { s nameptr "{v{}}{l{}}" format.name$ * }
          if$
          nameptr #1 + 'nameptr :=
          namesleft #1 - 'namesleft :=
        }
      while$
      numnames #4 >
```

```
        { "{\etalchar{+}}" *
          #1 'et.al.char.used :=
        }
        'skip$
      if$
    }
    { s #1 "{v{}}{l{}}" format.name$
      duplicate$ text.length$ #2 <
        { pop$ s #1 "{ll}" format.name$ #3 text.prefix$ }
        'skip$
      if$
    }
  if$
}
```

10.2.2 Four Auxiliary Functions

Exactly what fields we look at in constructing the primary part of the label depends on the entry type[11]; this selectivity (as opposed to, say, always looking at author, then editor, then key) helps ensure that "ignored" fields, as described in the LaTeX book, really are ignored. Note that MISC is part of the deepest 'else' clause in the nested part of calc.label; thus, any unrecognized entry type in the database is handled correctly.

There is one auxiliary function for each of the four different sequences of fields we use. The first of these functions looks at the author field, and then, if necessary, the key field. The other three functions, which might look at two fields and the key field, are similar, except that the key field takes precedence over the organization field (for labels—not for sorting).

author.key.label

```
FUNCTION {author.key.label}
{ author empty$
    { key empty$
#if SORTED
        { cite$ #1 #3 substring$ }
#else !SORTED % need warning here because we won't give it later
        { "for label, need author or key in " cite$ * warning$
          cite$ #1 #3 substring$
        }
#endif SORTED
        { key #3 text.prefix$ }
      if$
    }
    { author format.lab.names }
  if$
}
```

(11) *BibTeXing* [1a] の "4 Helpful Hints" の 15. と 16. に、以下のような説明があります：

15. ではまず、

For most entry types the "author" information is simply the author field. However:

- For the BOOK and INBOOK entry types it's the author field, but if there's no author then it's the editor field;
- for the MANUAL entry type it's the author field, but if there's no author then it's the organization field; and
- for the PROCEEDINGS entry type it's the editor field, but if there's no editor then it's the organization field.

とあって、続けて、16. で、

When creating a label, the alpha style uses the "author" information described above, but with a slight change—for the MANUAL and PROCEEDINGS entry types, the key field takes precedence over the organization field.

と補足されています。

author.editor.key.label
```
FUNCTION {author.editor.key.label}
{. . . . .
```

or.key.organization.label
```
FUNCTION {author.key.organization.label}
{. . . . .
```

or.key.organization.label
```
FUNCTION {editor.key.organization.label}
{. . . . .
```

10.2.3 Function calc.label

calc.label

The calc.label function calculates the preliminary label of an entry, which is formed by taking three letters of information from the author or editor or key or organization field (depending on the entry type and on what's empty, but ignoring a leading "The␣" in the organization), and appending the last two characters (digits) of the year. It is an error if the appropriate fields among author, editor, organization, and key are missing, and we use the first three letters of the cite$ in desperation when this happens. The resulting label has the year part, but not the name part, purify$ed (purify$ing the year allows some sorting shenanigans by the user).

This function also calculates the version of the label to be used in sorting.

The final label may need a trailing 'a', 'b', etc., to distinguish it from otherwise identical labels, but we can't calculated [*sic*] those "extra.label"s until after sorting.

```
calc.label ==
 BEGIN
      if type$ = "book" or "inbook" then
          author.editor.key.label
      else if type$ = "proceedings" then
          editor.key.organization.label
      else if type$ = "manual" then
          author.key.organization.label
      else
          author.key.label
      fi fi fi
      label := label * substring$(purify$(field.or.null(year)), -1, 2)
              % assuming we will also sort, we calculate a sort.label
      sort.label := sortify(label), but use the last four, not two, digits
 END
```

```
FUNCTION {calc.label}
{ type$ "book" =
  type$ "inbook" =
  or
    'author.editor.key.label
    { type$ "proceedings" =
        'editor.key.organization.label
        { type$ "manual" =
            'author.key.organization.label
            'author.key.label
          if$
        }
      if$
    }
  if$
  duplicate$
  year field.or.null purify$ #-1 #2 substring$
  *
  'label :=
  year field.or.null purify$ #-1 #4 substring$
```

```
  *
  sortify 'sort.label :=
}
```

It doesn't seem like a particularly good idea to use an order-of-citation reference list when using alphabetic labels, but we need to have a special pass to calculate labels when this happens.

```
#if !SORTED
ITERATE {calc.label}

#endif !SORTED

#endif LAB_ALPH
```

10.3 For Sorting: presort

When sorting, we compute the sortkey by executing "presort" on each entry. The presort key contains a number of "sortify"ed strings, concatenated with multiple blanks between them. This makes things like "brinch␣␣per" come before "brinch␣hansen␣␣per".

The fields used here are: the sort.label for alphabetic labels (as set by calc.label), followed by the author names (or editor names or organization (with a leading "The␣" removed) or key field, depending on entry type and on what's empty), followed by year, followed by the first bit of the title (chopping off a leading "The␣", "A␣", or "An␣")[12].

Names are formatted: Von Last First Junior.

The names within a part will be separated by a single blank (such as "brinch␣hansen"), two will separate the name parts themselves (except the von and last), three will separate the names, four will separate the names from year (and from label, if alphabetic), and four will separate year from title[13].

10.3.1 Helper Functions

sort.format.names

The sort.format.names function takes an argument that should be in BibTeX name format, and returns a string containing "␣␣␣"-separated names in the format described above. The function is almost the same as format.names.

---

[12] *BibTeXing* [1a] の "2.2 Changes to the standard styles" の 1. に、次のような説明があります：

> In general, sorting is now by "author", then year, then title—the old versions didn't use the year field. (The alpha style, however, sorts first by label, then "author", year, and title.) The quotes around author mean that some entry types might use something besides the author, like the editor or organization.

[13] つまり、例えば、

- author = "Aho, Alfred V. and Kernighan, Brian W. and Weinberger, Peter J."
- title = "The AWK Programming Language"
- year = 1988

であるとき、calc.label が生成するこのエントリの label は "AKW88" で、sort.label は "akw1988" になります。

それに対して、presort がこのエントリの sort.key$ に代入する文字列は、alpha.bst の場合は、

- akw1988␣␣␣␣aho␣␣alfred␣v␣␣␣kernighan␣␣brian␣w␣␣␣weinberger␣␣peter␣j␣␣␣␣1988␣␣␣␣awk␣programming␣language

という風になり、plain.bst の場合には、

- aho␣␣alfred␣v␣␣␣kernighan␣␣brian␣w␣␣␣weinberger␣␣peter␣j␣␣␣␣1988␣␣␣␣awk␣programming␣language

になるということです。

```
#if SORTED

FUNCTION {sort.format.names}
{ 's :=
  #1 'nameptr :=
  ""
  s num.names$ 'numnames :=
  numnames 'namesleft :=
    { namesleft #0 > }
    { nameptr #1 >
        { "   " * }
        'skip$
      if$
#if NAME_FULL
      s nameptr "{vv{ } }{ll{ }}{  ff{ }}{  jj{ }}" format.name$ 't :=
#else
      s nameptr "{vv{ } }{ll{ }}{  f{ }}{  jj{ }}" format.name$ 't :=
#endif NAME_FULL
      nameptr numnames = t "others" = and
        { "et al" * }
        { t sortify * }
      if$
      nameptr #1 + 'nameptr :=
      namesleft #1 - 'namesleft :=
    }
  while$
}
```

sort.format.title

The `sort.format.title` function returns the argument, but first any leading "A␣"'s, "An␣"'s, or "The␣"'s are removed. The `chop.word` function uses `s`, so we need another string variable, `t`

```
FUNCTION {sort.format.title}
{ 't :=
  "A " #2
    "An " #3
      "The " #4 t chop.word
    chop.word
  chop.word
  sortify
  #1 global.max$ substring$
}
```

10.3.2 Four Auxiliary Functions

The auxiliary functions here, for the `presort` function, are analogous to the ones for `calc.label`; the same comments apply, except that the `organization` field takes precedence here over the `key` field. For sorting purposes, we still remove a leading "The␣" from the `organization` field.

author.sort

```
FUNCTION {author.sort}
{ author empty$
    { key empty$
        { "to sort, need author or key in " cite$ * warning$
          ""
        }
        { key sortify }
      if$
    }
    { author sort.format.names }
  if$
}
```

author.editor.sort
```
FUNCTION {author.editor.sort}
{. . . . .
```

author.organization.sort
```
FUNCTION {author.organization.sort}
{. . . . .
```

editor.organization.sort
```
FUNCTION {editor.organization.sort}
{. . . . .
```

10.3.3 Function presort

presort

There is a limit, entry.max$, on the length of an entry string variable (which is what its sort.key$ is), so we take at most that many characters of the constructed key, and hope there aren't many references that match to that many characters!

```
FUNCTION {presort}
#if LAB_ALPH
{ calc.label
  sort.label
  "    "
  *
  type$ "book" =
#else !LAB_ALPH
{ type$ "book" =
#endif LAB_ALPH
  type$ "inbook" =
  or
    'author.editor.sort
    { type$ "proceedings" =
        'editor.organization.sort
        { type$ "manual" =
            'author.organization.sort
            'author.sort
          if$
        }
      if$
    }
  if$
#if LAB_ALPH
  *
#endif LAB_ALPH
  "    "
  *
  year field.or.null sortify
  *
  "    "
  *
  title field.or.null
  sort.format.title
  *
  #1 entry.max$ substring$
  'sort.key$ :=
}
```

```
ITERATE {presort}
```

11 SORTing

And now we can sort

```
SORT
```

```
#endif SORTED
```

12 Final Calculations for Labels

12.1 For Alphabetic Labels

12.1.1 sorted

This long comment applies only to alphabetic labels, when sorted

Now comes the final computation for alphabetic labels, putting in the ‘a’ s and ‘b’ s and so forth if required. This involves two passes: a forward pass to put in the ‘b’ s, ‘c’ s and so on, and a backwards pass to put in the ‘a’ s (we don’t want to put in ‘a’ s unless we know there are ‘b’ s).

We have to keep track of the longest (in `width$` terms) label, for use by the “`thebibliography`” environment.

```
#if LAB_ALPH

#if SORTED
```

```
VAR: longest.label, last.sort.label, next.extra: string
     longest.label.width, last.extra.num: integer
```

```
STRINGS { longest.label last.sort.label next.extra }

INTEGERS { longest.label.width last.extra.num }
```

initialize.longest.label

```
initialize.longest.label ==
 BEGIN
      longest.label := ""
      last.sort.label := int.to.chr$(0)
      next.extra := ""
      longest.label.width := 0
      last.extra.num := 0
 END
```

```
FUNCTION {initialize.longest.label}
{ "" 'longest.label :=
  #0 int.to.chr$ 'last.sort.label :=
  "" 'next.extra :=
  #0 'longest.label.width :=
  #0 'last.extra.num :=
}
```

forward.pass

```
forward.pass ==
 BEGIN
      if last.sort.label = sort.label then
          last.extra.num := last.extra.num + 1
          extra.label := int.to.chr$(last.extra.num)
      else
          last.extra.num := chr.to.int$("a")
          extra.label := ""
          last.sort.label := sort.label
      fi
 END
```

```
FUNCTION {forward.pass}
{ last.sort.label sort.label =
    { last.extra.num #1 + 'last.extra.num :=
      last.extra.num int.to.chr$ 'extra.label :=
    }
```

```
    { "a" chr.to.int$ 'last.extra.num :=
      "" 'extra.label :=
      sort.label 'last.sort.label :=
    }
  if$
}
```

reverse.pass

```
reverse.pass ==
 BEGIN
      if next.extra = "b" then
          extra.label := "a"
      fi
      label := label * extra.label
      if width$(label) > longest.label.width then
          longest.label := label
          longest.label.width := width$(label)
      fi
      next.extra := extra.label
 END
```

```
FUNCTION {reverse.pass}
{ next.extra "b" =
    { "a" 'extra.label := }
    'skip$
  if$
  label extra.label * 'label :=
  label width$ longest.label.width >
    { label 'longest.label :=
      label width$ 'longest.label.width :=
    }
    'skip$
  if$
  extra.label 'next.extra :=
}
```

```
EXECUTE {initialize.longest.label}

ITERATE {forward.pass}

REVERSE {reverse.pass}
```

12.1.2 not sorted

It still doesn't seem like a good idea to use an order-of-citation reference list when using alphabetic labels, but when this happens we must compute the longest label

```
#else !SORTED

STRINGS { longest.label }

INTEGERS { longest.label.width }
```

initialize.longest.label

```
FUNCTION {initialize.longest.label}
{ "" 'longest.label :=
  #0 'longest.label.width :=
}
```

longest.label.pass

```
FUNCTION {longest.label.pass}
{ label width$ longest.label.width >
    { label 'longest.label :=
      label width$ 'longest.label.width :=
    }
    'skip$
```

```
  if$
}
```

```
EXECUTE {initialize.longest.label}

ITERATE {longest.label.pass}

#endif SORTED
```

12.2 For Numeric Labels

Now comes the computation for numeric labels.

We use either the sorted order or original order.

We still have to keep track of the longest (in `width$` terms) label, for use by the "`thebibliography`" environment.

```
#else !LAB_ALPH

STRINGS { longest.label }

INTEGERS { number.label longest.label.width }
```

initialize.longest.label

```
FUNCTION {initialize.longest.label}
{ "" 'longest.label :=
  #1 'number.label :=
  #0 'longest.label.width :=
}
```

longest.label.pass

```
FUNCTION {longest.label.pass}
{ number.label int.to.str$ 'label :=
  number.label #1 + 'number.label :=
  label width$ longest.label.width >
    { label 'longest.label :=
      label width$ 'longest.label.width :=
    }
    'skip$
  if$
}
```

```
EXECUTE {initialize.longest.label}

ITERATE {longest.label.pass}

#endif LAB_ALPH
```

13 Writing onto the `.bbl` File

Now we're ready to start writing the `.BBL` file.

We begin, if necessary, with a LaTeX macro for unnamed names in an alphabetic label; next comes stuff from the 'preamble' command in the database files. Then we give an incantation containing the command

```
\begin{thebibliography}{...}
```

where the '...' is the longest label.

begin.bib

```
FUNCTION {begin.bib}
#if LAB_ALPH
{ et.al.char.used
    { "\newcommand{\etalchar}[1]{$^{#1}$}" write$ newline$ }
    'skip$
```

```
  if$
  preamble$ empty$
#else !LAB_ALPH
{ preamble$ empty$
#endif LAB_ALPH
    'skip$
    { preamble$ write$ newline$ }
  if$
  "\begin{thebibliography}{"  longest.label  * "}" * write$ newline$
}

EXECUTE {begin.bib}
```

We also call `init.state.consts`, for use by the output routines.

```
EXECUTE {init.state.consts}
```

Now we produce the output for all the entries

```
ITERATE {call.type$}
```

Finally, we finish up by writing the '`\end{thebibliography}`' command.

end.bib

```
FUNCTION {end.bib}
{ newline$
  "\end{thebibliography}" write$ newline$
}

EXECUTE {end.bib}
```

終

Rex quadrupedum & immanissima omnium
Johann Amos Comenius, *Orbis Pictus*

Merry TEXmas
&
Happy
TEX
ing
!