5 Bibliography-style hacking

A printer friendly PDF version of this page is available bthack.pdf (84Kb)

This document starts (and ends) with Section 5, because in reality it is the final section of "BibTeXing'' [4], the general documentation for . But that document was meant for all users, while this one is just for style designers, so the two are physically separate. Still, you should be completely familiar with "BibTeXing'', and all references in this document to sections and section numbers assume that the two documents are one.

This section, along with the standard-style documentation file `btxbst.doc`, should explain how to modify existing style files and to produce new ones. If you're a serious style hacker you should be familiar with van Leunen [7] for points of style, with Lamport [3] and Knuth [2] for formatting matters, and perhaps with *Scribe* [6] for compatibility details. And while you're at it, if you don't read the great little book by Strunk and White [5], you should at least look at its entries in the database and the reference list to see how handles multiple names.

To create a new style, it's best to start with an existing style that's close to yours, and then modify that. This is true even if you're simply updating an old style for version 0.99 (I've updated four nonstandard styles, so I say this with some experience). If you want to insert into a new style some function you'd written for an old (version 0.98i) style, keep in mind that the order of the arguments to the assignment (`:=`) function has been reversed. When you're finished with your style, you may want to try running it on the entire `XAMPL.BIB` database to make sure it handles all the standard entry types.

If you find any bugs in the standard styles, or if there are things you'd like to do with bibliography-style files but can't, please complain to Oren Patashnik.

5.1 General description

You write bibliography styles in a postfix stack language. It's not too hard to figure out how by looking at the standard-style documentation, but this description fills in a few details (it will fill in more details if there's a demand for it).

Basically the style file is a program, written in an unnamed language, that tells how to format the entries that will go in the reference list (henceforth ''the entries'' will be ''the entry list'' or simply ''the list'', context permitting). This programming language has ten commands, described in the next subsection. These commands manipulate the language's objects: constants, variables, functions, the stack, and the entry list. (Warning: The terminology in this documentation, chosen for ease of explanation, is slightly different from 's. For example, this documentation's ''variables'' and ''functions'' are both ''functions'' to . Keep this in mind when interpreting 's error messages.)

There are two types of functions: *built-in* ones that provides (these are described in Section 5.3), and ones you define using either the `MACRO` or `FUNCTION` command.

Your most time-consuming task, as a style designer, will be creating or modifying functions using the `FUNCTION` command (actually, becoming familiar with the references listed above will be more time consuming, but assume for the moment that that's done).

Let's look at a sample function fragment. Suppose you have a string variable named `label` and an integer variable named `lab.width`, and suppose you want to append the character 'a' to `label` and to increment `lab.width`:

```
. . .
label "a" * 'label :=          % label := label * "a"
lab.width #1 + 'lab.width :=   % lab.width := lab.width + 1
. . .
```

In the first line, `label` pushes that variable's value onto the stack. Next, the `"a"` pushes the string constant 'a' onto the stack. Then the built-in function `*` pops the top two strings and pushes their concatenation. The `'label` pushes that variable's name onto the stack. And finally, the built-in function `:=` pops the variable name and the concatenation and performs the assignment treats the stuff following the `%` as a comment in the style file. The second line is similar except that it uses `#1`, with no spaces intervening between the '`#`' and the '`1`', to push this integer constant.

The nonnull spacing here is arbitrary: multiple spaces, tabs, or newlines are equivalent to a single one (except that you're probably better off not having blank lines within commands, as explained shortly).

For string constants, absolutely any printing character is legal between two consecutive double quotes, but here (and only here) treats upper- and lower-case equivalents as different. Furthermore, spacing *is* relevant within a string constant, and you mustn't split a string constant across lines (that is, the beginning and ending double quotes must be on the same line).

Variable and function names may not begin with a numeral and may not contain any of the ten restricted characters on page 143 of the LaTeX book, but may otherwise contain any printing characters. Also, considers upper- and lower-case equivalents to be the same.

Integers and strings are the only value types for constants and variables (booleans are implemented simply as 0-or-1 integers). There are three kinds of variables:

global variables
: These are either integer- or string-valued, declared using an `INTEGERS` or `STRINGS` command.

entry variables
: These are either integer- or string-valued, declared using the `ENTRY` command. Each has a value for each entry on the list (example: a variable `label` might store the label string you'll use for the entry).

fields
: These are string-valued, read-only variables that store the information from the database file; their values are set by the `READ` command. As with entry variables, each has a value for each entry.

5.2 Commands

There are ten style-file commands: Five (`ENTRY`, `FUNCTION`, `INTEGERS`, `MACRO`, and `STRINGS`) declare and define variables and functions; one (`READ`) reads in the database information; and four (`EXECUTE`, `ITERATE`, `REVERSE`, and `SORT`) manipulate the entries and produce output. Although the command names appear here in upper case, ignores case differences.

Some restrictions: There must be exactly one ENTRY and one READ command; the ENTRY command, all MACRO commands, and certain FUNCTION commands (see next subsection's description of call.type$) must precede the READ command; and the READ command must precede the four that manipulate the entries and produce output.

Also it's best (but not essential) to leave at least one blank line between commands and to leave no blank lines within a command; this helps recover from any syntax errors you make.

You must enclose each argument of every command in braces. Look at the standard-style documentation for syntactic issues not described in this section. Here are the ten commands:

ENTRY

Declares the fields and entry variables. It has three arguments, each a (possibly empty) list of variable names. The three lists are of: fields, integer entry variables, and string entry variables. There is an additional field that

automatically declares, crossref, used for cross referencing. And there is an additional string entry variable automatically declared, sort.key$, used by the SORT command. Each of these variables has a value for each entry on the list.

EXECUTE

Executes a single function. It has one argument, the function name.

FUNCTION

Defines a new function. It has two arguments; the first is the function's name and the second is its definition. You must define a function before using it; recursive functions are thus illegal.

INTEGERS

Declares global integer variables. It has one argument, a list of variable names. There are two such automatically-declared variables, entry.max$ and global.max$, used for limiting the lengths of string variables. You may have any number of these commands, but a variable's declaration must precede its use.

ITERATE

Executes a single function, once for each entry in the list, in the list's current order (initially the list is in citation order, but the SORT command may change this). It has one argument, the function name.

MACRO

Defines a string macro. It has two arguments; the first is the macro's name, which is treated like any other variable or function name, and the second is its definition, which must be double-quote-delimited. You must have one for each three-letter month abbreviation; in addition, you should have one for common journal names. The user's database may override any definition you define using this command. If you want to define a string the user can't touch, use the FUNCTION command, which has a compatible syntax.

READ

Dredges up from the database file the field values for each entry in the list. It has no arguments. If a database entry doesn't have a value for a field (and probably no database entry will have a value for every field), that field variable is marked as missing for the entry.

REVERSE

Exactly the same as the ITERATE command except that it executes the function on the entry list in reverse order.

SORT
Sorts the entry list using the values of the string entry variable `sort.key$`. It has no arguments.

STRINGS

Declares global string variables. It has one argument, a list of variable names. You may have any number of these commands, but a variable's declaration must precede its use.

5.3 The built-in functions

Before we get to the built-in functions, a few words about some other built-in objects. There is one built-in string entry variable, `sort.key$`, which the style program must set if the style is to do sorting. There is one built-in field, `crossref`, used for the cross referencing feature described in Section 4. And there are two built-in integer global variables, `entry.max$` and `global.max$`, which are set by default to some internal constants; you should truncate strings to these lengths before you assign to string variables, so as to not generate any warning messages.

There are currently 37 built-in functions. Every built-in function with a letter in its name ends with a '`$`'. In what follows, ''first'', ''second'', and so on refer to the order popped. A ''literal'' is an element on the stack, and it will be either an integer value, a string value, a variable or function name, or a special value denoting a missing field. If any popped literal has an incorrect type,

complains and pushes the integer 0 or the null string, depending on whether the function was supposed to push an integer or string.

>
Pops the top two (integer) literals, compares them, and pushes the integer 1 if the second is greater than the first, 0 otherwise.

<
Analogous.

=
Pops the top two (both integer or both string) literals, compares them, and pushes the integer 1 if they're equal, 0 otherwise.

+
Pops the top two (integer) literals and pushes their sum.

-
Pops the top two (integer) literals and pushes their difference (the first subtracted from the second).

*****
Pops the top two (string) literals, concatenates them (in reverse order, that is, the order in which pushed), and pushes the resulting string.

:=
Pops the top two literals and assigns to the first (which must be a global or entry variable) the value of the second.

add.period$
Pops the top (string) literal, adds a '`.`' to it if the last non'`}`' character isn't a '`.`', '`?`', or '`!`', and pushes this resulting string.

call.type$
Executes the function whose name is the entry type of an entry. For example if an entry is of type `book`, this function executes the `book` function. When given as an argument to the `ITERATE` command, `call.type$` actually produces the output for the entries. For an entry with an unknown type, it executes the function `default.type`. Thus you should define (before the

READ command) one function for each standard entry type as well as a default.type function.

change.case$
Pops the top two (string) literals; it changes the case of the second according to the specifications of the first, as follows. (Note: The word 'letters' in the next sentence refers only to those at brace-level 0, the top-most brace level; no other characters are changed, except perhaps for ''special characters'', described in Section 4.) If the first literal is the string 't', it converts to lower case all letters except the very first character in the string, which it leaves alone, and except the first character following any colon and then nonnull white space, which it also leaves alone; if it's the string 'l', it converts all letters to lower case; and if it's the string 'u', it converts all letters to upper case. It then pushes this resulting string. If either type is incorrect, it complains and pushes the null string; however, if both types are correct but the specification string (i.e., the first string) isn't one of the legal ones, it merely pushes the second back onto the stack, after complaining. (Another note: It ignores case differences in the specification string; for example, the strings t and T are equivalent for the purposes of this built-in function.)

chr.to.int$
Pops the top (string) literal, makes sure it's a single character, converts it to the corresponding ASCII integer, and pushes this integer.

cite$
Pushes the string that was the \cite-command argument for this entry.

duplicate$
Pops the top literal from the stack and pushes two copies of it.

empty$
Pops the top literal and pushes the integer 1 if it's a missing field or a string having no non-white-space characters, 0 otherwise.

format.name$
Pops the top three literals (they are a string, an integer, and a string literal). The last string literal represents a name list (each name corresponding to a person), the integer literal specifies which name to pick from this list, and the first string literal specifies how to format this name, as explained in the next subsection. Finally, this function pushes the formatted name.

if$
Pops the top three literals (they are two function literals and an integer literal, in that order); if the integer is greater than 0, it executes the second literal, else it executes the first.

int.to.chr$
Pops the top (integer) literal, interpreted as the ASCII integer value of a single character, converts it to the corresponding single-character string, and pushes this string.

int.to.str$
Pops the top (integer) literal, converts it to its (unique) string equivalent, and pushes this string.

missing$
Pops the top literal and pushes the integer 1 if it's a missing field, 0 otherwise.

newline$
Writes onto the bbl file what's accumulated in the output buffer. It writes a blank line if and only if the output buffer is empty. Since write$ does reasonable line breaking, you should use this function only when you want a blank line or an explicit line break.

num.names$
Pops the top (string) literal and pushes the number of names the string represents--one plus the number of occurrences of the substring ''and'' (ignoring case differences) surrounded by nonnull white-space at the top brace level.

pop$
Pops the top of the stack but doesn't print it; this gets rid of an unwanted stack literal.

preamble$
Pushes onto the stack the concatenation of all the `@PREAMBLE` strings read from the database files.

purify$
Pops the top (string) literal, removes nonalphanumeric characters except for white-space characters and hyphens and ties (these all get converted to a space), removes certain alphabetic characters contained in the control sequences associated with a ''special character'', and pushes the resulting string.

quote$
Pushes the string consisting of the double-quote character.

skip$
Is a no-op.

stack$
Pops and prints the whole stack; it's meant to be used for style designers while debugging.

substring$
Pops the top three literals (they are the two integers literals *len* and *start*, and a string literal, in that order). It pushes the substring of the (at most) *len* consecutive characters starting at the *start*th character (assuming 1-based indexing) if *start* is positive, and ending at the *start*th character from the end if *start* is negative (where the first character from the end is the last character).

swap$
Swaps the top two literals on the stack.

text.length$
Pops the top (string) literal, and pushes the number of text characters it contains, where an accented character (more precisely, a ''special character'', defined in Section 4) counts as a single text character, even if it's missing its matching right brace, and where braces don't count as text characters.

text.prefix$
Pops the top two literals (the integer literal *len* and a string literal, in that order). It pushes the substring of the (at most) *len* consecutive text characters starting from the beginning of the string. This function is similar to `substring$`, but this one considers a ''special character'', even if it's missing its matching right brace, to be a single text character (rather than however many ASCII characters it actually comprises), and this function doesn't consider braces to be text characters; furthermore, this function appends any needed matching right braces.

top$
Pops and prints the top of the stack on the terminal and log file. It's useful for debugging.

type$
Pushes the current entry's type (book, article, etc.), but pushes the null string if the type is either unknown or undefined.

warning$
Pops the top (string) literal and prints it following a warning message. This also increments a count of the number of warning messages issued.

while$
Pops the top two (function) literals, and keeps executing the second as long as the (integer) literal left on the stack by executing the first is greater than 0.

width$
Pops the top (string) literal and pushes the integer that represents its width in some relative units (currently, hundredths of a point, as specified by the June 1987 version of the font; the only

white-space character with nonzero width is the space). This function takes the literal literally; that is, it assumes each character in the string is to be printed as is, regardless of whether the character has a special meaning to TEX, except that ‘‘special characters’’ (even without their right braces) are handled specially. This is meant to be used for comparing widths of label strings.

write$

Pops the top (string) literal and writes it on the output buffer (which will result in stuff being written onto the `bbl` file when the buffer fills up).

Note that the built-in functions `while$` and `if$` require two function literals on the stack. You get them there either by immediately preceding the name of a function by a single quote, or, if you don't feel like defining a new function with the `FUNCTION` command, by simply giving its definition (that is, giving what would be the second argument to the `FUNCTION` command, including the surrounding braces). For example the following function fragment appends the character ‘`a`’ if the string variable named `label` is nonnull:

```
. . .
label "" =
  'skip$
  { label "a" * 'label := }
if$
. . .
```

A function whose name you quote needn't be built in like `skip$` above--it may, for example, be a field name or a function you've defined earlier.

5.4 Name formatting

What's in a name? Section 4 pretty much describes this. Each name consists of four parts: First, von, Last, and Jr; each consists of a list of name-tokens, and any list but Last's may be empty for a nonnull name. This subsection describes the format string you must supply to the built-in function `format.name$`.

Let's look at an example of a very long name. Suppose a database entry [1] has the field

```
author = "Charles Louis Xavier Joseph de la Vall{\'e}e Poussin"
```

and suppose you want this formatted ‘‘last name comma initials’’. If you use the format string

```
"{vv~}{ll}{, jj}{, f}?"
```

will produce

```
de~la Vall{\'e}e~Poussin, C.~L. X.~J?
```

as the formatted string.

Let's look at this example in detail. There are four brace-level 1 *pieces* to this format string, one for each part of a name. If the corresponding part of a name isn't present (the Jr part for this name), everything in that piece is ignored. Anything at brace-level 0 is output verbatim (the presumed typo ‘`?`’ for this name is at brace-level 0), but you probably won't use this feature much.

Within each piece a double letter tells to use whole tokens, and a single letter, to abbreviate them (these letters must be at brace-level 1); everything else within the piece is used verbatim (well, almost everything--read on). The tie at the end of the von part (in `{vv~}`) is a discretionary tie-- will output a tie at that point if it thinks there's a need for one; otherwise it will output a space. If you really, really, want a tie there, regardless of what thinks, use two of them (only one will be output); that is, use `{vv~~}`. A tie is discretionary only if it's the last character of the piece; anywhere else it's treated as an ordinary character.

puts default strings *between* tokens of a name part: For whole tokens it uses either a space or a tie, depending on which one it thinks is best, and for abbreviated tokens it uses a period followed by either a space or a tie. However it doesn't use this default string after the last token in a list; hence there's no period following the 'J' for our example. You should have used

```
"{vv~}{ll}{, jj}{, f.}" to get to produce the same formatted string but with the question mark replaced by a period. Note that the period should go inside the First-name piece, rather than where the question mark was, in case a name has no First part.
```

If you want to override 's default between-token strings, you must explicitly specify a string. For example suppose you want a label to contain the first letter from each token in the von and Last parts, with no spaces; you should use the format string

```
"{v{}}{l{}}"
```

so that will produce '`dlVP`' as the formatted string. You must give a string for each piece whose default you want overridden (the example here uses the null string for both pieces), and this string must immediately follow either the single or double letter for the piece. You may not have any other letters at brace-level 1 in the format string.

Bibliography

1

Charles Louis Xavier Joseph de la Vallée Poussin.
A strong form of the prime number theorem, 19th century.

2

Donald E. Knuth.
The TEXbook.
Addison-Wesley, 1984.

3

Leslie Lamport.
LATEX: *A Document Preparation System*.
Addison-Wesley, 1986.

4

Oren Patashnik.
ing.
Documentation for general users, 8 February 1988.

5

William Strunk, Jr. and E. B. White.
The Elements of Style.
Macmillan, third edition, 1979.

6

Unilogic, Ltd., Pittsburgh.
Scribe Document Production System User Manual, April 1984.
Chapter twelve and appendices E8 through E10 deal with bibliographies.

Mary-Claire van Leunen.
A Handbook for Scholars.
Knopf, 1979.

About this document ...

This document was generated using the **LaTeX**2HTML translator Version 2002-1 (1.68)

The command line arguments were:
latex2html `-no_subdir -split 0 -show_section_numbers /tmp/lyx_tmpdir16750g899CL/lyx_tmpbuf1/bthack.tex`

The translation was initiated by root on 2002-12-22
