# labtx

Lua による BibTeX の実装です．

## 1　インストール

TeX Live または W32TeX とともに使うことを想定しています．拡張子が lua であるファイルを全て kpathsea が探せる場所においてください．例えば `$TEXMF/srcpts/labtx` 以下にすべておきます．その後

- UNIX: labtx.lua へのリンクを適当な bin ディレクトリに作る．
- Windows (TeX Live): bin/win32/runscript.exe を bin/win32/labtx.exe としてコピーする．
- W32TeX: bin/win32/runscr.exe を bin/win32/labtx.exe としてコピーする．

とします．

## 2　使い方

```
$ bibtex sample
```

としていた代わりに

```
$ labtx sample
```

とします．つまり，sample.tex を処理するには

```
$ latex sample.tex
$ labtx sample
$ latex sample.tex
$ latex sample.tex
```

とします．文字コードは（現在のところ）UTF-8 に限定されています．

## 3　データベースについて

スタイルファイルの作り方の前に，言葉の定義を兼ねて読み込むデータベースについて書いておきます．データベースは典型的には次のようになっています．

```
@article{reference,
```

```
   author = "Last, First",
   title = {Some title},
}
```

- 各々のデータを「エントリー」
- article を「エントリータイプ」
- reference を「エントリーキー」
- 著者名などの情報を「フィールド」
- author = "Last, First"における author を「キー」"Last, First"を「値」

と呼ぶことにします．エントリータイプ，エントリーキー，またフィールドのキーは大文字小文字を無視して処理されます．

また次のようなデータ

```
@string{str = "some string"}
@article{reference
   title = "Title and " # str
}
```

に対しては，文字列の連結と置換が行われます．たとえばこの例では reference 内の title に対する値が “Title and some string” と置き換えられます．このような置き換えのルール（今の場合は str を “some string” に置き換える）をマクロと呼ぶことにします．

# 4 スタイルファイルの作り方

スタイルファイル名 <style> の実体は，labtx-<style>_bst.lua という Lua スクリプトファイルです．次の形をとります．

```
local style = BibTeX:get_default_style() -- スタイルのひな形の取得
-- <style に設定を入れていく>
return style -- 設定したスタイルを返す
```

実際に style に設定を入れていく部分は次の流れになります．

(1) style.templates と style.formatters にエントリータイプごとの出力テンプレートを設定する．
(2) style.crossref にクロスリファレンスの設定を行う．
(3) style.sorting にソートの設定をする．
(4) style.label にラベル出力の設定をする．

順番に見ていきます．

## 4.1 テンプレート設定

style.templates と style.formatters を通じて設定を行います．以下の例を見てみます．

```
local Functions = require "labtx-funcs" -- 便利関数ロード

style.templates["article"]
  = "[$<author>:<\\emph{|$<title>|}>:$<journal>:$<year>]"
function style.formatters:title(c)
  if c.fields["title"] == nil then return nil end
  return c.fields["title"]:upper() -- 大文字にする
end
function style.formatters:author(c)
  if c.fields["author"] == nil then return nil end
  local a = Functions.split_names(c.fields["author"])
  if #a <= 2 then
    return Functions.make_name_list(a,
      "{ff~}{vv~}{ll}{,␣jj}",{",␣","␣and␣"},",␣et~al.")
  else
    return Functions.make_name_list(a,
      "{ff~}{vv~}{ll}{,␣jj}",{",␣",",␣and␣"},",␣et~al.")
  end
end
```

lbtx-funcs については 6 節をご覧ください．style.templates["article"] にエントリータイプ article の出力設定を入れます．上の例の通り，特殊な書式を持った文字列で指定します．

- “[A:B:C...:X]” は「ブロック」を表します．各ブロックには「セパレータ」“<sep>” と「終端文字列」“<last>” が設定されており，“A<sep>B<sep>C...<sep>X<last>” というように出力されます．ただし，たとえば “B” が空文字列の場合は，“A<sep>C...<sep>X<last>” というように出力されます．なお，このセパレータや終端文字列では，“.” が連続しないように処理がされます．ブロックはネストが可能です．セパレータと終端文字列は style.blockseparator で指定できます（後述）．デフォルトはセパレータが “,”，終端文字列が “.” です．
- “$<A>” はフィールド A の出力を行います．A がフィールドにない場合は空文字列になります．また “$<A|B|...|X>” と続けることもできて，この場合は A,B,...,X の中で最初に定義されているものが出力されます．
- “<A|B|C>” は，B が空文字列ならば空文字列に，そうでないならば “ABC” という文字列になります．ネストが可能です．
- 特殊文字は “%” でエスケープできます．

ブロックのセパレータと終端文字列は style.blockseparator で設定します．中身は配列で，

```
style.blockseparator = {
    {<ネストレベル 1 のセパレータ>,<ネストレベル 1 の終端文字列>},
    {<ネストレベル 2 のセパレータ>,<ネストレベル 2 の終端文字列>},
    ...
}
```

という形です．

“$<A>” で出力されるフィールド A の出力を，style.formatters により整形することができます．その実体は関数で，“$<A>” の整形を行う関数は

```
function style.formatters:A(c)
-- 本体
end
```

という形で定義します．戻り値は文字列です．引数 c はテーブルで，

- c.key にはエントリーキー
- c.type にはエントリータイプ
- c.fields[name] にはキーが name のフィールドの中身

が入っています．より詳しくは節 7 を参照してください．上の author の例ではモジュール labtx-funcs の提供する関数を使っています．節 6 を参照してください．

style.formatters の名前は実際のフィールド名である必要はありません．たとえば

```
style.templates["article"] = "$<author_editor>:$<title>"
function style.formatters:author_editor(c)
    if c.fields["author"] == nil then return c.fields["editor"]
    else return c.fields["author"]
end
```

とすると，“$<author_editor>” は「author が定義されていれば author フィールドに，そうでなければ editor フィールド」という扱いになります．（つまり “$<author|editor>” と同等．）

少し発展的な内容です．

- ブロックの定義において，“[A:@S<sep>B:C]” とすると，B の前のセパレータを “sep” に変更できます．
- “$<A|(B)|C|...|X>” とすると，B はフィールド名ではなく，テンプレートして解釈されます．たとえば，“$<author|(<edited by |\$<editor>|.)>” とすると，
  - author が定義されていれば author フィールドそのまま．
  - author が定義されていなく，editor が定義されていれば “edited by <editor フィールド>.”
  - author も editor も定義されていなければ空文字列

  が出力されます．
- formatters にも templates のような書式が使えます．たとえば上の style.formatters:author_editor の例は

  ```
  style.formatters.author_editor = "$<author|editor>"
  ```

  と書くこともできます．なお，ここでの “$<A>” によるフィールド名の参照は，必ずフィールドの内容そのままとして解釈され，formatters による整形は行われません．
- formatters の関数の戻り値は原則文字列ですが，文字列の配列を返すこともできます．これは

ブロックとして扱われます．たとえば

```
style.templates["article"] = "[$<author>:$<title_journal_year>]"
function style.formatters:title_journal_year(c)
    return {c.fields["title"],c.fields["journal"],c.fields["year"]}
end
```

と

```
style.templates["article"] = "[$<author>:$<title>:$<journal>:$<year>]"
```

は等価です．

## 4.2 クロスリファレンス

クロスリファレンスの設定は style.corssref に対して行います．例としては次のようになります．

```
style.crossref.templates["article"]
  = "[$<author>:$<title>:\\cite{$<crossref>}]"
```

これにより，corssref フィールドが定義されている article に対しては，その出力が上で指定されたものに変わります．なお，style.formatters や style.blockseparator はそのまま使われます．また，style.crossref.templates["article"] が定義されていない場合は style.formatters["article"] が使われます．

### 4.2.1 クロスリファレンスの遺伝

クロスリファレンスが行われると，親エントリーから子エントリーへとフィルードのコピーが行われます．デフォルトでは，そのままのコピーが行われますが，この挙動は制御することができます．たとえば

```
style.crossref.inherit["article"]["book"] = {
    {"title","booktitle"},
    {{"author","editor"},"editor"},
    {{"A","B"},{"C","D"}}
}
```

とすると，親：article，子：book というクロスリファレンスに対して

- title は booktitle にコピー
- author と editor は editor にコピー
- A,B は C,D の両方にコピー

が行われます．各々の項目に空文字列 “” を指定すると，それは「全部」を表します．たとえば

```
style.crossref.inherit[""][""] = {
    {"title","booktitle"}
}
```

は全てのエントリータイプに対して，title を booktitle へとコピーします．個別の指定は，“” による全てへの指定より優先されます．たとえば

```
style.crossref.inherit[""][""] = {
    {"title","booktitle"}
}
style.crossref.inherit["article"][""] = {
    {"title","subtitle"}
}
```

という指定は，article からの場合に限り title を subtitile に，それ以外は title を booktitle にコピーします．

### 4.2.2 その他の設定

子エントリーに既にフィールドが存在している場合に上書きするかどうかは，style.crossref.override で制御します．簡単な方法は

```
style.crossref.override = true
```

とすることです．これで全てのフィールドが上書きされます．（なお，デフォルトは false です．）inherit と同様個別の定義を行うこともできます．たとえば

```
style.crossref.override["article"]["book"] = {
    {{"author","editor"},{"bookeditor"},true}
}
```

は親：article の author か editor フィールドが子：book の bookeditor フィールドにコピーされる場合に上書きを許すことを意味します．inherit と同様 “” は全ての項目を表します．

その他以下の項目が設定できます．

- style.crossref.mincrossrefs：ここに設定されているだけのクロスリファレンスがあれば，エントリーが現在の参考文献一覧に追加されます．デフォルト 2．
- style.crossref.reference_key_name：クロスリファレンスを表すフィールドのキー名です．デフォルト “crossref”．

## 4.3 ソート

ソートに関する設定は，style.sorting で行います．

```
style.sorting.targets = {"name","title","year"}
```

とすると，「名前」「タイトル」「年」の順番で比較されます．実際に比較する値は，style.sorting.formatters で設定可能です．

```
function style.sorting.formatters:name(c)
....
end
```

とすると，上の name に対応する定義を上書きすることができます．

比較するための関数は，

- 一致しているか否かを返す style.sorting.equal
- < であるかを返す style.sorting.lessthan

で設定できます．どちらも二つの文字列をとり，bool 値を返す関数です．

### 4.4 ラベル

thebibliography 環境における

```
\bibitem[label]{key} ....
```

の label の部分をラベルと呼ぶことにします．デフォルトでは，著者などから自動的に生成されます．ただし，shorthand フィールドがある場合には，その値が使われます．ラベルの生成を押さえる（標準スタイルの「plain」に対応）には

```
style.label.make = false
```

とします．

より細かく設定する場合は，style.label.templates と style.label.formatters を設定します．設定の方法はテンプレート（項 4.1）と同様です．なお，同じラベル名が生成された場合，デフォルトでは末尾に a,b,c,... が追加されます．末尾につく文字列を変更するには，style.label.suffix を再定義します．

```
-- 同じラベルを持つもののうち n 番目のものの末尾文字列を返す
function style.label:suffix(n)
  -- 1,2,3 とつける
  return string.format("%s",n)
end
```

なお，

```
function style.label:make(c)
  ....
  return ...
end
```

と関数として定義すると，その戻り値そのものがラベルとして利用されます．

## 5 言語機能

言語ごとに出力を切り替えるための簡単な機構が用意されています．

### 5.1 言語ごとの設定

style.languages.<**言語**>以下に，これまでと同様の設定を入れます．例えば言語 “ja”（日本語を想定）に関しては，style.languages.ja 以下に以下のように設定します．

```
-- 日本語用設定
style.languages.ja = {} -- テーブルを作成しておく
```

```
style.languages.ja.templates = {}
style.languages.ja.templates["article"]
    = "[<著者：|$<author>|>:<\\emph{|$<title>|}>:$<journal>:<|$<year>|年>]"
function style.languages.ja.formatters:author(c)
  if c.fields["author"] == nil then return nil end
  local a = Functions.split_names(c.fields["author"])
  if #a <= 2 then
    return Functions.make_name_list(a,"{vv~}{ff~}",{",␣",",␣"},"他")
  else
    return Functions.make_name_list(a,"{vv~}{ff~}",{",␣",",␣"},"他")
  end
end
-- 設定されていない項目(article 以外のエントリータイプなど)に対しては
-- 言語によらない設定が適用される.

style.languages.ja.crossref = {}
-- クロスリファレンスの設定
style.languages.ja.label = {}
-- ラベルの設定
```

### 5.2 言語決定

言語決定は，langid フィールドにより決定されます．langid フィールドが ja ならば，言語 ja の設定が使われます．これを変更するには，style.languages.<言語>.is を定義します．例えば，

```
function style.languages.ja.is(c)
  if c.fields["language"] == "japanese" then
    return true
  else
    return false
  end
end
```

とすると languages フィールドが japanese のエントリーが日本語であると判定されます．ただし，この場合 languages が空であるが landid フィールドが ja であっても日本語とは判定されません．これを避けるには上の条件分岐を

```
c.fields["language"] == "japanese" or c.fields["langid"] == "ja"
```

とします．

## 6 関数

有用そうな関数群やオリジナルの BibTeX に存在していた関数が，モジュール labtx-funcs で定義されています．

```
local Functions = require "labtx-funcs"
x = Functions.text_prefix(str,num)
```

のように使ってください．

### 6.1 stable_sort(list,comp)

配列 list に対して，安定なソートを行います．comp は比較関数です．省略された場合は標準演算子 < が使われます．

### 6.2 text_prefix(str,num)

str の先頭 num バイトを返します．ただし，文字を途中で切ることはなく，またコントロールシークエンス等や引数はバイト数に加算されません．たとえば，

```
text_prefix("a あい",2)
text_prefix("あいう",5)
```

はそれぞれ “a **あ**”，“**あい**” を返します．[1)]

### 6.3 text_length(str)

str のバイト数を返しますが，コントロールシークエンス等や引数は加算されません．

### 6.4 string_split(str,func)

検索関数 func により str を分割して返します．戻り値は二つの配列で，一つ目の配列には分割された文字列，二つ目の配列には分割文字列が入ります．たとえば

```
string_split("aXbYc",function(s) return s:find("[XY]") end)
```

は

```
{"a","b","c"},{"X","Y"}
```

を返します．

### 6.5 change_case(str,format)

大文字小文字の変換を行います．ただし，中括弧の中は処理されません．format は “t”，“u”，“l” のどれかで，

- “u”，“l” はそれぞれ大文字，小文字への変換を表す．
- “t” は小文字への変換を行うが，一文字目及び “: *” で表される文字の次の文字は変換されない．

### 6.6 split_names(names[,seps])

複数名の名前からなる文字列から，各人の名前の入った配列を得ます．人と人との区切りを配列 seps で与えることができます．（配列中のいずれかにマッチした部分で区切られる．）seps のデフォルトは{"[aA][nN][dD]"}です．

1) 内部コードは UTF-8 なので，“**あ**” や “**い**” は 3byte です．この扱いはどうするか考え中……．

## 6.7 get_name_parts(names)

名前から first name，last name，von part，jr part の四つの部分を抽出します．戻り値は

```
{first = <first part>, last = <last part>, von = <von part>, jr = <jr part>}
```

で，各々の部分は

```
{parts = <array of name>, seps = <separator of names>}
```

です．例えば von-von Last Last, First, jr に対しては，次のように返ります．

```
{
  first = {parts = {"First"}, seps = {}},
  last = {parts = {"Last","Last"}, seps = {"␣"}},
  von = {parts = {"von", "von"}, seps = {"-"}},
  jr = {pars = {"jr"}, seps = {}}
}
```

この関数は，次のルールに基づき名前を分解します．

(1) “[ ,~\t%-]+” に該当するパターン[2)]で区切り，配列を生成する．

(2) 1 で区切られた際に用いられた区切り文字のうち，最初の一文字がカンマ「,」のものの数を数える．この数に基づき，次の三つのパターンのどれかと見なす．

(a) カンマがない：First von Last のパターン．頭から見て von と見なされるパターンの前までが First，後ろから見て von と見なされるパターンの後ろまでが Last．von がない場合は 1 で区切られたうちの最後の一つのみが Last．（ただし，区切り文字が"-"のものはまとめて考える．例えば “First Last Last” の Last は “Last” であるが，“First Last-Last” ならば “Last-Last” である．）

(b) カンマが一つ：von Last, First のパターン．von Last から Last を抜き出す処理は (a) と同じ．

(c) カンマが二つ：von Last, Jr, First のパターン．von Last から Last を抜き出す処理は (a) と同じ．

(3) 2 における「von と見なされるパターン」とは，（基本的には）[3)]最初に現れたアルファベットが小文字であるもののことである．

## 6.8 forat_name_by_parts(nameparts,format)

format にて指定された書式に基づき，名前の整形を行います．nameparts は get_name_parts で得られる戻り値と同じかたちで与えます．format で与える書式は次の形です．

```
<str1>{<before1><name1><after1>}<str2>{<before2><name2><after2>}...
```

- <str1>はそのまま出力される．

2) Lua の意味でのパターン

3) 実際には中括弧内や，コントロールシークエンスで定義されたアクセントなども考慮に入れる．

- <name1>は “l”，“ll”，“f”，“ff”，“v”，“vv”，“j”，“jj” の何れか．Last name，First name，von part，jr part に対応し，二つ続いているものは名前全体を，そうでないものは短縮形を出力する．
- <before1>はそのまま出力される．ただし<name1>に対応する部分がない場合，出力されない．
- <after1>は{<sep1>}<after1_>か<after1_>（中括弧なし）の何れかである．<sep1>は<name1>の各部分をつなぐ文字として使われ，<after1_>は次の部分とのつなぐ文字として使われる．<sep1>が省略された場合や，“~” であった場合は，空白（“ ” か “~”）が状況に応じて使われる．もし常に “~” を出力したい場合は，“~~” を指定する．
- <str2>等も同様．

### 6.9 format_name(name,format)

BibTeX の format.names$と似た関数です．中身は

```
return forat_name_by_parts(get_name_parts(name),format)
```

です．

### 6.10 make_name_list(namearray, format, separray[, etalstr])

複数人の名前の配列から文字列を作ります．separray の長さを k，namearray の長さを n とすると，

```
namearray[1]separray[1]namaearray[2]separray[2] ....
namearray[n - k + 1]separray[2] ...
namearray[n - 1]separray[k]namearray[n]
```

という文字列を生成します．（実際には改行無し．）ただし，namearray の各項は format に従い整形され（書式は format_name_by_parts と同様），またもし namearray[n] が “others” の場合は，namearray[n] は etalstr に置き換えられます．デフォルトでは etalstr は空文字列です．

### 6.11 remove_TeX_cs(s)

s から TeX のコントロールシークエンスを取り除いた文字列を得ます．

## 7 文献データ

文献データは以下のようなテーブルに格納されています．変数名を Citation とします．

**Citation.type**
　エントリータイプ

**Citation.key**
　エントリーキー

`Citation.fields`

フィールドが格納されているテーブル．マクロなどが施された結果が返る．

**function** `Citation:clone()`

自分の複製を作ります．

**function** `Citation:set_field(key,cite,key1)`

文献データ `cite` のキー “`key1`” のフィールドを “`key`” に設定します．

**function** `Citation:get_raw_field(key)`

キー “`key`” のフィールドの生の値（マクロなど適用前）を返します．

# 8 変数 BibTeX

変数 `BibTeX` には，現在の labtx の状態が格納されています．

## 8.1 各種状態

`BibTeX.style_name`

スタイル名．

`BibTeX.cites`

引用されている文献一覧からなる配列．各々の中身は節 7 の通り．

`BibTeX.db`

読み込まれたデータベースを表すテーブル．エントリーキー “`key`” には

```
BibTeX.db["key"]
```

でアクセスできる．各々の中身は節 7 の通り．

`BibTeX.aux`

`aux` ファイル名．

`BibTeX.aux_contents`

`aux` ファイル名の中身．`aux` の各行の

```
\somecs{arg1}[arg2](arg3)
```

という行から

```
{somecs = {
  {arg = "arg1", open = "{", close = "}"},
  {arg = "arg2", open = "[", close = "]"},
  {arg = "arg3", open = "(", close = ")"}
}}
```

というテーブルが生成されて，ここに格納されている．括弧は上記の “`{}`”, “`[]`”, “`()`” が認識され，対応がとれているものとして扱われる．

### 8.2 関数

**BibTeX:output(str)**

bbl への出力を行う．

**BibTeX:outputline(str)**

bbl への一行出力を行う．

**BibTeX:outputthebibliography(style)**

style に従い文献データを出力する．

**BibTeX:warning(str)**

文字列 str を警告として出力する．出力は標準出力および blg に対して行われる．

**BibTeX:error(str,exit_code)**

文字列 str をエラーとして出力し，終了コード exit_code でプログラムを終了する．出力は標準エラー出力および blg に対して行われる．

**BibTeX:log(str)**

blg に str を出力する．

**BibTeX:message(str)**

標準出力に str を出力する．

## 9 デバッグ

次のようにしておくと，デバッグに有用な情報がはき出されたりする……かもしれません．

```
local labtxdebug = require "labtx-debug"
labtxdebug.debugmode = true -- デバッグモード ON

-- 以下スタイルファイルの処理
```