Autotyping

Convenient features for text that you enterfrequently in Emacs

Daniel Pfeiffer
additions by Dave Love

Autotyping

何度も何度も同じ文字を入力するようなことがありませんか．何度も同じ文字を入力することは手紙を書くことからプログラムを作るときまで起こり得ます．あるプロジェクトに特有なコメントや flow control constructs，マジックナンバーなどを何度も入力することはあらゆる時に起こります．Emacs は略称展開 (see ⟨undefined⟩ [(emacs)Abbrevs], page ⟨undefined⟩) の他にも退屈な繰り返しをするためのさまざま機能があります．

一つの答えは skeleton を使うことです．これは，何を挿入して，何をするかを柔軟に定義できます．さまざまなプログラミング言語はすぐに使用できる skeleton があります．そして，自分の必要性や好みに応じて合わせたり，新しく定義したりできます．

他の機能はテンプレート機能です．これは新しいファイルにファイル名やモードに応じて適当なものを挿入します．挿入されるファイルか skelton あるいは呼び出される関数を持つことができるのです．それから，マジックナンバーを自動的に変更するようなスクリプトを保存時に実行することもできます．あるいは，著作権表示を保存するたびに更新することもできるのです．これは，ファイルの更新時間の機能と似ています．

カーソル付近の単語に応じて URL を挿入できます．挿入される URL は柔軟に定義できます．meta-expansion の機能は代替補完とカーソル付近の文字に応じた展開との組を試すのに利用できます．

1 Skeleton を使う *(2004/04/02)*

Emacs で一連の文字やプログラミング言語のコンストラクタを入力したい時、skeleton を活用できます。普通それぞれの skeleton には専用のコマンドがあります。そして、それらのコマンドは普通の方法 (`M-x`やキー割り当てなど) で呼ぶことができます (see ⟨undefined⟩ [(emacs)Commands], page ⟨undefined⟩)。さまざまな skeleton を利用できるモードでは `C-c`にキーが割当てられていることもあります。モードによっては *Insert* メニューや定義済みの abbrev もあるかもしれません (see Chapter 3 [Skeletons as Abbrevs], page 4)。

skeleton で最も単純なものはメジャーモードに応じたテキストを挿入し、編集する可能性のありそうな場所にカーソルを移動させることです。テンプレートによっては、ユーザに入力を求め、挿入されたテキストの中に埋めていくこともあります。

skeleton は何度も入力する文字を尋ねるかもしれません。skeleton はループ機能さえ持っており、ユーザが入力を終えたいと思うまで何度でも尋ねさせることができるのです。何度も入力させるような例として「else if」があります。ループかどうかを確認するには `RET`や `C-g`、`C-h` を利用します。単に `RET` として空文字を入力すると、そこでループは終了します。一方、`C-g`で終了させると、ループだけでなくループ以降も実行されずに終了します。したがって、「else if」の例であれば「else」を閉じる処理は実行されません。ただし、中止させても構文上必要となるテキストは挿入されます。

2 すでに存在するテキストを Skelton で加工 *(2004/04/02)*

こんなことはないだろうか。いくらかのコードが理由は何であれ突然条件節に変えたくなった。ちょっとしたテキストをある定型的な文書の中に入れたい。これらは skeleton を使えば可能になります。プログラミング言語で埋め込まれたコードを再インデントすることでさえ可能なのです。

skeleton はオプションで数引数 (see ⟨undefined⟩ [(emacs)Arguments], page ⟨undefined⟩) を取ります。この引数は正であれば順方向、負であれば逆方向という 2 種類に解釈されます。

正引数の場合は後に続く語に skeleton を適用することになる。実際には、skeleton を挿入した後でカーソルがある場所に skeleton を適用している語を入れている (see Chapter 1 [Using Skeletons], page 2)。そして、ポイント (see ⟨undefined⟩ [(emacs)Point], page ⟨undefined⟩) は skeleton で次に入力すべき場所に移ります。

負引数は前もってマークされた中間リージョン (see ⟨undefined⟩ [(emacs)Mark], page ⟨undefined⟩) に対して同様のことを行う。単純な例では、skeleton コマンドを実行する前に `M--`を入力すると、リージョン内のテキストを内包するように働く。これは、正引数がいくつかの単語を内包するように働くのに対応します。

より小さな負引数を指定すると skeleton コマンドは連続して多くの中間リージョン (interregions) 内文字を必要な場所へ埋め込み、カーソルを次に入力すべき場所へ移動します。ここでリージョンではなく、中間リージョン (interregions) という言葉を使ってきました。これは、skelton が見つかった順番に処理を行い、マークされたリージョンが順番に並んでいる時としか同じにならないためです。

つまり、A B C []([] はカーソル位置) というテキストがあって、アルファベット順にマークしていったとします。そして、引数として `M-- 3` を与えて skeleton コマンドを実行したとします。すると、最初に skelton の必要な場所へ A から B のテキストが入り、次に B から C までが、3 番目に C からポイントまでが入り、カーソルは 4 番目に入力すべき場所へ移動します。もし引数よりもマークが少ない、あるいは skelton の定義場所が少ない時には、余分なものが無視されます。

一方、[] A C B というテキストをアルファベット順にマークしたとします。そして、引数として `M-- 3` を与えて skeleton コマンドを実行すると、

3 略称展開 *(2004/04/02)*

すべての skeleton コマンドをキーバインドを割り当てなくても，略称を定義する (see ⟨undefined⟩ [(emacs)Defining Abbrevs], page ⟨undefined⟩) し，skeleton として展開 (see ⟨undefined⟩ [(emacs)Expanding Abbrevs], page ⟨undefined⟩) させることもできます.

C 言語で ‘`ifst`’を通常の構文の時に略称として登録したい場合には，‘`ifst`’ を空文字の略称とし，それから skelton コマンドを呼ぶようにします．Emacs Lisp では `(define-abbrev c-mode-abbrev-table "ifst" "" 'c-if)` のようにすることができます．あるいは，*M-x list-abbrevs* から出力を以下のように編集することもできます.

```
(c-mode-abbrev-table)
"if"        0    ""        c-if
```

(意味のない空行や他モードの略称との間にある空行は削除されます)

4 Skeleton 言語 *(2004/04/02)*

skeleton は Lisp を少し拡張したもので，さまざまなアトムはカレントバッファでコマンドや一部の作業を実行することができます．skeleton は `skeleton-insert` により処理されます．

skelton とは相互に情報をやりとりするためのリストで，大抵はプロンプトに表示する文字になります．必要がなければ，`nil` になりますが，複雑な入力関数や計算した値を返すような Lisp 言語でも構いません．リストの残りの部分は以下の表で説明する要素になります．

`"`*`string`*`"`, `?`*`c`*, `?\`*`c`*
: 文字列や文字を挿入する．文字通りの文字列や文字は `skeleton-transformation` が 非 `nil` であれば，その関数で処理される．

`?\n`
: 改行を挿入し，現在行に配置します．配置を調整するためには，`?\n` を使います．

`_`
: ポイントを挿入する．連続するリージョンに対しスケルトンを使うような時に，使われます．ポイント位置は最初の_の左になります．

`>`
: メジャーモードに応じてインデントを行います．続く要素が_で，そこを内包するような内部リージョンがあると，その内部リージョンをインデントします．

`&`
: 論理 AND．カーソルを移動させるような要素 (大抵は文字の挿入など) が先にあると，続く要素を実行します．

`|`
: 論理 OR．カーソルを移動させない要素 (大抵は何も挿入しないようなもの) が先にあると，続く要素を実行します．

`-`*`number`*
: number の数だけ先に続く文字を削除します．`skeleton-untabify` の値に依存します．

`()` or `nil`
: 無視されます．

lisp-expression
: Lisp 式を評価して，戻り値をスケルトンの要素として再び評価します．

`str`
: これは特別な変数で，最初に評価されると，大抵入力を求めます．そして，戻り値で値が設定されます．各サブスケルトンはこの値のローカルコピーを持ちます．

`v1`, `v2`
: スケルトンのローカルユーザ変数

`'`*`expression`*
: Lisp 式を評価しますが，スケルトンの要素としての評価はされません．

skeleton
: サブスケルトンが再帰的に挿入されます．一度だけでなく，ユーザがサブスケルトンとして何かを入力しただけ挿入されます．それにより，サブスケルトンには `str` が必ず存在します．入力を要素を返すような Lisp 式で行う場合には，これらを非対話的に使うこともできます．

`resume:`
: 無視されます．スケルトンの解釈中にユーザが停止すると，ここで実行が再開します．

`quit`
: ユーザによる中止のために `resume:` 部分が実行されると 非 `nil` な値となる定数．

いくつかのモードでは，既に定義した他のスケルトンも使用します．例えば，shell スクリプトモードのスケルトンでは, `<` が後方への厳密なインデントになっていますし, CC モードでは`{` と `}` が自身を挿入する要素となっています．これらはバッファローカル変数である `skeleton-further-elements` で定義されており，スケルトンを解釈する際に利用されるリストになっています．

マクロ `define-skeleton` はスケルトンを解釈するための関数を定義します．1 番目の引数は関数名です．2 番目は説明用のドキュメントで，残りは解釈子やスケルトンを共に形成するスケルトン要素になります．このスケルトンは関数と同じ名前の変数として作成され，`~/.emacs` ファイル (see ⟨undefined⟩ [(emacs)Init File], page ⟨undefined⟩) で定義できます．

訳者注：skelton.el には下記の例が載せられています．評価して M-x local-variables-section を試してみてください．

```
(define-skeleton local-variables-section
  "Insert a local variables section.  Use current comment syntax if any."
  (completing-read
   "Mode: " obarray
   (lambda (symbol)
     (if (commandp symbol)
         (string-match "-mode$" (symbol-name symbol))))
   t)
  '(save-excursion
     (if (re-search-forward page-delimiter nil t)
         (error "Not on last page")))
  comment-start "Local Variables:" comment-end \n
  comment-start "mode: " str
  & -5 | '(kill-line 0) & -1 | comment-end \n
  ((completing-read
    (format "Variable, %s: " skeleton-subprompt)
    obarray
    (lambda (symbol)
      (or (eq symbol 'eval)
          (user-variable-p symbol)))
    t)
    comment-start str ": "
    (read-from-minibuffer
     "Expression: " nil read-expression-map nil
     'read-expression-history) | _
     comment-end \n)
  resume:
  "\n"
  comment-start "End:" comment-end \n)
```

5 文字のペアを挿入

いくつかの文字は通常ペアで用いられます．例えば，開き括弧を挿入する時，プログラミング言語であれ散文であれ，大抵閉じ括弧を入力するでしょう．この両者を同時に入力し，カーソルを間に残すことで，Emacs はそのような括弧の釣り合いが取れていることを保証できるのです．そして，もし qwerty キーボードではなく，ある種のプログラミング言語においてシンボルを入力するのに指を無理に曲げないといけないような時でも，苦痛を取り除くこともできるでしょう．

これは `self-insert-command` の代わりに `skeleton-pair-insert-maybe` にバインドする (see ⟨undefined⟩ [(emacs)Rebinding], page ⟨undefined⟩) ことで実現されています．「ひょっとすると」最初はこの驚かされる挙動は初期時にはオフになっていることに端を発するのかもしれません．これを有効にするためには，`skeleton-pair` を非 `nil` にしなければなりません．そして，正引数 (see ⟨undefined⟩ [(emacs)Arguments], page ⟨undefined⟩) を与えると，通常の挿入キー (see ⟨undefined⟩ [(emacs)Inserting Text], page ⟨undefined⟩) と同じ動作になります．

続く文字が単語の一部などで，ペアでの挿入が起こって欲しくなく，いつも釣り合いが取れたペアになるような挙動をやめたくなるような場合があります．そのような時でさえペアで挿入したい場合には，`skeleton-pair-on-word` を非 `nil` にします．

ペアにすることはすべての文字で可能です．デフォルトでは丸括弧 ‘(’，角括弧 ‘[’，大括弧 ‘{’，‘<’，逆引用符 (バッククォート) ‘‘’ といった左右対象となるペアです．他のすべての文字のペアはその文字自身になっています．これは変数 `skeleton-pair-alist` で制御できます．実際には，これはスケルトンのリストで (see Chapter 4 [Skeleton Language], page 5)，各リストの最初は入力した文字に一致する部分になります．この位置は解釈する位置になりますが，`str` 要素は必要なく無視されます．

いくつかのモードでは関数 `skeleton-pair-insert-maybe` を適切なキーに割り当てています．これらのモードではペアについても適切に設定しています．例えば，英語の散文を入力する時，逆引用符 (‘‘’) を入力すると，引用符 (‘’’) とペアになることを期待するでしょう．Shell スクプトモードではペアでなければなりません．ある文脈ではペアにすることを禁止できます．例えば，エスケープするとその文字自身を意味します．

6 新規ファイルへの自動挿入

M-x auto-insert とすると，バッファの先頭にあらかじめ定義されたテキストを挿入します．この関数は，その名前が示すように，主に新規作成時や空のファイルを開いた時にだけ機能するように使います．これを実現するには，`(add-hook 'find-file-hook 'auto-insert)` を `~/.emacs` ファイルに追加します (see ⟨undefined⟩ [(emacs)Init File], page ⟨undefined⟩).

訳者注：これで自動挿入が動作しない場合には `(auto-insert-mode t)` も追加してみてください．

挿入されるものが何であっても変数 `auto-insert-alist` で定義されます．リストの CAR はモード名で，そのバッファがそのモードであれば，要素が実行されます．あるいは，リストの CAR が文字である場合には，バッファのファイル名が正規表現に一致するかどうかで適用するかどうかが決まります．これにより同じモードを持ついろいろな種類のファイルを Emacs は区別できるのです．リストの CAR は上述したようなモード名や正規表現，さらなる記述成るコンスセルかもしれません．

一致する要素があると，その CDR が何をするべきかを表します．それは，挿入されるファイルのファイル名かもしれません．もし，ファイルが `auto-insert-directory` にあるか絶対パスで存在していれば，そのファイルは挿入されます．そうでなければ，挿入されるべきスケルトンとして処理されます (see Chapter 4 [Skeleton Language], page 5) .

あるいは様々なことを行うための関数であるかもしれません．関数は単にいくつかのテキストを挿入することもできますし，スケルトンも利用できます (see Chapter 1 [Using Skeletons], page 2)．例えば他の関数を呼ぶようなラムダ関数でも構いません．また，そのような機能を順番に実行したければ，ベクトルを使うこともできます．つまり，上記の要素を角括弧 (‘`[...]`’) で括ればいいのです．

デフォルトでは C や C++ ヘッダファイルでは多重読み込みを避けるため，ファイル名から得られたシンボル定義を挿入します．C や C++のソーソファイルではヘッダのインクルードを挿入します．Makefile では存在すれば makefile.inc を挿入します．

TeX や bibTeX モードでは存在すれば tex-insert.tex を挿入します，一方，LaTeX モードのファイルでは典型的な `\documentclass` の枠組みを挿入します．html ファイルは最低限の枠組みをスケルトンで挿入します．

Ada モードでは Ada のヘッダ挿入のためのスケルトンが実行されます．Emacs Lisp では通常のヘッダである `$ORGANIZATION` に応じたたものか FSF のコピーライト，キーワード，内容の記述を挿入します．`bin` ディレクトリ (パスが通ったところ) にあって，Emacs がモードを決定できない (see ⟨undefined⟩ [(emacs)Choosing Modes], page ⟨undefined⟩) ファイルは shell スクリプトモードになります．

Lisp 式では (see ⟨undefined⟩ [(emacs)Init File], page ⟨undefined⟩)，関数 `define-auto-insert` を使うことで `auto-insert-alist` に要素を追加したり編集したりできます．*C-h f auto-insert-alist* とすることで関数のドキュメントを参照ください．

変数 `auto-insert` は `auto-insert` が非対話的に呼ばれた際 (ファイルが空だった場合など) どうすべきかを決定します．

`nil` 何もしない．

`t` もし可能なら何かを挿入します．つまり，`auto-insert-alist` に一致する要素があれば挿入します．

other 可能なら何かを挿入するが，未変更のままとする．

変数 `auto-insert-query` は何かを挿入する際に確認するかどうかを制御します．`nil` であれば，挿入は *M-x auto-insert* で実行した時だけになります． `function` であれば，空のファイルを開いた時や，上記の hook を設定した場合など，`auto-insert` が関数として呼ばれる時はいつでも確認されます．それ以外なら，いつも確認されます．

確認時に変数 `auto-insert-prompt` の値が y-or-n タイプのプロンプトとして使われます．もし，‘`%s`’を含むと，選択された挿入規則で置き換えられます．これは説明文かバッファのモード名，ファイル名に一致した正規表現のいづれかになります．

7 著作権表示の挿入と更新

M-x copyright はスケルトンの挿入コマンドで, ポイント位置に著作権表示を追加します.「by」部分には環境変数 `$ORGANIZATION` の値か設定されていなければプロンプトで入力された値が使われます. バッファにコメントの記述方法 (see ⟨undefined⟩ [(emacs)Comments], page ⟨undefined⟩) があれば, コメントとして挿入します.

M-x copyright-update は `copyright-limit` で制限された箇所まで著作権表示を探し, 必要なら更新します. 現在年 (変数 `copyright-current-year`) が既存のものに同じ書式 (つまり, 1994 や' 94, 94) で追加されます. ダッシュで区切られたリストに最新年があれば, 現在年まで拡大するかカンマで区切られた年を追加します. 前置引数をつけて実行した場合には元のものを置き換えます. ヘッダが間違ったバージョンの GNU General Public License (see ⟨undefined⟩ [(emacs)Copying], page ⟨undefined⟩) を参照している場合には, アップデートも行います.

この関数は保存時に毎回実行するようにしておくと簡単でいいでしょう. これは `(add-hook 'write-file-functions 'copyright-update)` を`~/.emacs` ファイルに追加 (see ⟨undefined⟩ [(emacs)Init File], page ⟨undefined⟩) することで実現できます.

変数 `copyright-query` は著作権表示をアップデートするかを尋ねるかどうかを制御します. `nil` なら, *M-x copyright-update* を実行した時のみ更新されます. `function` なら, `write-file-functions` のように `copyright-update` が関数として呼ばれた時はいつでも更新されます. もしそうでなければ, 確認されます.

8 スクリプトインタプリタの作成

Shell スクリプトモードや AWK モードのようにさまざまなインタプリタモードは自動的にバッファのマジックナンバや最初の行に置かれ `exec` システムコールにスクリプトを実行させる方法を教えるための特別なコメントを挿入や更新します．保存時には自動的に `executable-chmod` を引数としてシステムの `chmod` コマンドを実行します．マジックナンバーは `executable-prefix` の値が置かれます．

ファイル名が `executable-magicless-file-regexp` に一致するものはマジックナンバーがなく，実行可能にもしません．これは主にリソースファイル向けのもので，読み込み用のファイルに適用されます．

変数 `executable-insert` は `executable-set-magic` が非対話的に実行された時 (ファイルにマジックナンバが無い時や間違っていた時など) にどう処理するかを決定しています．

`nil` 何もしません．

`t` マジックナンバを挿入し更新します．

other マジックナンバを挿入し更新しますは，未更新のままとします．

変数 `executable-query` はマジックナンバーを挿入や更新する際に確認するかどうかを制御します．`nil` であれば，*`M-x executable-set-magic`* で実行した時のみ更新します．`function` であれば， `executable-set-magic` が shell スクリプトモードで呼ばれた時のように関数として実行された時はいつでも確認します．そうでなければ，いつでも確認します．

`M-x executable-self-display` adds a magic number to the buffer, which will turn it into a self displaying text file, when called as a Un*x command. The “interpreter” used is `executable-self-display` with argument ‘`+2`’.

9 ファイルの変更時間を更新

time-stamp はファイルを保存する時に新しい更新時間を自動的にある書式で更新するためのものです．write-file-functions に関数 time-stamp を追加するようにカスタマイズします．

更新時間は time-stamp-active がオンの時 (デフォルト) のみ更新されます．time-stamp-toggle-active はこれをオン/オフできます．更新時間の書式は変数 time-stamp-format をカスタマイッズすることで変更できます．

変数 time-stamp-line-limit，time-stamp-start，time-stamp-end，time-stamp-count，time-stamp-inserts-lines はテンプレートを探す方法を制御します．これらの変数は初期化ファイル (.emacs) で変更しない方がいいでしょう．さもないと，他の人が作成したファイルを編集する際に問題となります．変更する場合には，ファイルのローカル変数で変更しなければなりません．

通常は，テンプレートはファイルの 8 行目までに以下のような書式がなければなりません．

```
Time-stamp: <>
Time-stamp: " "
```

更新時間は括弧や引用符の間に記入されます．

```
Time-stamp: <1998-02-18 10:20:51 gildea>
```

10 QuickURL: ポイント位置のテキストに応じて URL 挿入

M-x quickurl はポイント付近のテキストに応じてテキストを挿入することができます．URL は変数 quickurl-url-file で設定されたファイルで定義されており，(*key* . *URL*) という cons cells か (*key URL comment*) というリストになっています．ポイント位置の単語が *key* であれば，*M-x quickurl* は *URL* を挿入します．例えば，以下のような設定が可能です．

```
(("FSF"      "http://www.fsf.org/" "The Free Software Foundation")
 ("emacs"  . "http://www.emacs.org/")
 ("hagbard"  "http://www.hagbard.demon.co.uk" "Hagbard's World"))
```

M-x quickurl-add-url は新規に *key*/*URL* のペアを追加できます．*M-x quickurl-list* で URL リストの対話的編集ができます．

11 Tempo: 柔軟なテンプレート挿入

Tempo パッケージは機能的なテンプレートやマクロを簡単に定義できます．主にある種のドキュメントを簡単に編集できる方法を作るプログラマを意図しています (それに限定しているわけではありませんが).

テンプレートはポイント位置に挿入されるべきアイテムのリストとして定義されます．単純な文字列もあれば，書式を制御するようなものや挿入したテキストのある箇所を定義するようなものもあります．*M-x tempo-backward-mark* と *M-x tempo-forward-mark* でそうしたポイントを順番に移動していくことができます.

より柔軟なテンプレートとして Lisp シンボルを含むこともできます．その Lisp シンボルは変数やリスト，Lisp 式として評価されます．特定のタグの自動補完機能も用意されています.

Tempo テンプレートを定義するために使われる他の要素は `tempo-define-template` のドキュメントを参照ください.

Tempo に関する詳細は `tempo.el` のコメントを参照ください.

12 ‘Hippie’ 展開

M-x hippie-expand は 1 つのコマンドでさまざまな補完や展開を行うことができます。繰り返し実行すると、すべての候補を連続で展開できます。

どの方法をどういう順番で実行するかは 変数 hippie-expand-try-functions-list で決まります。カスタマイズすることで、順番を変えたり、ある方法を削除したり、新しい方法を追加したりできます。数引数を与えると、*M-x hippie-expand* は hippie-expand-try-functions-list で「数引数」番目にある関数が実行されます。他の引数 (負引数や *C-u* のみ) 補完されたテキストを元に戻します。

より詳しいことは hippie-exp.elのコメントを参照してください。

一般的には hippie-expand を dabbrev-expand が割り当てられている *M-/* にバインドします。そして、dabbrev-expand は補完する関数の 1 つ (try-expand-dabbrev などが容易されています) にしておくのです。

概念索引

(Index is nonexistent)

コマンド索引

(Index is nonexistent)

変数索引

(Index is nonexistent)

Table of Contents