Autotyping

Convenient features for text that you enterfrequently in Emacs

Daniel Pfeiffer
additions by Dave Love

Autotyping

Under certain circumstances you will find yourself typing similar things over and over again. This is especially true of form letters and programming language constructs. Project-specific header comments, flow-control constructs or magic numbers are essentially the same every time. Emacs has various features for doing tedious and repetitive typing chores for you in addition to the Abbrev features (see ⟨undefined⟩ [(emacs)Abbrevs], page ⟨undefined⟩).

One solution is using skeletons, flexible rules that say what to insert, and how to do it. Various programming language modes offer some ready-to-use skeletons, and you can adapt them to suit your needs or taste, or define new ones.

Another feature is automatic insertion of what you want into empty files, depending on the file-name or the mode as appropriate. You can have a file or a skeleton inserted, or you can call a function. Then there is the possibility to have Un*x interpreter scripts automatically take on a magic number and be executable as soon as they are saved. Or you can have a copyright notice's year updated, if necessary, every time you save a file. Similarly for time stamps in the file.

URLs can be inserted based on a word at point. Flexible templates can be defined for inserting and navigating between text more generally. A sort of meta-expansion facility can be used to try a set of alternative completions and expansions of text at point.

1 Using Skeletons

When you want Emacs to insert a form letter or a typical construct of the programming language you are using, skeletons are a means of accomplishing this. Normally skeletons each have a command of their own, that, when called, will insert the skeleton. These commands can be issued in the usual ways (see ⟨undefined⟩ [(emacs)Commands], page ⟨undefined⟩). Modes that offer various skeletons will often bind these to key-sequences on the `C-c` prefix, as well as having an *Insert* menu and maybe even predefined abbrevs for them (see Chapter 3 [Skeletons as Abbrevs], page 4).

The simplest kind of skeleton will simply insert some text indented according to the major mode and leave the cursor at a likely place in the middle. Interactive skeletons may prompt you for a string that will be part of the inserted text.

Skeletons may ask for input several times. They even have a looping mechanism in which you will be asked for input as long as you are willing to furnish it. An example would be multiple "else if" conditions. You can recognize this situation by a prompt ending in ⟨RET⟩, `C-g` or `C-h`. This means that entering an empty string will simply assume that you are finished. Typing quit on the other hand terminates the loop but also the rest of the skeleton, e.g. an "else" clause is skipped. Only a syntactically necessary termination still gets inserted.

2 Wrapping Skeletons Around Existing Text

Often you will find yourself with some code that for whatever reason suddenly becomes conditional. Or you have written a bit of text and want to put it in the middle of a form letter. Skeletons provide a means for accomplishing this, and can even, in the case of programming languages, reindent the wrapped code for you.

Skeleton commands take an optional numeric prefix argument (see ⟨undefined⟩ [(emacs)Arguments], page ⟨undefined⟩). This is interpreted in two different ways depending on whether the prefix is positive, i.e. forwards oriented or negative, i.e. backwards oriented.

A positive prefix means to wrap the skeleton around that many following words. This is accomplished by putting the words there where the point is normally left after that skeleton is inserted (see Chapter 1 [Using Skeletons], page 2). The point (see ⟨undefined⟩ [(emacs)Point], page ⟨undefined⟩) is left at the next interesting spot in the skeleton instead.

A negative prefix means to do something similar with that many precedingly marked interregions (see ⟨undefined⟩ [(emacs)Mark], page ⟨undefined⟩). In the simplest case, if you type `M--` just before issuing the skeleton command, that will wrap the skeleton around the current region, just like a positive argument would have wrapped it around a number of words.

Smaller negative arguments will wrap that many interregions into successive interesting spots within the skeleton, again leaving the point at the next one. We speak about interregions rather than regions here, because we treat them in the order they appear in the buffer, which coincides with successive regions only if they were marked in order.

That is, if you marked in alphabetical order the points A B C [] (where [] represents the point) and call a skeleton command with `M-- 3`, you will wrap the text from A to B into the first interesting spot of the skeleton, the text from B to C into the next one, the text from C to the point into the third one, and leave the point in the fourth one. If there are less marks in the buffer, or if the skeleton defines less interesting points, the surplus is ignored.

If, on the other hand, you marked in alphabetical order the points [] A C B, and call a skeleton command with `M-- 3`, you will wrap the text from point to A, then the text from A to C and finally the text from C to B. This is done because the regions overlap and Emacs would be helplessly lost if it tried to follow the order in which you marked these points.

3 Skeletons as Abbrev Expansions

Rather than use a key binding for every skeleton command, you can also define an abbreviation (see ⟨undefined⟩ [(emacs)Defining Abbrevs], page ⟨undefined⟩) that will expand (see ⟨undefined⟩ [(emacs)Expanding Abbrevs], page ⟨undefined⟩) into the skeleton.

Say you want '`ifst`' to be an abbreviation for the C language if statement. You will tell Emacs that '`ifst`' expands to the empty string and then calls the skeleton command. In Emacs-lisp you can say something like `(define-abbrev c-mode-abbrev-table "ifst" "" 'c-if)`. Or you can edit the output from *M-x list-abbrevs* to make it look like this:

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

(Some blank lines of no semantic significance, and other abbrev tables, have been omitted.)

4 Skeleton Language

Skeletons are an shorthand extension to the Lisp language, where various atoms directly perform either actions on the current buffer or rudimentary flow control mechanisms. Skeletons are interpreted by the function `skeleton-insert`.

A skeleton is a list starting with an interactor, which is usually a prompt-string, or `nil` when not needed, but can also be a Lisp expression for complex read functions or for returning some calculated value. The rest of the list are any number of elements as described in the following table:

`"`*string*`"`, `?`*c*, `?\`*c*
: Insert string or character. Literal strings and characters are passed through `skeleton-transformation` when that is non-`nil`.

`?\n`
: Insert a newline and align under current line. Use newline character `?\n` to prevent alignment.

`_`
: Interesting point. When wrapping skeletons around successive regions, they are put at these places. Point is left at first `_` where nothing is wrapped.

`>`
: Indent line according to major mode. When following element is `_`, and there is a interregion that will be wrapped here, indent that interregion.

`&`
: Logical and. Iff preceding element moved point, i.e. usually inserted something, do following element.

`|`
: Logical xor. Iff preceding element didn't move point, i.e. usually inserted nothing, do following element.

`-`*number*
: Delete preceding number characters. Depends on value of `skeleton-untabify`.

`()` or `nil`
: Ignored.

lisp-expression
: Evaluated, and the return value is again interpreted as a skeleton element.

`str`
: A special variable that, when evaluated the first time, usually prompts for input according to the skeleton's interactor. It is then set to the return value resulting from the interactor. Each subskeleton has its local copy of this variable.

`v1`, `v2`
: Skeleton-local user variables.

`'`*expression*
: Evaluate following lisp expression for its side-effect, but prevent it from being interpreted as a skeleton element.

skeleton
: Subskeletons are inserted recursively, not once, but as often as the user enters something at the subskeletons interactor. Thus there must be a `str` in the subskeleton. They can also be used non-interactively, when prompt is a lisp-expression that returns successive list-elements.

`resume:`
: Ignored. Execution resumes here if the user quits during skeleton interpretation.

`quit`
: A constant which is non-`nil` when the `resume:` section was entered because the user quit.

Some modes also use other skeleton elements they themselves defined. For example in shell script mode's skeletons you will find `<` which does a rigid indentation backwards, or in CC mode's skeletons you find the self-inserting elements `{` and `}`. These are defined by the buffer-local variable `skeleton-further-elements` which is a list of variables bound while interpreting a skeleton.

The macro `define-skeleton` defines a command for interpreting a skeleton. The first argument is the command name, the second is a documentation string, and the rest is an interactor and any number of skeleton elements together forming a skeleton. This skeleton is assigned to a variable of the same name as the command and can thus be overridden from your '`~/.emacs`' file (see ⟨undefined⟩ [(emacs)Init File], page ⟨undefined⟩).

5 Inserting Matching Pairs of Characters

Various characters usually appear in pairs. When, for example, you insert an open parenthesis, no matter whether you are programming or writing prose, you will surely enter a closing one later. By entering both at the same time and leaving the cursor inbetween, Emacs can guarantee you that such parentheses are always balanced. And if you have a non-qwerty keyboard, where typing some of the stranger programming language symbols makes you bend your fingers backwards, this can be quite relieving too.

This is done by binding the first key (see ⟨undefined⟩ [(emacs)Rebinding], page ⟨undefined⟩) of the pair to `skeleton-pair-insert-maybe` instead of `self-insert-command`. The “maybe” comes from the fact that this at-first surprising behavior is initially turned off. To enable it, you must set `skeleton-pair` to some non-`nil` value. And even then, a positive argument (see ⟨undefined⟩ [(emacs)Arguments], page ⟨undefined⟩) will make this key behave like a self-inserting key (see ⟨undefined⟩ [(emacs)Inserting Text], page ⟨undefined⟩).

While this breaks with the stated intention of always balancing pairs, it turns out that one often doesn’t want pairing to occur, when the following character is part of a word. If you want pairing to occur even then, set `skeleton-pair-on-word` to some non-`nil` value.

Pairing is possible for all visible characters. By default the parenthesis ‘(’, the square bracket ‘[’, the brace ‘{’, the pointed bracket ‘<’ and the backquote ‘`’ all pair with the symmetrical character. All other characters pair themselves. This behavior can be modified by the variable `skeleton-pair-alist`. This is in fact an alist of skeletons (see Chapter 4 [Skeleton Language], page 5), with the first part of each sublist matching the typed character. This is the position of the interactor, but since pairs don’t need the `str` element, this is ignored.

Some modes have bound the command `skeleton-pair-insert-maybe` to relevant keys. These modes also configure the pairs as appropriate. For example, when typing english prose, you’d expect the backquote (‘`’) to pair with the quote (‘'’), while in Shell script mode it must pair to itself. They can also inhibit pairing in certain contexts. For example an escaped character stands for itself.

6 Autoinserting Text in Empty Files

M-x auto-insert will put some predefined text at the beginning of the buffer. The main application for this function, as its name suggests, is to have it be called automatically every time an empty, and only an empty file is visited. This is accomplished by putting `(add-hook 'find-file-hooks 'auto-insert)` into your '`~/.emacs`' file (see ⟨undefined⟩ [(emacs)Init File], page ⟨undefined⟩).

What gets inserted, if anything, is determined by the variable `auto-insert-alist`. The CARs of this list are each either a mode name, making an element applicable when a buffer is in that mode. Or they can be a string, which is a regexp matched against the buffer's file name. In that way different kinds of files that have the same mode in Emacs can be distinguished. The CARs may also be cons cells consisting of mode name or regexp as above and an additional descriptive string.

When a matching element is found, the CDR says what to do. It may be a string, which is a file name, whose contents are to be inserted, if that file is found in the directory `auto-insert-directory` or under a absolute file name. Or it can be a skeleton (see Chapter 4 [Skeleton Language], page 5) to be inserted.

It can also be a function, which allows doing various things. The function can simply insert some text, indeed, it can be skeleton command (see Chapter 1 [Using Skeletons], page 2). It can be a lambda function which will for example conditionally call another function. Or it can even reset the mode for the buffer. If you want to perform several such actions in order, you use a vector, i.e. several of the above elements between square brackets ('`[...]`').

By default C and C++ headers insert a definition of a symbol derived from the filename to prevent multiple inclusions. C and C++ sources insert an include of the header. Makefiles insert the file makefile.inc if it exists.

TeX and bibTeX mode files insert the file tex-insert.tex if it exists, while LaTeX mode files insert a typical `\documentclass` frame. Html files insert a skeleton with the usual frame.

Ada mode files call the Ada header skeleton command. Emacs lisp source files insert the usual header, with a copyright of your environment variable `$ORGANIZATION` or else the FSF, and prompt for valid keywords describing the contents. Files in a '`bin`' directory for which Emacs could determine no specialised mode (see ⟨undefined⟩ [(emacs)Choosing Modes], page ⟨undefined⟩) are set to Shell script mode.

In Lisp (see ⟨undefined⟩ [(emacs)Init File], page ⟨undefined⟩) you can use the function `define-auto-insert` to add to or modify `auto-insert-alist`. See its documentation with *C-h f auto-insert-alist*.

The variable `auto-insert` says what to do when `auto-insert` is called non-interactively, e.g. when a newly found file is empty (see above):

`nil` Do nothing.

`t` Insert something if possible, i.e. there is a matching entry in `auto-insert-alist`.

other Insert something if possible, but mark as unmodified.

The variable `auto-insert-query` controls whether to ask about inserting something. When this is `nil`, inserting is only done with *M-x auto-insert*. When this is `function`, you are queried whenever `auto-insert` is called as a function, such as when Emacs visits an empty file and you have set the above-mentioned hook. Otherwise you are alway queried.

When querying, the variable `auto-insert-prompt`'s value is used as a prompt for a y-or-n-type question. If this includes a '`%s`' construct, that is replaced by what caused the insertion rule to be chosen. This is either a descriptive text, the mode-name of the buffer or the regular expression that matched the filename.

7 Inserting and Updating Copyrights

M-x copyright is a skeleton inserting command, that adds a copyright notice at the point. The “by” part is taken from your environment variable `$ORGANIZATION` or if that isn't set you are prompted for it. If the buffer has a comment syntax (see ⟨undefined⟩ [(emacs)Comments], page ⟨undefined⟩), this is inserted as a comment.

M-x copyright-update looks for a copyright notice in the first `copyright-limit` characters of the buffer and updates it when necessary. The current year (variable `copyright-current-year`) is added to the existing ones, in the same format as the preceding year, i.e. 1994, '94 or 94. If a dash-separated year list up to last year is found, that is extended to current year, else the year is added separated by a comma. Or it replaces them when this is called with a prefix argument. If a header referring to a wrong version of the GNU General Public License (see ⟨undefined⟩ [(emacs)Copying], page ⟨undefined⟩) is found, that is updated too.

An interesting application for this function is to have it be called automatically every time a file is saved. This is accomplished by putting `(add-hook 'write-file-hooks 'copyright-update)` into your '`~/.emacs`' file (see ⟨undefined⟩ [(emacs)Init File], page ⟨undefined⟩).

The variable `copyright-query` controls whether to update the copyright or whether to ask about it. When this is `nil` updating is only done with *M-x copyright-update*. When this is `function` you are queried whenever `copyright-update` is called as a function, such as in the `write-file-hooks` feature mentioned above. Otherwise you are always queried.

8 Making Interpreter Scripts Executable

Various interpreter modes such as Shell script mode or AWK mode will automatically insert or update the buffer's magic number, a special comment on the first line that makes the `exec` systemcall know how to execute the script. To this end the script is automatically made executable upon saving, with `executable-chmod` as argument to the system `chmod` command. The magic number is prefixed by the value of `executable-prefix`.

Any file whose name matches `executable-magicless-file-regexp` is not furnished with a magic number, nor is it made executable. This is mainly intended for resource files, which are only meant to be read in.

The variable `executable-insert` says what to do when `executable-set-magic` is called non-interactively, e.g. when file has no or the wrong magic number:

`nil` Do nothing.

`t` Insert or update magic number.

other Insert or update magic number, but mark as unmodified.

The variable `executable-query` controls whether to ask about inserting or updating the magic number. When this is `nil` updating is only done with *M-x executable-set-magic*. When this is `function` you are queried whenever `executable-set-magic` is called as a function, such as when Emacs puts a buffer in Shell script mode. Otherwise you are alway queried.

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 Maintaining Timestamps in Modified Files

The `time-stamp` command can be used to update automatically a template in a file with a new time stamp every time you save the file. Customize the hook `write-file-hooks` to add the function `time-stamp` to arrange this.

The time stamp is updated only if the customizable variable `time-stamp-active` is on, which it is by default; the command `time-stamp-toggle-active` can be used to toggle it. The format of the time stamp is set by the customizable variable `time-stamp-format`.

The variables `time-stamp-line-limit`, `time-stamp-start`, `time-stamp-end`, `time-stamp-count`, and `time-stamp-inserts-lines` control finding the template. Do not change these in your init file or you will be incompatible with other people's files. If you must change them, do so only in the local variables section of the file itself.

Normally the template must appear in the first 8 lines of a file and look like one of the following:

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

The time stamp is written between the brackets or quotes:

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

10 QuickURL: Inserting URLs Based on Text at Point

M-x quickurl can be used to insert a URL into a buffer based on the text at point. The URLs are stored in an external file defined by the variable `quickurl-url-file` as a list of either cons cells of the form `(`*key* `.` *URL*`)` or lists of the form `(`*key URL comment*`)`. These specify that *M-x quickurl* should insert *URL* if the word *key* is at point, for example:

```
(("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 can be used to add a new *key*/*URL* pair. *M-x quickurl-list* provides interactive editing of the URL list.

11 Tempo: Flexible Template Insertion

The Tempo package provides a simple way to define powerful templates, or macros, if you wish. It is mainly intended for, but not limited to, programmers to be used for creating shortcuts for editing certain kinds of documents.

A template is defined as a list of items to be inserted in the current buffer at point. Some can be simple strings, while others can control formatting or define special points of interest in the inserted text. `M-x tempo-backward-mark` and `M-x tempo-forward-mark` can be used to jump between such points.

More flexible templates can be created by including lisp symbols, which will be evaluated as variables, or lists, which will be evaluated as lisp expressions. Automatic completion of specified tags to expanded templates can be provided.

See the documentation for `tempo-define-template` for the different items that can be used to define a tempo template with a command for inserting it.

See the commentary in '`tempo.el`' for more information on using the Tempo package.

12 'Hippie' Expansion

M-x hippie-expand is a single command providing a variety of completions and expansions. Called repeatedly, it tries all possible completions in succession.

Which ones to try, and in which order, is determined by the contents of the customizable option `hippie-expand-try-functions-list`. Much customization of the expansion behavior can be made by changing the order of, removing, or inserting new functions in this list. Given a positive numeric argument, *M-x hippie-expand* jumps directly that number of functions forward in this list. Given some other argument (a negative argument or just *C-u*) it undoes the tried completion.

See the commentary in '`hippie-exp.el`' for more information on the possibilities.

Typically you would bind `hippie-expand` to *M-/* with `dabbrev-expand`, the standard binding of *M-/*, providing one of the expansion possibilities.

Concept Index

Command Index

A

C

D

E

H

Q

S

T

Variable Index

A

C

E

H

Q

S

T

W

Table of Contents