Smartparens Documentation

Release 1.10.1

Matúš Goljer

Sep 17, 2017

Contents

Smartparens is minor mode for Emacs that *deals with parens pairs and tries to be smart about it*. It started as a unification effort to combine functionality of several existing packages in a single, compatible and extensible way to deal with parentheses, delimiters, tags and the like. With the basic features found in other packages it also brings many improvements as well as completely new features. Here's a highlight of some features, for a complete list and detailed documentation look in Getting Started below.

- support for pairs of any length, for example `"\{" "\}"` pair used in LaTeX to typeset literal braces in math mode. These are fully *user definable and customizable*. Pairs can have same or different strings for opening and closing part.
- intelligent handling of closing pairs. If user types `(`, `(|)` is inserted. If then the user types `word)` the result is `(word)|` not `(word)||`.
- automatic deletion of complete pairs. With pair `("\{" "\}")`, `\{|\}` and backspace will remove both delimiters. `\{\}|` and backspace will remove the closing pair, and result in `\{|`. Hitting backspace again will remove the opening pair. You can also set it to skip over the pairs to keep the structure balanced instead by enabling `smartparens-strict-mode` (a la paredit).
- wraps active region in defined pairs or special structured tag pairs for "tag-modes" (xml/html...). Different tags are supported, for example, languages that would use `{tag}` instead of `<tag>` or LaTeX's `\begin{}` `\end{}` pair. *Everything* is user definable as usual.
- Jumping around the pairs (extending forward-sexp and similar functions to custom user pairs)
- Functions to manipulate s-expressions, delete, wrap and unwrap, extend and contract..

Almost all features are fully customizable via `M-x customize-group smartparens`. You can turn many behaviours on or off to fit your workflow.

CHAPTER 1

Getting Started

The process of installing smartparens is as simple as calling `package-install`. However, if you want to learn in detail what packages smartparens depend on and what settings it modifies in order to function, read the installation manual.

The quick tour lists the most basic use cases, while here follows a list of features smartparens provide, complete with detailed explanation on how to fine-tune it to your very own taste.

- *Pair management*. How do add and remove pairs, how to overload global pairs locally.
- *Permissions*. How to set permissions for each pair, to restrict in which modes and which contexts it is permitted to automatically insert the closing pair or wrap the active region.
- Action hooks. How to set up hooks to perform custom operations before or after smartparens actions.
- Wrapping. Automatically wrap active regions with pairs or structured tags (for example html tags). Learn how to add or remove tags and how you can customize wrapping to fit your needs.
- Automatic escaping. Were you ever annoyed by quote escaping in strings? Well, no more! Learn how you can let smartparens do the boring for you.
- Navigation. Navigating balanced expressions is a powerful way to move around in buffers and edit code. Learn how you can quickly jump back and forth around the paired expressions.
- Expression manipulation. Extend, contract, split, splice, wrap, unwrap and much more! How to quickly transform your elisp (or any other!) code with powerful manipulation functions.
- Hybrid S-expressions. Sometimes, S-expressions just aren't going to cut it. Learn how to edit (not only) "line based" languages (C, Java, C#,...) in a way that helps you to keep the delimiters balanced.
- Show smartparens mode. Do you want to see where a pair starts and where it ends? Turn on the `show-smartparens-mode` and let smartparens highlight the pairs.
- User interface. Smartparens provide a few user-interface features, like highlighting currently "active" region between pair delimiters and during wrapping. If the defaults doesn't fit your color scheme, read this section and learn how to customize it.
- *Example configuration*. Author's current working smartparens configuration. See how the knowledge you've just obtained works in practice.

- Default configuration. Smartparens ships with powerful default configuration. Chances are that you won't even need to configure anything at all! Look at the internals and see how is smartparens configured by default.

After you familiarize yourself with smartparens, make sure to read tips and tricks to get that extra 20% of coolness out of it ;)

Migrating from paredit

Among other features, smartparens also replicate a lot of paredit functionality and extend it to arbitrary pairs and modes. It also provides many improvements like adding numeric or raw prefix arguments to modify the behaviour of these commands. Read the paredit-and-smartparens article for comparsion of paredit and smartparens features.

CHAPTER 2

Are you an elisp developer?

If you are, check out these articles about the internals of spartparens! There are many interesting functions you can use in your own code that deal with pairs and navigation.

- how-to-use-structural-functions-in-my-own-code. Create powerful functions that operate on pair delimited structures.
- hooks-and-advices-smartparens-modify, this is mostly for the situations where you would modify pre/post command hooks and want to learn how smartparens modify them.

And of course, if you write some cool plugin or extension for smartparens, or have an idea how to do it, contribute!

CHAPTER 3

Contributing

We love contributions. It is most awesome if they come with tests. Read about our testing infrastructure.

CHAPTER 4

Contents

Automatic escaping

Smartparens can automatically escape quote characters (`'`, `"`) and the escape character `\` (as determined by Emacs's syntax tables) when you wrap a region or insert them inside a string.

Autoescaping is specified on a per-pair basis. For convenience, there are two global options to enable or disable escaping after wrapping and after insertion of a pair.

User Option `sp-escape-wrapped-region`
If non-nil, escape special chars inside the just wrapped region.

User Option `sp-escape-quotes-after-insert`
If non-nil, escape string quotes if typed inside string.

Escaping with post hooks

To enable automatic escaping after wrapping or insertion it is necessary to add post-handlers on the pairs where you want to trigger the escaping. These are typically the `"` or `'` quote pairs. Escaping after wrapping is enabled by default.

The hooks are named after the global options—if the option is disabled the function does nothing.

Function `sp-escape-wrapped-region` `()`
Escape quotes and special chars when a region is wrapped.

The wrapping is smart in the sense that it only escapes when necessary. The behaviour of wrapping a word (`bar`) which is already inside a string is summarized by the following table:

enclosing \ wrapped	‘	“
‘	‘foo \’bar\’ baz’	‘foo “bar” baz’
“	“foo ‘bar’ baz”	“foo \”bar\” baz”

The behaviour of wrapping existing text (`foo 'bar' baz` or `foo "bar" baz`) containing a string is summarized by the following table:

containing \ wrapped	‘	“
foo ‘bar’ baz	‘foo \’bar\’ baz’	“foo ‘bar’ baz”
foo “bar” baz	‘foo “bar” baz’	“foo \”bar\” baz”

Function `sp-escape-quotes-after-insert` ()

Escape quotes inserted via ‘sp-insert-pair’.

If auto-pairing is enabled and the pair was successfully inserted inside a string, this hook ensures that it is escaped properly if necessary. The same rules as for wrapping apply, that is, escaping only takes place when necessary. This hook is triggered on the `'insert` action, and so if a `:when` or `:unless` handler prevents pairing, no action is taken.

Escape action

Some users prefer not to pair quotes inside strings and instead only insert the single delimiter (this is the paredit behaviour). However, if the user disabled auto-pairing inside strings, that means the `sp-escape-quotes-after-insert` handler is never called and the quote will get inserted unescaped.

This situation can be handled by adding an `'escape` action on the pair. That tells smartparens to still escape the single inserted delimiter even if the insert action wasn't performed.

Having a separate action might seem extraneous, but it gives us better flexibility in defining the escaping rules to precisely match what is expected. Also, by the nature of the post-handlers, which are only run after the action is performed successfully, it is necessary to have an action separate from `'insert` which might get inhibited by other predicates.

As usual, actions are tried in sequence: `'wrap`, `'insert`, `'escape`, so if the pair is wrapped or inserted the escape action is skipped (and the escaping can be handled with the handlers from previous section).

The single-pair behaviour is summarized in the following table:

enclosing \ inserted	‘	“
‘foo | bar’	‘foo \’| bar’	‘foo “| bar’
“foo | bar”	“foo ‘| bar”	“foo \”| bar”

Pair management

Adding pairs

Function `sp-pair` (open, close)

To define a new pair use the `sp-pair` function. Here is an example of the most basic use:

```
(sp-pair "\{" "\}") ;; latex literal brackets (included by default)
(sp-pair "<#" "#>")
(sp-pair "$" "$")   ;; latex inline math mode. Pairs can have same opening and␣
↪closing string
```

Pairs defined this way are by default used on all *Actions*. However, you can disable certain pairs for auto insertion and only have them for wrapping, only use them for navigation or use any other combination of actions. This is achieved by setting the pair's actions to allow or disable certain operations in certain contexts.

The `sp-pair` function accepts a family of *keyword* arguments which can further specify the behaviour. The keyword arguments can be arbitrarily combined in any order, the only requirement is that the first two positional arguments are always `open` and `close` for the pair.

`(sp-pair :wrap binding)`

You can add a binding for a "wrapping" action. Smartparens automatically binds a command that wraps the next expression with this pair to the supplied binding. The bound command accepts the same prefix arguments as `sp-select-next-thing`. In addition, if a region is already active, it wraps this region.

Note: This is useful in combination with evil visual selection mode, since with regular emacs, smartparens wraps the active regions automatically when you press the delimiter.

Warning: No syntax check is performed on the active region. This might change in the future.

To add the binding use the `:wrap` keyword:

```
(sp-pair "(" ")" :wrap "C-(")
;; |foobar
;; hit C-(
;; becomes (|foobar)
```

`(sp-pair :insert binding :trigger trigger)`

You can also add a binding for "insert" action. This is done the exact same way as for wrapping, but the keyword is `:insert`. Pressing this simply inserts the pair in the buffer. This is useful if you want to insert the pair with a modifier hotkey or a chord. To simply provide a shorter (expandable) trigger, you can specify a `:trigger` keyword.

```
(sp-local-pair 'LaTeX-mode "\\left(" "\\right)" :insert "C-b l" :trigger "\\l(")
```

This will make smartparens insert `\left(|\right)` when you type `\l(` or hit `C-b l` (where `|` is the point). Typing out the entire opening delimiter `\left(` will also work.

Note: Many such commands for LaTeX are provided in configuration file `smartparens-latex.el`. Check it out!

Note: You don't have to use both `:trigger` and `insert`; one or the other (or both) are fine.

It is generally better to add these bindings only to certain major modes where you wish you use this functionality instead of binding them globally to avoid hotkey clashes. See the section about *local pair definitions*.

Removing pairs

You can remove pairs by calling `sp-pair` using the optional key argument `:actions` with value `:rem`. This will also automatically delete any assigned *Permissions*! This command is mostly only useful for debugging or removing built-in pairs.

```
;; the second argument is the closing delimiter, so you need to skip it with nil
(sp-pair "\{" nil :actions :rem)
(sp-pair "'" nil :actions :rem)
```

Default pairs

Since some pairs are so common that virtually every user would use them, smartparens comes with a list of global default pairs. At the moment, this list includes:

```
("\\\\(" . "\\\\)") ;; emacs regexp parens
("\\{"   . "\\}")   ;; latex literal braces in math mode
("\\("   . "\\)")   ;; capture parens in regexp in various languages
("\\\""  . "\\\"")  ;; escaped quotes in strings
("\""    . "\"")    ;; string double quotes
("'"     . "'")     ;; string single quotes/character quotes
("("     . ")")     ;; parens (yay lisp)
("["     . "]")     ;; brackets
("{"     . "}")     ;; braces (a.k.a. curly brackets)
("`"     . "`")     ;; latex strings. tap twice for latex double quotes
```

Local pair definitions

Sometimes, a globally defined pair is not appropriate for certain major modes. You can redefine globally defined pairs to have different definition in specific major modes. For example, globally defined pair `` is used in `markdown-mode` to insert inline code. However, `emacs-lisp-mode` uses `' for links in comments and in `LaTeX-mode` this pair is used for quotes. Since they share the opening sequence (the "trigger"), it's impossible to have both defined globally at the same time. Therefore, it is desired to redefine this global pair to this new value locally.

That is accomplished by using `sp-local-pair` function:

```
(sp-local-pair 'emacs-lisp-mode "`" "'") ;; adds `' as a local pair in emacs-lisp-mode
```

If a global pair with the same trigger does not exist, the pair is defined locally and will only be used in the specified mode. Therefore, you do not need to define a pair globally and then overload it locally. The local definition is sufficient.

Instead of one mode, you can also specify a list to handle multiple modes at the same time (for example `'(emacs-lisp-mode LaTeX-mode)`).

If you specify a parent major mode all the derived modes will automatically inherit all the definitions. If you want to add a pair to all programming modes you can define it for `prog-mode`. If you want to add a pair to all text modes you can define it for `text-mode`.

You can use the following snippet to get all the parent major modes of the `major-mode` in the current buffer

```
(let ((parents (list major-mode)))
  (while (get (car parents) 'derived-mode-parent)
    (push (get (car parents) 'derived-mode-parent) parents))
  parents)
```

Local pairs can be removed by calling `sp-local-pair` with optional keyword argument `:actions` with value `:rem`:

```
(sp-local-pair 'LaTeX-mode "`" nil :actions :rem)
```

Warning: This only removes the pairs you have previously added using `sp-local-pair`. It does not remove/disable a global pair in the specified mode. If you want to disable some pair in specific modes, set its permissions accordingly.

Macro `sp-with-modes` (mode-or-modes **&rest** , forms)

When configuring a mode it is often the case that we modify multiple pairs at the same time. The macro `sp-with-modes` automatically supplies the `mode-or-modes` as first argument to all later forms (it can be a single symbol or a list of symbols for the multiple major modes).

```
(sp-with-modes 'emacs-lisp-mode
  ;; disable ', it's the quote character!
  (sp-local-pair "'" nil :actions nil)
  ;; also only use the pseudo-quote inside strings where it
  ;; serves as hyperlink.
  (sp-local-pair "`" "'" :when '(sp-in-string-p sp-in-comment-p)
```

Named pair definitions (buffer-local)

In addition to using the major mode or a parent mode you can also use an arbitrary symbol as the name of the configuration. This way you can build sets of pairs independent of the major mode hierarchies and you can apply them locally to buffers as you see fit.

The syntax for defining custom named definitions is the same as with `sp-local-pair` only except a major mode as the first argument you pass your desired name.

You can for example define a set of escaped pairs to be used cross-major-mode

```
(sp-local-pair 'escaped-pairs "\\<" "\\>")
(sp-local-pair 'escaped-pairs "\\`" "\\'")
```

You can then apply this definition buffer-locally to the current buffer with

```
(sp-update-local-pairs 'escaped-pairs)
```

This will merge this named configuration into the current buffer's `sp-local-pairs` definitions.

Alternatively you can also use an *anonymous* configuration. The configuration is a plist of arguments with the same meaning as those of `sp-local-pair` with the additional requirement of adding `:open` and `:close` keywords for the opening and closing delimiters. You can also pass a list of such plists to apply all of them at once.

Warning: Make sure to specify at least one action in the `:actions` key otherwise the pair will be removed by the virtue of having no actions. When using `sp-local-pair` many of the keyword arguments get sensible defaults so you don't have to specify them; this is **not** the case when using an anonymous configuration directly. When possible, use mode configrations or named configurations.

This can be used for example for local configuration with major-mode hooks:

```
(defun my-php-mode-init ()
  (sp-update-local-pairs '(:open "#"
                           :close "#"
                           :actions (insert))))
(add-hook 'php-mode-hook 'my-php-mode-init)
```

This will add the # # pair to `php-mode` buffers via the major mode hook.

Permissions

All the permissions settings are handled through these two functions:

- `sp-pair` for setting global pair properties,
- `sp-local-pair` for setting local pair properties.

Each pair has opening and closing delimiter and a number of additional *simple* or *list* properties. Simple properties are simple scalars like a string or a number. List properties are lists of functions, predicates or constants and determine various behaviours of the pair. You can specify all the properties in one call. The distinction between simple and list properties is important when it comes to property inheritance.

There are additional properties you can set that are not handled by this system. See `M-x customize-group smartparens` for available options. This mostly includes enabling and disabling various heuristics to handle special cases.

List property inheritance

Note: If this section is not clear, consider looking at the *Actions* or *Filters* sections first to familiarize yourself with the concept of list properties.

When specifying a list of properties for a local pair, you can use special forms to inherit the global values and add or remove some of them locally. To do this, you specify the list with the special structure:

- if the list is a normal list, the global definition is ignored and this list is used locally. This also means that specifying `nil` will simply remove all the values from the list.
- if the first item in the list is keyword `:add` all the items in the list will be **added** to the global value
- if the first item in the list is keyword `:rem` all the items in the list will be **removed** from the global value
- if the list is of the form `((:add ...) (:rem ...))` the items in the "`:add` list" are added and then the items in the "`:rem` list" are removed from the global value.

Therefore, if you first add a global filter and then wish to remove it locally in a specific major mode, you can do it like this:

```
;; add a global filter
(sp-pair "'" nil :unless '(sp-point-after-word-p))

;; then remove it in c-mode
(sp-local-pair 'c-mode "'" nil :unless '(:rem sp-point-after-word-p))

;; in case there is only one filter the same result would be achieved with
(sp-local-pair 'c-mode "'" nil :unless nil)
```

By default all the properties are inherited from the global definition if one exist.

Actions

Each pair has with it associated a list of actions that it can perform. The supported actions are:

- **insert** - Autoinsert the closing pair when opening pair is typed.
- **wrap** - Wrap an active region with this pair if its opening delimiter is typed while region is active.

- **autoskip** - If the point is before the closing delimiter and you start typing it, smartparens will skip over it in order to not create unbalanced expression. However, this is also controlled by various other settings, see `M-x customize-group RET smartparens RET`, and search for `autoskip`. Setting `autoskip` action will only tell smartparens it is *possible* (not necessary) to do the skipping. Therefore, to completely disable autoskip for a pair, you should remove this action.
- **navigate** - Enable this pair for navigation/highlight and strictness checks.
- **escape** - Allow autoescaping of this delimiter in string contexts.

All of these actions are enabled for each pair by default. If you want to only use pair for a subset of actions, you can specify the explicit actions set by using the keyword argument (just argument from now on) `:actions`:

```
(sp-pair "'" "'" :actions '(wrap))          ;; only use '' pair for wrapping
(sp-pair "%" "%" :actions '(insert))        ;; only use %% pair for auto insertion,␣
↪never for wrapping
(sp-pair "(" ")" :actions '(wrap insert))   ;; use () pair for both actions.
```

To disable some action only in a specific major mode you add the local property for the pair. This is done using the function `sp-local-pair`. To completely disable a pair in a specific mode, you can use `nil` as the list of actions, then no actions will be performed.

```
(sp-local-pair 'emacs-lisp-mode "'" nil :actions nil)           ;; no '' pair in␣
↪emacs-lisp-mode
(sp-local-pair 'markdown-mode "`" nil :actions '(insert))       ;; only use ` for␣
↪auto insertion in markdown-mode
(sp-local-pair 'latex-mode "\\"" nil :actions '(:rem insert))   ;; do not use \" for␣
↪insert action but use it for any other action
```

Note: You have to specify *some* value for the 3rd argument (closing delimiter) in `sp-local-pair`. If you specify nil, the old value is preserved. If you specify a string, this will *locally override* the definiton of the closing pair.

Warning: You must specify the action lists together with all other properties in the same call with which you define the pair. Calling `sp-local-pair` twice will ignore the previous call and reapply the current properties on top of the global definitions.

Filters

Each pair has with it associated two lists of predicates that decide if the action should be performed or not. These are the *when* list and *unless* list.

If the *when* list is not empty, at least one predicate on this list has to return non-nil in order for the action to be performed.

If the *unless* list is not empty, all of the predicates must return nil in order for the action to be performed.

You can use both lists, in which case the formula is:

```
(and (test-predicates when-list)
     (not (test-predicates unless-list)))
```

This expression has to return true in order for the action to be performed.

To specify the filter lists for the pair, you use the arguments `:when` and `:unless`:

```
;; The '' pair will autopair UNLESS the point is right after a word,
;; in which case you want to insert a single apostrophe.
(sp-pair "'" nil :unless '(sp-point-after-word-p))

;; You can also define local filters. Only pair the `' pair inside
;; emacs-lisp-mode WHEN point is inside a string. In other modes, the
;; global definition is used.
(sp-local-pair 'emacs-lisp-mode "`" nil :when '(sp-in-string-p)))
```

The built-in predicates are:

- `sp-in-string-p` - non-nil if point is inside a string
- `sp-in-docstring-p` - non-nil if point is inside an elisp docstring
- `sp-in-code-p` - non-nil if point is inside code
- `sp-in-comment-p` - non-nil if point is inside comment
- `sp-in-math-p` - non-nil if point is inside a LaTeX math block
- `sp-point-in-empty-line-p` - non-nil if point is on an empty line (line filled only with whitespace)

Following predicates are only tested for the `insert` action:

- `sp-point-before-eol-p` - non-nil if point is before whitespace followed by newline
- `sp-point-after-bol-p` - non-nil if point is after beginning of line followed by optional whitespace
- `sp-point-at-bol-p` - non-nil if point is exactly at the beginning of line
- `sp-point-before-symbol-p` - non-nil if point is before symbol
- `sp-point-before-word-p` - non-nil if point is before word, where word is either `w` or _ syntax class
- `sp-point-after-word-p` - non nil if point is after word
- `sp-point-before-same-p` - non-nil if point is before an opening pair same as the currently inserted one

You can supply any number of your own predicates. These predicates should accept three arguments:

- id of the pair, which is the opening delimiter.
- action - See *Actions*.
- context - currently `'string`, `'code` or `'comment`. Note that the string/code distinction only makes sense in programming modes and modes that define what a "string" is.

Here's a definition of the built-in predicate `sp-point-after-word-p` for your inspiration:

```
(defun sp-point-after-word-p (id action context)
  "Return t if point is after a word, nil otherwise.  This
predicate is only tested on \"insert\" action."
  (when (eq action 'insert)
    (save-excursion
      (backward-char 1)
      (looking-back "\\sw\\|\\s_"))))
```

Delayed insertion

When using pairs not made of punctuation but of words, such as Ruby's `def` and `end` pair, we usually don't want to expand `def` in `indefinable`. To solve this issue, we can setup a "delayed insertion". This will tell smartparens to

wait and not insert the closing pair right away, but see what the next action might be. Usually we want to pair when the next action is a `SPC` or `RET` otherwise we don't.

Setting up delayed insertion is very simple. Instead of the usual predicate you add a *list* of *triggers* to the `:when` filter. These triggers will be tested after the next command is run (next command means next interactive function in Emacs).

While you can add this special form to a list of regular tests, if it is present in the `:when` argument it will always take precedence and the insertion will always be delayed.

Here's a shortened example from `smartparens-ruby.el`:

```
(sp-local-pair 'ruby-mode "def" "end"
               :when '(("SPC" "RET" "<evil-ret>"))
               )
```

This will set up a delayed insertion and insert the closing pair only if the *next* action was invoked by `SPC`, `RET` or `<evil-ret>` key. In addition to key codes, you can supply any number of names of the commands (such as `'newline`) or regular `:when` predicates which will be tested after the next command finished.

The predicates on `:unless` list are still considered and if they fail the delayed insertion is not set up.

Make sure to also read the built-in function documentation for `sp-pair` inside Emacs by invoking `C-h f sp-pair RET`.

Pre and post action hooks

Each pair has with it associated two lists of functions that are executed before and after an action is performed with this pair. These lists are:

- pre-handlers - functions that run *before* the action is executed.
- post-handlers - functions that run *after* the action is executed.

These lists are specified by using argument `:pre-handlers` and `:post-handlers` respectivelly.

These actions are supported:

- `insert` - run in `sp-insert-pair` before/after closing delimiter is inserted
- `wrap` - run after region is wrapped with a pair

Additionally, special actions for various navigation or sexp manipulation commands are added to allow users to do some additional formatting. These actions trigger both for `:pre-handlers` and `:post-handlers`

- `slurp-forward` - run in forward-slurp before/after the closing delimiter is moved
- `slurp-backward` - same, but for backward slurp
- `barf-forward` - run in forward-barf before the closing delimiter is moved
- `barf-backward` - same, but for backward barf.
- `split-sexp` - for `sp-split-sexp`

These actions only run in `:post-handlers`:

- `beginning-of-sexp` - run after `sp-beginning-of-sexp`
- `end-of-sexp` - run after `sp-end-of-sexp`
- `indent-adjust-sexp` - run after `sp-indent-adjust-sexp`
- `dedent-adjust-sexp` - run after `sp-dedent-adjust-sexp`
- `skip-closing-pair` - run after `sp-skip-closing-pair`

- `rewrap-sexp` - run after `sp-rewrap-sexp`

You should always test for the type of action in your hooks and only run the code when apropriate. One can either create one callback with a "dispatch table" that then calls the relevant code, or supply many smaller functions that each test the action separately.

Each handler should be either a lambda, a symbol referring to a named function or an *insertion specification*. If a function, it should take these threes arguments (same as for the *filter* predicates):

- `id` - the id of the pair,
- `action` - the action that triggered the handler,
- `context` - context in which the action was triggered.

You can also provide a special form `(function conditions...)` as a handler. The condition can be either a name of command or a string describing an event (in the format of `single-key-description`). If the last command or the event matches any on the conditions, the hook will be executed. This means these hooks are run not after the insertion, but after the *next* command is executed. **This handler is only executed after a command following an insertion action.**

With these handlers (or hooks, using standard Emacs terminology) you can perform various actions after the autoinsertions or wrappings. For example, a simple function might be used to automatically add newlines and position the point inside a `{}` block in `c-mode`:

```
(defun my-open-block-c-mode (id action context)
  (when (eq action 'insert)
    (newline)
    (newline)
    (indent-according-to-mode)
    (previous-line)
    (indent-according-to-mode)))
```

To add this function to the `{}` pair post-handlers:

```
;; we use :add to keep any global handlers. If you want to replace
;; them, simply specify the "bare list" as an argument:
;; '(my-open-block-c-mode)
(sp-local-pair 'c-mode "{" nil :post-handlers '(:add my-open-block-c-mode))
```

Another useful hook might be to add a space after a pair if it is directly followed by a word or another pair. An example for the `()` in emacs lisp:

```
(sp-local-pair 'emacs-lisp-mode "(" nil :post-handlers '(:add my-add-space-after-sexp-
↪insertion))

(defun my-add-space-after-sexp-insertion (id action _context)
  (when (eq action 'insert)
    (save-excursion
      (forward-char (length (plist-get (sp-get-pair id) :close)))
      (when (or (eq (char-syntax (following-char)) ?w)
                (looking-at (sp--get-opening-regexp)))
        (insert " ")))))
```

Finally, an example of a special delayed action form. This will run the `my-create-newline-and-enter-sexp` function after you've inserted `{}` pair and immediately after hit `RET`.

```
(sp-local-pair 'c++-mode "{" nil :post-handlers '((my-create-newline-and-enter-sexp
↪"RET")))
```

```
(defun my-create-newline-and-enter-sexp (&rest _ignored)
  "Open a new brace or bracket expression, with relevant newlines and indent. "
  (newline)
  (indent-according-to-mode)
  (forward-line -1)
  (indent-according-to-mode))
```

Of course, you can perform tasks of any complexity in these hooks, but it's a good idea to keep them as simple as possible so the "editing flow" won't be disrupted (e.g.: connecting to wikipedia and fetching 100 pages in a post-handler is *not* a good idea :)).

Insertion specification

Because it is very common to insert text inside a pair after this is inserted, smartparens provides a simple specification "language" you can use to make this process easier. Let us first define the language in a semi-formal way, then we will list some examples.

- character `|` means "save-excursion", that is, all actions after this character will be carried out and then the point returns here.
- sequence `||` means the same as above, but after all the instructions are executed, `indent-according-to-mode` is called here as well.
- a square bracket block `[...]` inserts a special directive/command. Right now, these are supported:
  - `i` - call `indent-according-to-mode` at this point.
  - `d#` - call `delete-char` with the specified number (`#` stands for an integer) as argument.
- to insert special characters `|` or `[` put a backslash `\` before it. You don't need to escape `]`.
- all other non-special chatacters are inserted literally.

You can use this in `:pre-handlers` and `:post-handlers` instead of a symbol specifying a function—both in the immediate and delayed hooks.

The specification string is actually translated into a small lisp program which is then evaluated with point inside the newly inserted pair. Here we provide couple examples, for a more comprehensive list you can look into the test suite here (look for test `sp-test-insertion-specification-parser`).

```
"ab" => (progn (insert "ab"))
"a|b" =>
(progn
  (insert "a")
  (save-excursion
    (insert "b")))
"||\n[i]" =>   ;; this is useful as a delayed RET action for {} pair in C-like␣
↪languages
(progn          ;; cf. my-create-newline-and-enter-sexp above
  (save-excursion
    (insert "\n")
    (indent-according-to-mode))
  (indent-according-to-mode))
"* ||\n[i]" =>  ;; pretty-formats /**/ style comments after RET
(progn
  (insert "* ")
  (save-excursion
    (insert "\n")
    (indent-according-to-mode))
```

```
  (indent-according-to-mode))
;; like so
;; /*|*/ =>
;; /*
;;  * |
;;  */
```

Here are some examples replicating some of the above handlers that use function callbacks.

```
(sp-local-pair 'c++-mode "{" nil :post-handlers '((my-create-newline-and-enter-sexp
↪"RET")))
;; =>
(sp-local-pair 'c++-mode "{" nil :post-handlers '(("||\n[i]" "RET")))

;; insert space, remember position, insert space
(sp-local-pair 'emacs-lisp-mode "(" nil :post-handlers '(:add " | "))
;; so that typing `(' results into `( | )', where | is the point.
```

Hooks for wrapping actions

Due to the nature of the wrapping action the pre-handlers are **NOT** executed for this action. If you wish to query for information about the last wrap in the post handler you can use the `sp-last-wrapped-region` variable to do so.

If you change the positions of the opening and closing delimiters (for example by opening new lines or inserting text) you should also update the `sp-last-wrapped-region` variable to reflect these changes, otherwise some functions might not work correctly (repeated wrapping and last wrap deletion for example). See the documentation for this variable for details.

Example configuration

The latest working configuration of the author can be found at GitHub. Before looking at it, have a look at the default-configuration for things already set up for you!

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

E

S