Slimv Tutorial - Part One

Contents

SLIMV stands for ***S**uperior **L**isp **I**nteraction **M**ode for **V**im*. The name comes from SLIME (Superior Lisp Interaction Mode for Emacs), as Slimv tries to provide the same functionality using Vim.

There is an excellent SLIME tutorial movie created by Marco Barringer. The present series of tutorials is trying to introduce the same concepts using Vim and the Slimv plugin through similar actions like Marco did in his 55 minutes long movie.

Downloading

The Slimv plugin can be downloaded from the project page on vim.org. There is also a public Slimv repository containing the latest development builds.

You will also need a Python enabled Vim, and the same Python version installed that is Vim compiled against. This is because the communication part of the plugin is written in Vim's embedded Python (unfortunately there is no embedded Lisp for Vim).

The script is installed the same way as any regular Vim plugin: unzip the contents of the archive in the vimfiles directory.

Configuration

Naturally the configuration of Slimv is different from the setting up of Emacs with SLIME. But if you are lucky, Slimv will just work for you out of the box.

Slimv contains a Swank server (the same that comes with SLIME), but it is also possible to use a separate Swank server of the user's choice. Slimv starts the Swank server in a separate process, so it is recommended to use **`:dont-close t`** in the Swank startup procedure (**`start-swank.lisp`**). Vim does not support asynchronous buffer updates, so the Swank server cannot call Slimv directly, therefore **`*use-dedicated-output-stream*`** must be set to **`nil`**. These are however the default settings in the SLIME embdedded in Slimv.

The choice of Lisp is probably the same as for Emacs: it is recommended to use an implementation that has better support for SLIME. For this tutorial I have chosen SBCL (just like Marco did), but all other major Common Lisp implementations would be OK, and even Clojure or MIT Scheme (I'll write more on these later).

The most important Slimv option is **`g:slimv_swank_cmd`**, but you need it only if there is a problem with the autodetection of Swank and you want Slimv to start the Swank server for you. In this case this option should be set in the .vimrc file to a Vim command that is able to start the Swank server (and return immediately, hence the tailing & in the Linux version).
On Linux this can be something like:

```
let g:slimv_swank_cmd =
    '! xterm -e sbcl --load /usr/share/common-lisp/source/slime/start-swank.lisp &'
```

If there is a difficulty using xterm to open a new terminal window for the Swank server, then I suggest to run Vim inside a GNU screen or tmux session. In this case Slimv will create a new screen/tmux session instead of creating a new window.

Here is Windows example starting Swank in Clozure CL:

```
let g:slimv_swank_cmd = '!start "c:\Program Files\Lisp Cabinet\bin\ccl\wx86cl.exe"
    \ -l "c:\Program Files\Lisp Cabinet\site\lisp\slime\start-swank.lisp" '
```

There are also numerous other Slimv options, most of them are named as **`g:slimv_...`** but these are not covered here, we accept the default values for now.

Starting up

Nothing special here. You just run Vim and open or create a .lisp file. Slimv is automatically loaded, this can be verified by the presence of the "Slimv" menu (and an additional "REPL" menu if you are in the REPL buffer). You can start the Swank server manually, or let Slimv to start it for you. In both cases the server is connected at the first evaluation or when pressing **`,c`** (or selecting Repl/Connect-Server from the Slimv menu). Currently it is not possible to start a remote Swank server via SSH, the server must be on localhost.

The Swank server maintains its state until closed, so even if you exit and restart Vim then reconnect the server, you have all your previous definitions in the REPL.

When the Swank server is connected or a form is evaluated, the REPL buffer is opened in Vim (in a split window by default). If you are using gvim then I recommend you switch on Vim menu (set guioptions+=m) and study the Slimv menu, because it contains most of the Slimv commands together with their keyboard mappings:

```
File  Edit  Tools  Syntax  Buffers  Window  Slimv  REPL  Help
SBCL  Port: 4005  Pid: 2920                 Edit           >
; SWANK 2012-03-06                          Evaluation     > Eval-Defun         ,d
CL-USER>                                    Debugging      > Eval-Current-Exp   ,e
                                            Compilation    > Eval-Region        ,r
                                            Xref           > Eval-Buffer        ,b
                                            Profiling      > Interactive-Eval   ,v
Slimv.REPL.lisp^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Documentation  > Undefine-Function  ,u
                                            Repl           >

morse.lisp                                                   1,0-1          All
Connected to SWANK server on port 4005.
```

This and the following snippets contain both Lisp source code and REPL output. In this tutorial the source code has purple background in order to distinguish it from the REPL buffer. The REPL prompt is yellow here, text following the prompt to the right is manually entered. Explanatory comments (not printed by the REPL) are red and prefixed by "`;;`".

You can switch between the REPL window and the source code window with the **`<Ctrl-w>w`** Normal mode key combination.

Basic REPL operations

Let's switch to the REPL buffer. Here we can see the Lisp prompt and we can type an s-expression just like in any regular REPL. When you press **`Enter`** (in Insert mode) then the form entered is sent to the Swank server for evaluation:

```
CL-USER> (+ 2 2)
4
CL-USER> (make-hash-table)
#<HASH-TABLE :TEST EQL :COUNT 0 {B20CE49}>
```

Note that Paredit Mode is on by default, so each time you insert an opening paren, its closing pair is also added, so the parens are always kept balanced. You can disable Paredit Mode by adding **`let g:paredit_mode=0`** to .vimrc.

When using Paredit Mode, an editor feature called 'electric return' is also enabled by default. When pressing **`Enter`**, an extra newline is inserted, so the line just opened is an empty one. This allows easy linewise editing of sub-forms. The electric returns are then gathered upon pressing **`")"`**. Check this animation to get a hint on how it works. If you want to disable electric returns, just add **`let g:paredit_electric_return=0`** to .vimrc.

You can type `*`, `**`, `***` to get the objects returned by previous evaluations, or **`+`**, **`++`**, **`+++`** to get the forms previously evaluated, just like in SLIME:

```
CL-USER> *
#<HASH-TABLE :TEST EQL :COUNT 0 {B20CE49}>
CL-USER> *
#<HASH-TABLE :TEST EQL :COUNT 0 {B20CE49}>
CL-USER> **
#<HASH-TABLE :TEST EQL :COUNT 0 {B20CE49}>
CL-USER> ***
#<HASH-TABLE :TEST EQL :COUNT 0 {B20CE49}>
CL-USER> (+ 2 2)
4
CL-USER> +
(+ 2 2)
```

The test of presented object lookup Marco did does not work in Slimv, so don't try this at home or the universe will collapse:

```
CL-USER> #<HASH-TABLE :TEST EQL :COUNT 0 {B20CE49}>
;; Should return #<HASH-TABLE :TEST EQL :COUNT 0 {B20CE49}>
;; Instead it drops you in the debugger.
;; More on the debugger in Part Two.
```

Editing a source file

Let's make a new file called morse.lisp and begin typing in the program source code.
Note that I have enabled rainbow parenthesis by adding **`let g:lisp_rainbow=1`** to my .vimrc, so matching parens have the same color:

```
(defpackage :morse
  (:use :common-lisp))

(in-package :morse)
```

Move the mouse cursor over **`defpackage`** and wait for the tooltip. The function description will be displayed in the balloon. Now move the keyboard cursor somewhere in **`defpackage`** then press **`,s`** (or select Documentation/Describe-Symbol from the Slimv menu) and the symbol description will be displayed in the status line and also copied to the REPL buffer.

If you want to look up the symbol in the Common Lisp Hyperspec (CLHS) then move the cursor on the symbol and press **`,h`** (or select Documentation/Hyperspec). This will open the related CLHS page in the default browser.

Now it's time to test our little program. Let's evaluate the first form: place the cursor anywhere in the **`defpackage`** form and press **`,d`** (or select Evaluation/Eval-Defun from the Slimv menu). This will evaluate the current top-level form. Now move the cursor inside the **`in-package`** form and press **`,e`** (or select Evaluation/Eval-Current-Exp). This will evaluate the current s-expression in the REPL buffer:

```
CL-USER>
;; ,d on (defpackage ...) in the source buffer
(defpackage :morse
  (:use :common-lisp))
#<Package "MORSE">
CL-USER>
;; ,d on (in-package ...) in the source buffer
(in-package :morse)
#<Package "MORSE">
```

We can find the package via **`find-package`**. We can also set the current package to **`morse`** by pressing **`,g`** (or selecting Set-Package from the REPL menu) and entering **`morse`**, this will be reflected in the new **`MORSE>`** prompt and in the value of **`*package*`**:

```
CL-USER> (find-package :morse)
#<Package "MORSE">
CL-USER>
;; Set-Package command was used here
MORSE> *package*
#<Package "MORSE">
```

Let's add the morse code mapping function. After typing **`defparameter`** and pressing **`Space`** the function argument list is displayed in the status line. This feature works for all functions defined by the user, not just the built-in ones:

```
(defparameter )


morse.lisp [+]^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^7,15^^^^^^^^^^^Top
(defparameter VAR VAL &OPTIONAL (DOC NIL DOCP))
```

Let's fill in the morse mapping table. Marco googled and copy-pasted the morse table into the source code, then he transformed each line into an s-expression. We can do the same via a Vim macro (**`q`**), I do not cover this topic here. He also auto-indented the code, in Vim we can do that by selecting the form(s) and pressing **`=`**. The indentation is similar in Slimv and in SLIME: **`&body`** arguments in lambda lists and macros are indented by two spaces (and not below the previous argument).

We also define a new function to lookup a character in the morse mapping table:

```
(defparameter *morse-mapping*
  '((#\A ".-")
    (#\B "-...")
    (#\C "-.-.")
    ...
    (#\, "--..--")
    (#\? "..--..")))

(defun character-to-morse (character)
  (assoc character *morse-mapping* :test #'char-equal))
```

Then we realize that we need only the second part of the value returned, so we want to call **`cdr`** on the **`(assoc ...)`** part. Because of Paredit this is not done by entering an opening paren, as this would immediately insert its closing pair next to it. Rather we move the cursor to the opening paren of the form we want to wrap (the one before "assoc") and press **`,W`** or **`,w(`** which wraps the s-expression in a new pair of parentheses. Now we can enter the wrapping function name:

```
(defun character-to-morse (character)
  (cdr (assoc character *morse-mapping* :test #'char-equal)))
```

The inverse for Paredit Wrap is Splice, which removes the outer pair of parens by pressing **`,S`**. It is also possible to Split (**`,O`**) and Join (**`,J`**) s-expressions, Raise (**`,I`**) subforms, or Move parens to the Left (**`,<`**) or Right (**`,>`**).

Now let's compile our morse mapping: press **`,D`** or select Compilation/Compile-Defun from the Slimv menu. We can also compile and load the whole source file. To compile the file press **`,F`** (or select Compilation/Compile-File), to compile and load press **`,L`** (or select Compilation/Compile-Load-File):

```
MORSE>
; compiling (DEFPARAMETER *MORSE-MAPPING* ...)
Compilation finished. (No warnings)  [0.005 secs]

MORSE>
; compiling file "/home/kovisoft/morse.lisp" (written 09 APR 2011 10:00:17 PM):

; /home/kovisoft/morse.fasl written
; compilation finished in 0:00:00.036

Compilation finished. (No warnings)  [0.011 secs]

MORSE>
```

It's time to test our **`character-to-morse`** function. Switch to the REPL buffer and begin typing at the prompt: **`(char`**, then press **`<Tab>`**. You will see a list of possible completions in a popup menu to choose from:

```
MORSE> (char)
        char
        char-code
        char-code-limit
        char-downcase
        char-equal
        char-greaterp
        ...
```

If you type more letters then a subsequent **`<Tab>`** will bring up a subset of the previous completion list. Remember that we are in the **`MORSE`** package, this is why **`character-to-morse`** is included. Repeated pressing of **`<Tab>`** selects (and immediately inserts) the next possible completion.

```
MORSE> (charac)
        character
        characterp
        character-to-morse
        get-macro-character
        set-macro-character
        ...
```

The default method is "fuzzy completion", so you can even type **`ctm`** and press **`Tab`**, then you still get it completed to **`character-to-morse`**.

This kind of completion is not limited to the REPL buffer, it works the same way for the source code buffer, as long as Slimv is connected to the Swank server.

BTW, there is another kind of Vim completion via **`<Ctrl-p>`** or **`<Ctrl-n>`**. That one works by searching for the word with the same prefix in the current buffer up or down. This can be useful for completing words that are not symbol names, e.g. text in comments or strings.

We select the proper completion and finalize the function call:

```
MORSE> (character-to-morse #\a)
(".-")
MORSE> (character-to-morse #\m)
("--")
```

What we got is a list containing a string, but we need a string as the result. We realize that we should use **`second`** instead of **`cdr`** in the function, so we edit the code accordingly and then re-evaluate the defun with **`,d`**.

```
(defun character-to-morse (character)
  (second (assoc character *morse-mapping* :test #'char-equal)))
```

Switch to the REPL buffer again and recall the latest command by pressing **`Up`** in Insert mode, then press **`Enter`** to evaluate it:

```
MORSE>
(defun character-to-morse (character)
  (second (assoc character *morse-mapping* :test #'char-equal)))
STYLE-WARNING: redefining CHARACTER-TO-MORSE in DEFUN
CHARACTER-TO-MORSE
;; Pressing Up to recall last command:
MORSE> (character-to-morse #\m)
"--"
```

Fine, **`character-to-morse`** now returns the morse code string for the character passed as parameter.

Next: Part Two

Written by Tamas Kovacs
Last updated on Aug 28, 2020