SLIME

Superior Lisp Interaction Mode for Emacs

December 11, 2008

Outline

- Slime – An overview
- Slime – How does it work?
- Slime – What can it do?

Slime – An overview

Some initial remarks

- > 5 years old
- > 42k loc
- > 1600 `ChangeLog` entries
- > 100 contributors
  - Code base sometimes feels like a gigantic, mutually shared `.emacs`
  - or IOW, a big ball of mud:
  - no code poetry, but in general quite readable.

Use the source, Luke!

Development

- No release management
  - You're supposed to check out (and update) from CVS.
  - SLIME is mostly hacked during semester breaks, so it's safe to update in Nov/Dez and Jul/Jun.
  - Website terribly out of date.
  - Likewise last official release, Slime-2.0, has a veeery long beard.
- Changes are not necessarily backwards-compatible!
- But if something bugs you, or you miss something, drop a mail to `slime-devel@common-lisp.net`.

Related stuff

Stuff to use in combination with SLIME:

- Taylor Campbell's *paredit* (`http://mumble.net/~campbell/emacs/paredit.el`)
- Michael Weber's *redshank* (`http://www.foldr.org/~michaelw/emacs/redshank/`)

"Alternatives" to SLIME:

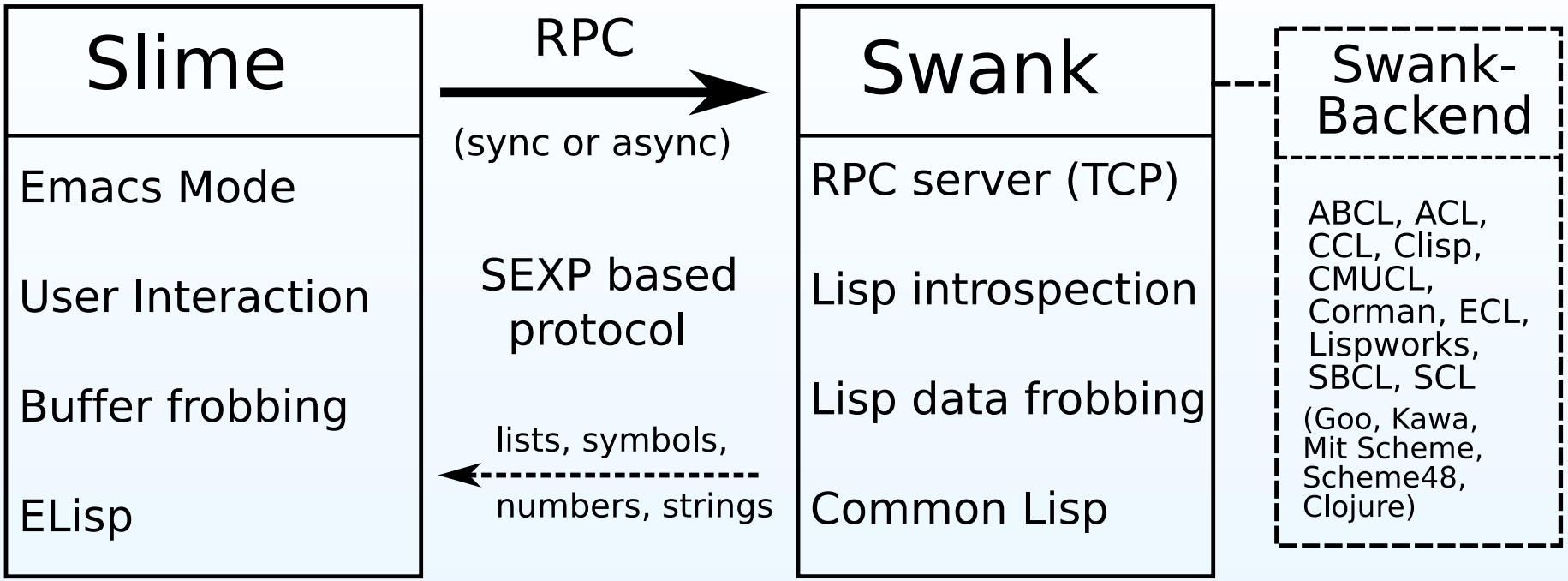
- Symbolics Lisp Machine (`http://www.symbolics.com`)
- *Cusp* for Eclipse (`http://www.bitfauna.com/projects/cusp/`)
- *Limp* for Vim (`http://common-lisp.net/project/limp/`)

.emacs bits


```
;;; A basic slime configuration (2008-12-03)

(add-to-list 'load-path "/home/tcr/.emacs.d/slime/")
(add-to-list 'load-path "/home/tcr/.emacs.d/slime/contrib/")

(require 'slime)

;; Important(!): `slime-fancy' is the meta contrib that will enable
;;               all sorts of advanced features.
(slime-setup '(slime-fancy slime-asdf))

;; ("foo") means that the programm "foo" is tried to be executed.
(setq slime-lisp-implementations
      '((acl70    ("mlisp")     :coding-system utf-8-unix)
        (clisp    ("clisp")     :coding-system utf-8-unix)
        (cmucl    ("lisp")      :coding-system iso-latin-1-unix) ; unicode available soon!
        (sbcl     ("sbcl")      :coding-system utf-8-unix)
        (sbcl-git ("sbcl-git")  :coding-system utf-8-unix)
        (sbcl-cvs-no-utf8 ("sbcl-cvs"))))

;; `M-x slime' will execute this entry in `slime-lisp-implementations'.
;; You can specify another entry to be used by `M-- M-x slime'.
(setf slime-default-lisp 'sbcl-git)
```

Slime – How does it work?

Architecture



- Remote (and reattachable!) sessions possible (securing e.g. via ssh tunneling)
- Multiple simultaneous connections to different implementations possible

Implementation bits

The Swank server is implemented differently depending on what the backend supports. On some implementations, you can even choose which characteristics you want—by tweaking the variable `*COMMUNICATION-STYLE*`:

- `:spawn`
  Using different threads:
  - reader thread (polling)
  - control thread (dispatching)
  - repl thread (executing repl requests)
  - worker threads (executing misc user requests)
- `:fd-handler`
  Using `SERVE-EVENT`: A `select()`-based polling framework to asynchronously invoke callbacks when a polled `FD` becomes available.
- `:sigio`
  Using POSIX signals (`SIGIO`.)
- `nil`
  Busy waiting.

More implementation bits

When browsing `swank.lisp`, watch out for

- `DEFSLIMEFUN`
  Defines a function that can be invoked via an RPC from SLIME:
  - receives data from Emacs, returns data to Emacs (caution: no real automatic marshalling.)
  - function's name automatically exported from `SWANK` package.

The file `swank-backend.lisp` defines lots of function prototypes that are supposed to be implemented by the actual backends in `swank-<foo>.lisp`

- `DEFINTERFACE`
  Defines such a function prototype; a default implementation can be given.
- `DEFIMPLEMENTATION`
  Defines an actual implementation for a function prototype; overwrites default implementation.

A warning for unimplemented prototypes is signalled during startup.

RPC protocol

Packet encoding:

- packet header = length, packet payload = sexp
- sexp format shared between Elisp and Common Lisp:
  - lists
  - symbols (explicitly qualified, except for `T`, `NIL`, and `QUOTE`)
  - numbers (fixnums, single-floats)
  - strings (encoding settings must be isomorphic)

Protocol semantics:

- based on so-called events (lists + keyword as `CAR`):
  - `(:emacs-rex form package thread-id cont-id)`
  - `(:return value cont-id)`
  - `(:emacs-interrupted thread-id)`
  - `(:indentation-update symbol-table)`
  - `(:presentation-start pres-id &optional target)`
  - `(:presentation-end pres-id &optional target)`
  - `...`

An example: C-c C-m

Emacs side

- C-c C-m is `slime-macroexpand-1`:
- grabs string representation of form at point from the buffer
- invokes RPC `swank:swank-macroexpand-1` via `slime-eval-async`
- sends `(:emacs-rex (swank:swank-macroexpand-1 "(foo ...)"))` over the wire

...

An example: C-c C-m

Swank side

- Reader Thread polls on socket...
- receives packet with `(:emacs-rex ...)`, passes it to Control Thread
- Control Thread dispatches on payload `(:emacs-rex ...)`:
- spawns Worker Thread to execute
  `(eval-for-emacs '(swank:swank-macroexpand-1 "(foo ...)"))`
- evaluates `(swank:swank-macroexpand-1 "(foo ...)")`:
- READS the "`(foo ...)`", calls MACROEXPAND on it
- result passed to Control Thread, which sends
  `(:return "(...expansion...)")` back over the wire

...

An example: `C-c C-m`

Emacs side again

- when Emacs goes idle, available input from the inferior Lisp is processed
- dispatches on payload `(:return ...)`
- invokes continuation with received value "`(...expansion...)`" (the continuation was stored by `slime-eval-async` previously.)
- continuation pops up buffer, inserts the expansion, indents and finally fontifies it.

Slime – What can it do?

Foreword

- This talk is based on SLIME checked out on 2008-12-11, and SBCL 1.0.22.
- Make sure that you use the `slime-fancy`, and `slime-asdf` contrib.
- The talk will not be about sexp frobbing. For that, let me refer you to `paredit`'s documentation.

Basics

All Slime operations are perfomed within global *or buffer-local* context:

- current connection

  buffer-local, for example, in Macroexpansion buffers—so further in-place macroexpand operations (see later) are performed within the original connection.

- current thread

  buffer-local for SLDB buffers, as you can interrupt specific threads.

- current package

  in `.lisp` buffers, the buffer's current package is determined by searching backwards for "`(in-package ...)`" forms.

Basics

Modeline indicates the context as follows:

- `Slime[swank, sbcl{n}]`
  Connected to “sbcl” with “swank” being buffer package and “n” pending (non-debugged) requests.

- `Slime[swank, CON:sbcl{n,m}]`
  Like above, but there are additional “m” debugged requests.

- `Slime[sbcl]`
  No buffer package could be determined, so `*PACKAGE*` is implicitly used. Furthermore, there are no pending requests.

- `Slime[swank, sbcl{local}]`
  The connection is buffer local; changing the default connection won’t affect operations in this buffer.

- `Slime[swank, {stale}]`
  A local connection was closed; this buffer thus became useless.

Compilation & Evaluation

Compilation

- `C-c C-c`, compile & load defun at point
- `C-u C-c C-c`, like above but compiled with maximum debug settings
- `C-c C-k`, compile & load buffer
- `C-c M-k`, compile only buffer[1]
- `M-x slime-compile-region`, compile & load region[2]

Evaluation

- `C-M-x`, evaluate defun at point (**re**evaluates `DEFVAR` forms!)

[1] Useful for solely testing compile-time effects.

[2] Useful for block-compiling regions involving `EVAL-WHEN` forms, or inlined functions.

Compilation & Evaluation

Compilation Notes

- in *SLIME Compiler-Notes* buffer
  - RET, jump to offending source location
- in .lisp buffer
  - M-n, go to next note overlay
  - M-p, go to previous note overlay
  - C-c M-c, delete all note overlays from the buffer

Arglist Display

- Arglist display for functions, methods, macros, special-forms.
  - type "`(remove `"
  - type "`(unwind-protect `"
- Arglist display for declare expressions and type specifiers
  - type "`(declare (inline `"
  - type "`(declare (type (simple-array `"
- Contextual arglist display
  - type "`(defmethod `"
  - type "`(defmethod initialize-instance `"
  - type "`(defclass foo () ((a :initarg :a) (b :initarg :b))`", then "`(make-instance 'foo `"

Arglist Display

- `&ANY` is non-existing lambda-list keyword to indicate *optimize qualities* or `EVAL-WHEN` *situations*.
  - type "`(declare (optimize `"
  - type "`(eval-when `"
- Placing point at a variable like `*UNIVERSE*`, or constant like `+GOD+` displays its value.
  - type "`*standard-output*`"
- Arglist displays also works for local functions defined via `FLET`, `LABELS`.
- Likewise for local macros defined via `MACROLET`.

Completion

- Compound-Prefix Completion on TAB at the REPL, on C-c TAB in .lisp buffers.
  - type "(m-v-b<TAB>" at the REPL
  - type "(remove x seq :from<TAB>"
  - type "#\<TAB>"
  - type "(load "/hom<TAB>"
- Fuzzy Completion on C-c M-i
  - type "(mvb<C-c M-i>"
  - type "(sb:with<C-c M-i>"
  - (Use UP, DOWN to navigate, TAB to select an entry, and C-g to quit.)
- C-c C-s inserts remaining arglist onto point
  - type "(eval-when <C-c C-s>"
- C-c C-y inserts call to the defun-at-point into the REPL
  - type "(defun foo (x) <C-c C-y>" in a .lisp buffer

REPL

- M-n, M-p, navigating in input history
- M-r, search backwards in input history
- C-c C-c, to interrupt the inferior Lisp
- C-c M-o, to clear the REPL buffer
- Often used REPL shortcuts:
  - ,!p
    switch package (also ,+p to push, and ,-p to pop)
  - ,load
    load an ASDF system (includes TAB completion!)
  - ,force-load-system
    recompile & load an ASDF system (e.g. to update stale .fasls!)
  - ,sayoonara
    close *all* connections and kill SLIME-related buffers.
  - ,quit
    close current connection only
  - ,restart
    restart the inferior lisp (brave new world!)
  - ,<TAB>

Inspector

Navigation

- TAB, cycle forward through inspectable parts
- S-TAB, cycle backward
- RET, follow link
- l, go backwards
- g, reinspect

Source finding

- ., find source of object-at-point
- M-., find source of symbol-at-point

Misc

- p, pretty-print object-at-point (useful to overcome truncating)
- M-RET, copy representation of object-at-point to the REPL (also works on the “header”, i.e. the object being inspected)

Inspector

FAQ: How to find out about what symbols a package exports?

Answer: Inspect the package!

- `C-c I (find-package :cl) RET`
- click on "`... external symbols`",
- click on "`[Group by classification]`"

Debugger

Navigation

- `n`, next frame
- `p`, previous frame

Frame operations

- `RET` (or `t`), toggle frame details
- `v`, jump to source location of frame-at-point
- `r`, restart from frame-at-point
- `C-c C-c`, recompile location of frame-at-point

Selecting restarts

- `q`, return to toplevel
- `a`, invoke `abort` restart
- `c`, invoke `continue` restart
- `0-9`, invoke *i*th restart

Misc

- `C`, inspect the condition currently being debugged

Debugger

FAQ: How can I get a local variable's value to the REPL, so I can frob it?

Answer:

- If you use the `slime-presentations` contrib (subsumed by `slime-fancy`)
  - use `Mouse-3` (right-click), then select "`Copy to REPL`"
  - (shortcut: `C-c C-v C-r`)
- Alternatively,
  - use `i` to inspect the value,
  - then use `M-RET` on the header representation

Debugger

FAQ: How can I do single-stepping?

Answer:

- functions you want to step through, must be compiled with `(debug 3)`
- invoke `(break)`, or arrange otherwise to land in the debugger
- `s`, to step into
- `x`, to step over (mnemonic: “cross”)
- `o`, to step out

Debugger

Use case: Recompiling frames.

- looking at a frame with too few frame details?
- recompile the frame with maximum debug settings via `C-u C-c C-c`
- now restart computation
  - either by invoking the `retry` restart possibly provided by SLIME,
  - or by restarting from an earlier frame

  (It is hence a good idea to always compile an application's dispatch loop with `(debug 2)` settings such that the function becomes restartable.)

Macroexpanding

In `.lisp` buffers

- `C-c C-m`, macroexpand the form-at-point one single stept
- `C-c M-m`, macroexpand the form-at-point fully (often useful on `LOOP`)

In the resulting `*SLIME Macroexpansion*` buffer

- `g`, redo last macroexpansion
- `C-c C-m`, still usable!
- `C-c M-m`, ditto
- `C-_` or `C-/`, undo last macroexpansion operation

M-. & XREF

In `.lisp` buffers

- `M-.`, goto definition of symbol-at-point
- `M-,`, return from definition (works stack-like)
- `C-M-.`, cycle through multiple available definitions

In an XREF buffer

- `RET`, goto definition-at-point, do not leave XREF buffer
- `SPC`, goto definition-at-point, close XREF buffer
- `n`, next definition
- `C-c C-c`, recompile definition-at-point
- `C-c C-k`, recompile all definitions

N.B.

- When using `M-.` on presentations, a definition of the object-at-point (not the symbol-at-point) is searched for.

M-. & XREF

Xross REFerence:

- `slime-who-binds` (`C-c C-w b`)
- `slime-who-calls` (`C-c C-w c`)
- `slime-calls-who` (`C-c C-w w`)
- `slime-who-macroexpands` (`C-c C-w m`)
- `slime-who-references` (`C-c C-w r`)
- `slime-who-sets` (`C-c C-w s`)
- `slime-who-specializes` (`C-c C-w a`)

M-. & XREF

FAQ: I changed a macro. How do I recompile all functions using the macro?

Answer:

- `C-c C-w C-m` `swank::with-bindings`
- `C-c C-k` in the `*Xref[...]*` buffer

NB: This is only as good as an implementation's `who-macroexpand` is.

Recipe: LOOP indentation

By default, LOOP is indented oddly in Emacs:

```
(loop for x from 0 to 10
    for y from 0 to 10
    collect (cons x y))
```

Place the following snippet in your .emacs to fix that:

```
(setq lisp-simple-loop-indentation 1
      lisp-loop-keyword-indentation 6
      lisp-loop-forms-indentation 6)
```

So we get

```
(loop for x from 0 to 10
      for y from 0 to 10
      collect (cons x y))
```

(Alternatively, you can use the slime-indent contrib which is *not* subsumed by slime-fancy.)

Recipe: Isearch-Yank

In `isearch-mode`, you can use `C-w` to yank the word at point. Unfortunately, the hyphen is a word seperator by default, so `C-w` works ineffectively on Lisp symbols.

But we can arrange that:

```
(defun isearch-yank-symbolic-word-or-char ()
  (interactive)
  (isearch-yank-internal
   (lambda ()
     (let ((distance (skip-syntax-forward "w_")))
       (when (zerop distance) (forward-char 1))
       (point)))))

(add-hook 'lisp-mode-hook
          (lambda ()
            (make-local-variable 'isearch-mode-map)
            (define-key isearch-mode-map "\C-w" 'isearch-yank-symbolic-word-or-char)))
```

Test case:

- place point *on front* of "`*standard-input*`"
- `C-s C-w`.

Recipe: FORMAT ~//

Put the following into ~/.swank.lisp[1]

```
(in-package :swank)

(defun p (stream object &optional colon-p at-sign-p)
  (declare (ignore colon-p at-sign-p))
  (flet ((slime-stream-p (stream)
           (with-struct (connection. user-output user-io trace-output repl-results)
               (default-connection)
             (member stream (list user-output user-io trace-output repl-results)))))
    (assert (slime-stream-p stream))
    (let ((id (and *record-repl-results* (save-presented-object object))))
      (finish-output stream)
      (send-to-emacs `(:presentation-start ,id))
      (send-to-emacs `(:write-string ,(prin1-to-string object)))
      (send-to-emacs `(:presentation-end ,id))
      (finish-output stream))))
```

Test case:

- Evaluate "`(format t "The current readtable is ~/swank::p/." *readtable*)`"

[1] User customization file which is executed once Swank is loaded.

Recipe: SLDB makeup

The look of SLDB is highly customizable[2]

- search for "`define-sldb-face`" in `slime.el` for all available faces.
- recently introduced faces:
  - `sldb-restartable-frame-line-face`
  - `sldb-non-restartable-frame-line-face`

[2] The look of the inspector is also customizable, search for "`slime-inspector-.*-faces`".

Tip: Renaming identifiers

- For functions, macros, global variables, make use of XREF's `who-foo`!
  In particular, use `C-M-.` + keyboard macro.

- For renaming a local binding
  - get in front of the toplevel form (`C-M-a`)
  - select the toplevel form (`C-SPC C-M-SPC`)
  - use `query-replace` on the region (`M-x query-replace`)

&c

There is *still* stuff that was not mentioned:

- profiler
- trace
- apropos
- disassemble
- setting up remote connections
- anything that I can't remember of..