

swank.lisp

Numerous authors

March 6, 2005

Contents

1	swank.lisp	2
2	Top-level variables, constants, macros	3
3	Hooks	5
4	Connections	5
5	Helper macros	7
6	TCP Server	9
	6.0.1 Thread based communication	12
	6.0.2 Signal driven IO	14
	6.0.3 SERVE-EVENT based IO	15
	6.0.4 Simple sequential IO	15
7	IO to Emacs	17
	7.1 Global I/O redirection framework	17
	7.2 Global redirection setup	18
	7.3 Global redirection hooks	19
	7.4 Redirection during requests	20
8	Reading and printing	23
9	Arglists	25
10	Evaluation	34
11	Debugger	38
	11.1 Debugger loop	38
	11.2 SLDB entry points	40
12	Compilation Commands.	42
13	Loading	44
14	Macroexpansion	45

15 Basic completion	45
15.1 Find completion set	46
15.2 Format completion results	47
15.3 Compound-prefix matching	48
15.4 Extending the input string by completion	48
15.5 Completion Tests	49
16 Fuzzy completion	50
16.1 Fuzzy completion core	53
16.2 Fuzzy completion scoring	55
17 Documentation	57
18 Package Commands	59
19 Tracing	60
20 Undefing	60
21 Profiling	60
22 Source Locations	60
23 Inspecting	62
24 Thread listing	75
25 Class browser	76
26 Automatically synchronized state	76
26.1 *FEATURES*	77
26.2 Indentation of macros	77

1 swank.lisp

This file defines the “Swank” TCP server for Emacs to talk to. The code in this file is purely portable Common Lisp. We do require a smattering of non-portable functions in order to write the server, so we have defined them in `swank-backend.lisp` and implemented them separately for each Lisp implementation. These extensions are available to us here via the `SWANK-BACKEND` package.

```
(defpackage :swank
  (:use :common-lisp :swank-backend)
  (:export #:startup-multiprocessing
           #:start-server
           #:create-swank-server
           #:create-server
           #:ed-in-emacs
           #:print-indentation-lossage
           #:swank-debugger-hook
```

```

;; These are user-configurable variables:
#: *communication-style*
#: *log-events*
#: *log-output*
#: *use-dedicated-output-stream*
#: *configure-emacs-indentation*
#: *readtable-alist*
#: *globally-redirect-io*
#: *global-debugger*
#: *sldb-printer-bindings*
#: *swank-pprint-bindings*
#: *default-worker-thread-bindings*
;; These are re-exported directly from the backend:
#:buffer-first-change
#:frame-source-location-for-emacs
#:restart-frame
#:sldb-step
#:sldb-break
#:sldb-break-on-return
#:profiled-functions
#:profile-report
#:profile-reset
#:unprofile-all
#:profile-package
#:default-directory
#:set-default-directory
#:quit-lisp))

```

```
(in-package :swank)
```

2 Top-level variables, constants, macros

```
(defconstant cl-package (find-package :cl)
  "The COMMON-LISP package.")
```

```
(defconstant keyword-package (find-package :keyword)
  "The KEYWORD package.")
```

```
(defvar *canonical-package-nicknames*
  '(("COMMON-LISP-USER" . "CL-USER"))
  "Canonical package names to use instead of shortest name/nickname.")
```

```
(defvar *auto-abbreviate-dotted-packages* t
  "Automatically abbreviate dotted package names to their last component when T")
```

```
(defvar *swank-io-package*
  (let ((package (make-package :swank-io-package :use '())))
    (import '(nil t quote) package)
    package))
```

```

(defconstant default-server-port 4005
  "The default TCP port for the server (when started manually).")

(defvar *swank-debug-p* t
  "When true, print extra debugging information.")

(defvar *sldb-printer-bindings*
  `((*print-pretty*          . nil)
    (*print-level*          . 4)
    (*print-length*         . 10)
    (*print-circle*         . t)
    (*print-readably*       . nil)
    (*print-pprint-dispatch* . ,(copy-pprint-dispatch nil))
    (*print-gensym*         . t)
    (*print-base*           . 10)
    (*print-radix*          . nil)
    (*print-array*          . t)
    (*print-lines*         . 200)
    (*print-escape*         . t))
  "A set of printer variables used in the debugger.")

(defvar *swank-pprint-bindings*
  `((*print-level*          . nil)
    (*print-length*         . nil)
    (*print-circle*         . t)
    (*print-gensym*         . t)
    (*print-readably*       . nil)
    (*print-pprint-dispatch* . ,(copy-pprint-dispatch nil)))
  "A list of variables bindings during pretty printing.
Used when printing macroexpansions and pprint-eval.")

(defvar *default-worker-thread-bindings* '()
  "An alist to initialize dynamic variables in worker threads.
The list has the form ((VAR . VALUE) ...). Each variable VAR will be
bound to the corresponding VALUE.")

(defun call-with-bindings (alist fun)
  "Call FUN with variables bound according to ALIST.
ALIST is a list of the form ((VAR . VAL) ...)."
  (let ((vars (mapcar #'car alist))
        (vals (mapcar #'cdr alist)))
    (progv vars vals
      (funcall fun))))

The DEFSLIMEFUN macro defines a function that Emacs can call via RPC.

(defmacro defslimefun (name arglist &body rest)
  "A DEFUN for functions that Emacs can call by RPC."
  `(progn

```

```

      (defun ,name ,arglist ,@rest)
      ;; see <http://www.franz.com/support/documentation/6.2/doc/pages/variables
      (eval-when (:compile-toplevel :load-toplevel :execute)
        (export ',name :swank))))

(declare (ftype (function () nil) missing-arg))
(defun missing-arg ()
  "A function that the compiler knows will never to return a value.
You can use (MISSING-ARG) as the initform for defstruct slots that
must always be supplied. This way the :TYPE slot option need not
include some arbitrary initial value like NIL."
  (error "A required &KEY or &OPTIONAL argument was not supplied."))

```

3 Hooks

We use Emacs-like add-hook and run-hook utilities to support simple indirection. The interface is more CLish than the Emacs Lisp one.

```

(defmacro add-hook (place function)
  "Add FUNCTION to the list of values on PLACE."
  `(pushnew ,function ,place))

(defun run-hook (functions &rest arguments)
  "Call each of FUNCTIONS with ARGUMENTS."
  (dolist (function functions)
    (apply function arguments)))

(defvar *new-connection-hook* '()
  "This hook is run each time a connection is established.
The connection structure is given as the argument.
Backend code should treat the connection structure as opaque.")

(defvar *connection-closed-hook* '()
  "This hook is run when a connection is closed.
The connection as passed as an argument.
Backend code should treat the connection structure as opaque.")

(defvar *pre-reply-hook* '()
  "Hook run (without arguments) immediately before replying to an RPC.")

```

4 Connections

Connection structures represent the network connections between Emacs and Lisp. Each has a socket stream, a set of user I/O streams that redirect to Emacs, and optionally a second socket used solely to pipe user-output to Emacs (an optimization).

```

(defvar *coding-system* ':iso-latin-1-unix)

```

```

(defstruct (connection
           (:conc-name connection.)
           (:print-function print-connection))
  ;; Raw I/O stream of socket connection.
  (socket-io (missing-arg) :type stream :read-only t)
  ;; Optional dedicated output socket (backending 'user-output' slot).
  ;; Has a slot so that it can be closed with the connection.
  (dedicated-output nil :type (or stream null))
  ;; Streams that can be used for user interaction, with requests
  ;; redirected to Emacs.
  (user-input nil :type (or stream null))
  (user-output nil :type (or stream null))
  (user-io nil :type (or stream null))
  ;; In multithreaded systems we delegate certain tasks to specific
  ;; threads. The 'reader-thread' is responsible for reading network
  ;; requests from Emacs and sending them to the 'control-thread'; the
  ;; 'control-thread' is responsible for dispatching requests to the
  ;; threads that should handle them; the 'repl-thread' is the one
  ;; that evaluates REPL expressions. The control thread dispatches
  ;; all REPL evaluations to the REPL thread and for other requests it
  ;; spawns new threads.
  reader-thread
  control-thread
  repl-thread
  ;; Callback functions:
  ;; (SERVE-REQUESTS <this-connection>) serves all pending requests
  ;; from Emacs.
  (serve-requests (missing-arg) :type function)
  ;; (READ) is called to read and return one message from Emacs.
  (read (missing-arg) :type function)
  ;; (SEND OBJECT) is called to send one message to Emacs.
  (send (missing-arg) :type function)
  ;; (CLEANUP <this-connection>) is called when the connection is
  ;; closed.
  (cleanup nil :type (or null function))
  ;; Cache of macro-indentation information that has been sent to Emacs.
  ;; This is used for preparing deltas to update Emacs's knowledge.
  ;; Maps: symbol -> indentation-specification
  (indentation-cache (make-hash-table :test 'eq) :type hash-table)
  ;; The list of packages represented in the cache:
  (indentation-cache-packages '())
  ;; The communication style used.
  (communication-style nil :type (member nil :spawn :sigio :fd-handler))
  ;; The coding system for network streams.
  (external-format *coding-system* :type (member :iso-latin-1-unix
                                                  :emacs-mule-unix
                                                  :utf-8-unix)))

(defun print-connection (conn stream depth)
  (declare (ignore depth))

```

```

(print-unreadable-object (conn stream :type t :identity t)))

(defvar *connections* '()
  "List of all active connections, with the most recent at the front.")

(defvar *emacs-connection* nil
  "The connection to Emacs currently in use.")

(defvar *swank-state-stack* '()
  "A list of symbols describing the current state. Used for debugging
and to detect situations where interrupts can be ignored.")

(defun default-connection ()
  "Return the 'default' Emacs connection.
This connection can be used to talk with Emacs when no specific
connection is in use, i.e. *EMACS-CONNECTION* is NIL.

The default connection is defined (quite arbitrarily) as the most
recently established one."
  (first *connections*))

(defslimefun state-stack ()
  "Return the value of *SWANK-STATE-STACK*."
  *swank-state-stack*)

(define-condition slime-protocol-error (error)
  ((condition :initarg :condition :reader slime-protocol-error.condition))
  (:report (lambda (condition stream)
             (format stream "~A" (slime-protocol-error.condition condition)))))

(add-hook *new-connection-hook* 'notify-backend-of-connection)
(defun notify-backend-of-connection (connection)
  (declare (ignore connection))
  (emacs-connected))

```

5 Helper macros

```

(defmacro with-io-redirection ((connection) &body body)
  "Execute BODY I/O redirection to CONNECTION.
If *REDIRECT-IO* is true then all standard I/O streams are redirected."
  `(if *redirect-io*
      (call-with-redirected-io ,connection (lambda () ,@body))
      (progn ,@body)))

(defmacro with-connection ((connection) &body body)
  "Execute BODY in the context of CONNECTION."
  `(let ((*emacs-connection* ,connection))
      (catch 'slime-toplevel
          (with-io-redirection (*emacs-connection*))

```

```

        (let ((*debugger-hook* #'swank-debugger-hook))
            ,@body))))))

(defmacro without-interrupts (&body body)
  `(call-without-interrupts (lambda () ,@body)))

(defmacro destructure-case (value &rest patterns)
  "Dispatch VALUE to one of PATTERNS.
  A cross between 'case' and 'destructuring-bind'.
  The pattern syntax is:
  ((HEAD . ARGS) . BODY)
  The list of patterns is searched for a HEAD 'eq' to the car of
  VALUE. If one is found, the BODY is executed with ARGS bound to the
  corresponding values in the CDR of VALUE."
  (let ((operator (gensym "op-"))
        (operands (gensym "rand-"))
        (tmp (gensym "tmp-")))
    `(let* ((,tmp ,value)
           (,operator (car ,tmp))
           (,operands (cdr ,tmp)))
      (case ,operator
        ,@(loop for (pattern . body) in patterns collect
              (if (eq pattern t)
                  `(t ,@body)
                  (destructuring-bind (op &rest rands) pattern
                    `(,op (destructuring-bind ,rands ,operands
                        ,@body))))))
        ,@(if (eq (caar (last patterns)) t)
              '()
              `(t (error "destructure-case failed: ~S" ,tmp)))))))

(defmacro with-temp-package (var &body body)
  "Execute BODY with VAR bound to a temporary package.
  The package is deleted before returning."
  `(let ((,var (make-package (gensym "TEMP-PACKAGE-"))))
    (unwind-protect (progn ,@body)
      (delete-package ,var))))

(defvar *log-events* nil)
(defvar *log-output* *error-output*)

(defun log-event (format-string &rest args)
  "Write a message to *terminal-io* when *log-events* is non-nil.
  Useful for low level debugging."
  (when *log-events*
    (apply #'format *log-output* format-string args)
    (force-output *log-output*)))

```

6 TCP Server

```
(defparameter *redirect-io* t
  "When non-nil redirect Lisp standard I/O to Emacs.
  Redirection is done while Lisp is processing a request for Emacs.")

(defvar *use-dedicated-output-stream* t)
(defvar *communication-style* (preferred-communication-style))

(defun start-server (port-file &key (style *communication-style*)
                   dont-close (external-format *coding-system*))
  "Start the server and write the listen port number to PORT-FILE.
  This is the entry point for Emacs."
  (setup-server 0 (lambda (port) (announce-server-port port-file port))
               style dont-close external-format))

(defun create-server (&key (port default-server-port)
                      (style *communication-style*)
                      dont-close (external-format *coding-system*))
  "Start a SWANK server on PORT running in STYLE.
  If DONT-CLOSE is true then the listen socket will accept multiple
  connections, otherwise it will be closed after the first."
  (setup-server port #'simple-announce-function style dont-close
               external-format))

(defun create-swank-server (&optional (port default-server-port)
                            (style *communication-style*)
                            (announce-fn #'simple-announce-function)
                            dont-close (external-format *coding-system*))
  (setup-server port announce-fn style dont-close external-format))

(defparameter *loopback-interface* "127.0.0.1")

(defun setup-server (port announce-fn style dont-close external-format)
  (declare (type function announce-fn))
  (let* ((socket (create-socket *loopback-interface* port))
        (port (local-port socket)))
    (funcall announce-fn port)
    (flet ((serve ())
            (serve-connection socket style dont-close external-format)))
      (ecase style
        (:spawn
         (spawn (lambda () (loop do (serve) while dont-close))
                :name "Swank"))
        (:fd-handler :sigio)
        (add-fd-handler socket (lambda () (serve))))
      (nil)
      (unwind-protect (loop do (serve) while dont-close)
        (close-socket socket))))
    port)))
```

```

(defun serve-connection (socket style dont-close external-format)
  (let ((client (accept-connection socket :external-format external-format)))
    (unless dont-close
      (close-socket socket))
    (let ((connection (create-connection client style external-format)))
      (run-hook *new-connection-hook* connection)
      (push connection *connections*)
      (serve-requests connection))))))

(defun serve-requests (connection)
  "Read and process all requests on connections."
  (funcall (connection.serve-requests connection) connection))

(defun announce-server-port (file port)
  (with-open-file (s file
                   :direction :output
                   :if-exists :overwrite
                   :if-does-not-exist :create)
    (format s "~S~%" port))
  (simple-announce-function port))

(defun simple-announce-function (port)
  (when *swank-debug-p*
    (format *debug-io* "~&; Swank started at port: ~D.~%" port)))

(defun open-streams (connection)
  "Return the 4 streams for IO redirection:
DEDICATED-OUTPUT INPUT OUTPUT IO"
  (multiple-value-bind (output-fn dedicated-output)
    (make-output-function connection)
    (let ((input-fn
           (lambda ()
             (with-connection (connection)
               (with-simple-restart (abort-read
                                     "Abort reading input from Emacs.")
                 (read-user-input-from-emacs))))))
      (multiple-value-bind (in out) (make-fn-streams input-fn output-fn)
        (let ((out (or dedicated-output out))
              (io (make-two-way-stream in out)))
          (mapc #'make-stream-interactive (list in out io))
          (values dedicated-output in out io))))))

(defun make-output-function (connection)
  "Create function to send user output to Emacs.
This function may open a dedicated socket to send output. It
returns two values: the output function, and the dedicated
stream (or NIL if none was created)."
  (if *use-dedicated-output-stream*
      (let ((stream (open-dedicated-output-stream

```

```

                (connection.socket-io connection)
                (connection.external-format connection))))
(values (lambda (string)
          (write-string string stream)
          (force-output stream))
        stream))
(values (lambda (string)
          (with-connection (connection)
            (with-simple-restart
              (abort "Abort sending output to Emacs.")
              (send-to-emacs `(:read-output ,string))))))
        nil)))

(defun open-dedicated-output-stream (socket-io external-format)
  "Open a dedicated output connection to the Emacs on SOCKET-IO.
Return an output stream suitable for writing program output.

This is an optimized way for Lisp to deliver output to Emacs."
  (let* ((socket (create-socket *loopback-interface* 0))
         (port (local-port socket)))
    (encode-message `(:open-dedicated-output-stream ,port) socket-io)
    (accept-connection socket :external-format external-format)))

(defun handle-request (connection)
  "Read and process one request. The processing is done in the extend
of the toplevel restart."
  (assert (null *swank-state-stack*))
  (let ((*swank-state-stack* `(:handle-request))
        (*debugger-hook* nil))
    (with-connection (connection)
      (with-simple-restart (abort "Abort handling SLIME request.")
        (read-from-emacs))))))

(defun current-socket-io ()
  (connection.socket-io *emacs-connection*))

(defun close-connection (c &optional condition)
  (let ((cleanup (connection.cleanup c)))
    (when cleanup
      (funcall cleanup c)))
    (close (connection.socket-io c))
    (when (connection.dedicated-output c)
      (close (connection.dedicated-output c)))
    (setf *connections* (remove c *connections*))
    (run-hook *connection-closed-hook* c)
    (when condition
      (format *debug-io* "~&; Connection to Emacs lost.~&; [~A]~%" condition)
      (finish-output *debug-io*)))

(defun with-reader-error-handler ((connection) &body body)

```

```

  `(handler-case (progn ,@body)
    (slime-protocol-error (e)
      (close-connection ,connection e))))

(defun simple-break ()
  (with-simple-restart (continue "Continue from interrupt.")
    (let ((*debugger-hook* #'swank-debugger-hook))
      (invoke-debugger
       (make-condition 'simple-error
                       :format-control "Interrupt from Emacs")))))

```

6.0.1 Thread based communication

```

(defvar *active-threads* '())

(defun read-loop (control-thread input-stream connection)
  (with-reader-error-handler (connection)
    (loop (send control-thread (decode-message input-stream)))))

(defun dispatch-loop (socket-io connection)
  (let ((*emacs-connection* connection))
    (handler-case
      (loop (dispatch-event (receive) socket-io))
      (error (e)
        (close-connection connection e)))))

(defun repl-thread (connection)
  (let ((thread (connection.repl-thread connection)))
    (if (thread-alive-p thread)
        thread
        (setf (connection.repl-thread connection)
              (spawn (lambda () (repl-loop connection))
                     :name "new-repl-thread")))))

(defun find-worker-thread (id)
  (etypecase id
    ((member t)
     (car *active-threads*))
    ((member :repl-thread)
     (repl-thread *emacs-connection*))
    (fixnum
     (find-thread id))))

(defun interrupt-worker-thread (id)
  (let ((thread (or (find-worker-thread id)
                   (repl-thread *emacs-connection*))))
    (interrupt-thread thread #'simple-break)))

(defun thread-for-evaluation (id)
  "Find or create a thread to evaluate the next request."

```

```

(let ((c *emacs-connection*))
  (etypecase id
    ((member t)
     (spawn-worker-thread c))
    ((member :repl-thread)
     (repl-thread c))
    (fixnum
     (find-thread id))))))

(defun spawn-worker-thread (connection)
  (spawn (lambda ()
           (call-with-bindings *default-worker-thread-bindings*
                               (lambda ()
                                 (handle-request connection))))
         :name "worker"))

(defun dispatch-event (event socket-io)
  "Handle an event triggered either by Emacs or within Lisp."
  (log-event "DISPATCHING: ~S~%" event)
  (destructure-case event
    ((:emacs-rex form package thread-id id)
     (let ((thread (thread-for-evaluation thread-id)))
       (push thread *active-threads*)
       (send thread `(eval-for-emacs ,form ,package ,id))))
    ((:return thread &rest args)
     (let ((tail (member thread *active-threads*)))
       (setq *active-threads* (nconc (ldiff *active-threads* tail)
                                     (cdr tail))))
      (encode-message `(:return ,@args) socket-io))
    ((:emacs-interrupt thread-id)
     (interrupt-worker-thread thread-id))
    (((:debug :debug-condition :debug-activate :debug-return)
      thread &rest args)
     (encode-message `((, (car event) ,(thread-id thread) ,@args) socket-io))
    ((:read-string thread &rest args)
     (encode-message `(:read-string ,(thread-id thread) ,@args) socket-io))
    ((:evaluate-in-emacs string thread &rest args)
     (encode-message `(:evaluate-in-emacs ,string ,(thread-id thread) ,@args)
                     socket-io))
    ((:read-aborted thread &rest args)
     (encode-message `(:read-aborted ,(thread-id thread) ,@args) socket-io))
    ((:emacs-return-string thread-id tag string)
     (send (find-thread thread-id) `(take-input ,tag ,string)))
    ((:eval thread &rest args)
     (encode-message `(:eval ,(thread-id thread) ,@args) socket-io))
    ((:emacs-return thread-id tag value)
     (send (find-thread thread-id) `(take-input ,tag ,value)))
    (((:read-output :new-package :new-features :ed :%apply :indentation-update
                    :eval-no-wait)
      &rest _)
     ))

```

```

(declare (ignore _))
(encode-message event socket-io))))

(defun spawn-threads-for-connection (connection)
  (let* ((socket-io (connection.socket-io connection))
        (control-thread (spawn (lambda ()
                                (let ((*debugger-hook* nil))
                                  (dispatch-loop socket-io connection))
                                :name "control-thread"))))
    (setf (connection.control-thread connection) control-thread)
    (let ((reader-thread (spawn (lambda ()
                                (let ((*debugger-hook* nil))
                                  (read-loop control-thread socket-io
                                             connection))
                                :name "reader-thread"))))
      (repl-thread (spawn (lambda () (repl-loop connection))
                       :name "repl-thread")))
      (setf (connection.reader-thread connection) reader-thread)
      (setf (connection.repl-thread connection) repl-thread)
      connection)))

(defun cleanup-connection-threads (connection)
  (let ((threads (list (connection.repl-thread connection)
                      (connection.reader-thread connection)
                      (connection.control-thread connection))))
    (dolist (thread threads)
      (unless (equal (current-thread) thread)
        (kill-thread thread)))))

(defun repl-loop (connection)
  (with-connection (connection)
    (loop (handle-request connection))))

(defun process-available-input (stream fn)
  (loop while (and (open-stream-p stream)
                  (listen stream))
    do (funcall fn)))

```

6.0.2 Signal driven IO

```

(defun install-sigio-handler (connection)
  (let ((client (connection.socket-io connection))
        (flet ((handler ()
                (cond ((null *swank-state-stack*)
                      (with-reader-error-handler (connection)
                        (process-available-input
                         client (lambda () (handle-request connection))))))
                  ((eq (car *swank-state-stack*) :read-next-form))
                  (t (process-available-input client #'read-from-emacs))))))
    (add-sigio-handler client #'handler)

```

```

(handler))))
(defun deinstall-sigio-handler (connection)
  (remove-sigio-handlers (connection.socket-io connection)))

```

6.0.3 SERVE-EVENT based IO

```

(defun install-fd-handler (connection)
  (let ((client (connection.socket-io connection)))
    (flet ((handler ()
            (cond ((null *swank-state-stack*)
                  (with-reader-error-handler (connection)
                    (process-available-input
                     client (lambda () (handle-request connection))))))
              ((eq (car *swank-state-stack*) :read-next-form)
               (t (process-available-input client #'read-from-emacs))))))
      (setq *debugger-hook*
            (lambda (c h)
              (with-reader-error-handler (connection)
                (block debugger
                 (with-connection (connection)
                  (swank-debugger-hook c h)
                  (return-from debugger))
                 (abort))))))
      (add-fd-handler client #'handler)
      (handler))))

(defun deinstall-fd-handler (connection)
  (remove-fd-handlers (connection.socket-io connection)))

```

6.0.4 Simple sequential IO

```

(defun simple-serve-requests (connection)
  (with-reader-error-handler (connection)
    (loop (handle-request connection))))

(defun read-from-socket-io ()
  (let ((event (decode-message (current-socket-io))))
    (log-event "DISPATCHING: ~S~%" event)
    (destructure-case event
      ((:emacs-rex form package thread id)
       (declare (ignore thread))
       `(eval-for-emacs ,form ,package ,id))
      ((:emacs-interrupt thread)
       (declare (ignore thread))
       '(simple-break))
      ((:emacs-return-string thread tag string)
       (declare (ignore thread))
       `(take-input ,tag ,string))
      ((:emacs-return thread tag value)

```

```

        (declare (ignore thread))
        `(take-input ,tag ,value))))))

(defun send-to-socket-io (event)
  (log-event "DISPATCHING: ~S~%" event)
  (flet ((send (o)
          (without-interrupts
            (encode-message o (current-socket-io))))))
    (destructure-case event
      ((:debug-activate :debug :debug-return :read-string :read-aborted
        :eval)
       thread &rest args)
      (declare (ignore thread))
      (send `((, (car event) 0 ,@args)))
      (:return thread &rest args)
      (declare (ignore thread))
      (send `(:return ,@args)))
      ((:read-output :new-package :new-features :debug-condition
        :indentation-update :ed :%apply :eval-no-wait)
       &rest _)
      (declare (ignore _))
      (send event))))))

(defun initialize-streams-for-connection (connection)
  (multiple-value-bind (dedicated in out io) (open-streams connection)
    (setf (connection.dedicated-output connection) dedicated
          (connection.user-io connection) io
          (connection.user-output connection) out
          (connection.user-input connection) in)
    connection))

(defun create-connection (socket-io style external-format)
  (let ((c (ecase style
             (:spawn
              (make-connection :socket-io socket-io
                              :read #'read-from-control-thread
                              :send #'send-to-control-thread
                              :serve-requests #'spawn-threads-for-connection
                              :cleanup #'cleanup-connection-threads))
             (:sigio
              (make-connection :socket-io socket-io
                              :read #'read-from-socket-io
                              :send #'send-to-socket-io
                              :serve-requests #'install-sigio-handler
                              :cleanup #'deinstall-sigio-handler))
             (:fd-handler
              (make-connection :socket-io socket-io
                              :read #'read-from-socket-io
                              :send #'send-to-socket-io
                              :serve-requests #'install-fd-handler

```

```

:cleanup #'deinstall-fd-handler))
((nil)
 (make-connection :socket-io socket-io
                  :read #'read-from-socket-io
                  :send #'send-to-socket-io
                  :serve-requests #'simple-serve-requests))))
(setf (connection.communication-style c) style)
(setf (connection.external-format c) external-format)
(initialize-streams-for-connection c)
c))

```

7 IO to Emacs

This code handles redirection of the standard I/O streams (`*standard-output*`, etc) into Emacs. The `connection` structure contains the appropriate streams, so all we have to do is make the right bindings.

7.1 Global I/O redirection framework

Optionally, the top-level global bindings of the standard streams can be assigned to be redirected to Emacs. When Emacs connects we redirect the streams into the connection, and they keep going into that connection even if more are established. If the connection handling the streams closes then another is chosen, or if there are no connections then we revert to the original (real) streams.

It is slightly tricky to assign the global values of standard streams because they are often shadowed by dynamic bindings. We solve this problem by introducing an extra indirection via synonym streams, so that `*STANDARD-INPUT*` is a synonym stream to `*CURRENT-STANDARD-INPUT*`, etc. We never shadow the “current” variables, so they can always be assigned to affect a global change.

```

(defvar *globally-redirect-io* nil
  "When non-nil globally redirect all standard streams to Emacs.")

(defmacro setup-stream-indirection (stream-var)
  "Setup redirection scaffolding for a global stream variable.
Supposing (for example) STREAM-VAR is *STANDARD-INPUT*, this macro:

```

1. Saves the value of `*STANDARD-INPUT*` in a variable called `*REAL-STANDARD-INPUT*`.
2. Creates `*CURRENT-STANDARD-INPUT*`, initially with the same value as `*STANDARD-INPUT*`.
3. Assigns `*STANDARD-INPUT*` to a synonym stream pointing to `*CURRENT-STANDARD-INPUT*`.

This has the effect of making `*CURRENT-STANDARD-INPUT*` contain the effective global value for `*STANDARD-INPUT*`. This way we can assign

the effective global value even when `*STANDARD-INPUT*` is shadowed by a dynamic binding."

```
(let ((real-stream-var (prefixed-var "REAL" stream-var))
      (current-stream-var (prefixed-var "CURRENT" stream-var)))
  `(progn
    ;; Save the real stream value for the future.
    (defvar ,real-stream-var ,stream-var)
    ;; Define a new variable for the effective stream.
    ;; This can be reassigned.
    (defvar ,current-stream-var ,stream-var)
    ;; Assign the real binding as a synonym for the current one.
    (setq ,stream-var (make-synonym-stream ',current-stream-var))))))

(eval-when (:compile-toplevel :load-toplevel :execute)
  (defun prefixed-var (prefix variable-symbol)
    "(PREFIXED-VAR \"FOO\" '*BAR*) => SWANK::*FOO-BAR*"
    (let ((basename (subseq (symbol-name variable-symbol) 1)))
      (intern (format nil "~A-~A" prefix basename) :swank))))
```

7.2 Global redirection setup

```
(setup-stream-indirection *standard-output*)
(setup-stream-indirection *error-output*)
(setup-stream-indirection *trace-output*)
(setup-stream-indirection *standard-input*)
(setup-stream-indirection *debug-io*)
(setup-stream-indirection *query-io*)
(setup-stream-indirection *terminal-io*)

(defparameter *standard-output-streams*
  '(*standard-output* *error-output* *trace-output*)
  "The symbols naming standard output streams.")

(defparameter *standard-input-streams*
  '(*standard-input*)
  "The symbols naming standard input streams.")

(defparameter *standard-io-streams*
  '(*debug-io* *query-io* *terminal-io*)
  "The symbols naming standard io streams.")

(defun globally-redirect-io-to-connection (connection)
  "Set the standard I/O streams to redirect to CONNECTION.
Assigns *CURRENT-* for all standard streams."
  (dolist (o *standard-output-streams*)
    (set (prefixed-var "CURRENT" o)
         (connection.user-output connection)))
  ;; FIXME: If we redirect standard input to Emacs then we get the
  ;; regular Lisp top-level trying to read from our REPL.
  ;;
```

```

;; Perhaps the ideal would be for the real top-level to run in a
;; thread with local bindings for all the standard streams. Failing
;; that we probably would like to inhibit it from reading while
;; Emacs is connected.
;;
;; Meanwhile we just leave *standard-input* alone.
#+NIL
(dolist (i *standard-input-streams*)
  (set (prefixed-var "CURRENT" i)
       (connection.user-input connection)))
(dolist (io *standard-io-streams*)
  (set (prefixed-var "CURRENT" io)
       (connection.user-io connection)))

(defun revert-global-io-redirection ()
  "Set *CURRENT-* to *REAL-* for all standard streams."
  (dolist (stream-var (append *standard-output-streams*
                              *standard-input-streams*
                              *standard-io-streams*))
    (set (prefixed-var "CURRENT" stream-var)
         (symbol-value (prefixed-var "REAL" stream-var)))))

```

7.3 Global redirection hooks

```

(defvar *global-stdio-connection* nil
  "The connection to which standard I/O streams are globally redirected.
NIL if streams are not globally redirected.")

(defun maybe-redirect-global-io (connection)
  "Consider globally redirecting to a newly-established CONNECTION."
  (when (and *globally-redirect-io* (null *global-stdio-connection*))
    (setq *global-stdio-connection* connection)
    (globally-redirect-io-to-connection connection)))

(defun update-redirection-after-close (closed-connection)
  "Update redirection after a connection closes."
  (when (eq *global-stdio-connection* closed-connection)
    (if (and (default-connection) *globally-redirect-io*)
        ;; Redirect to another connection.
        (globally-redirect-io-to-connection (default-connection))
        ;; No more connections, revert to the real streams.
        (progn (revert-global-io-redirection)
               (setq *global-stdio-connection* nil)))))

(add-hook *new-connection-hook* 'maybe-redirect-global-io)
(add-hook *connection-closed-hook* 'update-redirection-after-close)

```

7.4 Redirection during requests

We always redirect the standard streams to Emacs while evaluating an RPC. This is done with simple dynamic bindings.

```
(defun call-with-redirection (connection function)
  "Call FUNCTION with I/O streams redirected via CONNECTION."
  (declare (type function function))
  (let* ((io (connection.user-io connection))
         (in (connection.user-input connection))
         (out (connection.user-output connection))
         (*standard-output* out)
         (*error-output* out)
         (*trace-output* out)
         (*debug-io* io)
         (*query-io* io)
         (*standard-input* in)
         (*terminal-io* io))
    (funcall function)))

(defun read-from-emacs ()
  "Read and process a request from Emacs."
  (apply #'funcall (funcall (connection.read *emacs-connection*))))

(defun read-from-control-thread ()
  (receive))

(defun decode-message (stream)
  "Read an S-expression from STREAM using the SLIME protocol.
If a protocol error occurs then a SLIME-PROTOCOL-ERROR is signalled."
  (let ((*swank-state-stack* (cons :read-next-form *swank-state-stack*)))
    (handler-case
      (let* ((length (decode-message-length stream))
             (string (make-string length))
             (pos (read-sequence string stream)))
        (assert (= pos length) ())
        "Short read: length=~D pos=~D" length pos)
      (let ((form (read-form string)))
        (log-event "READ: ~A~%" string)
        form))
      (serious-condition (c)
        (error (make-condition 'slime-protocol-error :condition c))))))

(defun decode-message-length (stream)
  (let ((buffer (make-string 6)))
    (dotimes (i 6)
      (setf (aref buffer i) (read-char stream)))
    (parse-integer buffer :radix #x10)))

(defun read-form (string)
```

```

(with-standard-io-syntax
  (let ((*package* *swank-io-package*))
    (read-from-string string))))

(defvar *slime-features* nil
  "The feature list that has been sent to Emacs.")

(defun send-to-emacs (object)
  "Send OBJECT to Emacs."
  (funcall (connection.send *emacs-connection*) object))

(defun send-oob-to-emacs (object)
  (send-to-emacs object))

(defun send-to-control-thread (object)
  (send (connection.control-thread *emacs-connection*) object))

(defun encode-message (message stream)
  (let* ((string (princ1-to-string-for-emacs message))
        (length (1+ (length string))))
    (log-event "WRITE: ~A~%" string)
    (format stream "~6,'0x" length)
    (write-string string stream)
    (terpri stream)
    (force-output stream)))

(defun princ1-to-string-for-emacs (object)
  (with-standard-io-syntax
    (let ((*print-case* :downcase)
          (*print-readably* nil)
          (*print-pretty* nil)
          (*package* *swank-io-package*))
      (princ1-to-string object))))

(defun force-user-output ()
  (force-output (connection.user-io *emacs-connection*))
  (force-output (connection.user-output *emacs-connection*)))

(defun clear-user-input ()
  (clear-input (connection.user-input *emacs-connection*)))

(defvar *read-input-catch-tag* 0)

(defun intern-catch-tag (tag)
  ;; fixnums aren't eq in ABCL, so we use intern to create tags
  (intern (format nil "~D" tag) :swank))

(defun read-user-input-from-emacs ()
  (let ((tag (incf *read-input-catch-tag*)))
    (force-output)

```

```

(send-to-emacs `(:read-string ,(current-thread) ,tag))
(let ((ok nil))
  (unwind-protect
    (progl (catch (intern-catch-tag tag)
                 (loop (read-from-emacs)))
           (setq ok t))
    (unless ok
      (send-to-emacs `(:read-aborted ,(current-thread) ,tag))))))

(defslimefun take-input (tag input)
  "Return the string INPUT to the continuation TAG."
  (throw (intern-catch-tag tag) input))

(defun evaluate-in-emacs (string)
  (let ((tag (incf *read-input-catch-tag*)))
    (force-output)
    (send-to-emacs `(:evaluate-in-emacs ,string ,(current-thread) ,tag))
    (let ((ok nil))
      (unwind-protect
        (progl (catch (intern-catch-tag tag)
                     (loop (read-from-emacs)))
              (setq ok t))
        (unless ok
          (send-to-emacs `(:read-aborted ,(current-thread) ,tag))))))

(defun eval-in-emacs (form &optional nowait)
  "Eval FORM in Emacs."
  (destructuring-bind (fun &rest args) form
    (let ((fun (string-downcase (string fun))))
      (cond (nowait
             (send-to-emacs `(:eval-no-wait ,fun ,args)))
            (t
             (force-output)
             (let* ((tag (incf *read-input-catch-tag*)))
               (send-to-emacs `(:eval ,(current-thread) ,tag ,fun ,args))
               (receive-eval-result tag))))))

(defun receive-eval-result (tag)
  (let ((value (catch (intern-catch-tag tag)
                    (loop (read-from-emacs))))
        (destructure-case value
          (:ok value) value)
          (:abort) (abort))))

(defun connection-info ()
  "Return a list of the form:
\\(PID IMPLEMENTATION-TYPE IMPLEMENTATION-NAME FEATURES)."
  (setq *slime-features* *features*)
  (list (getpid)
        (lisp-implementation-type)

```

```

(lisp-implementation-type-name)
(features-for-emacs)
(connection.communication-style *emacs-connection*))

```

8 Reading and printing

```

(defmacro define-special (name doc)
  "Define a special variable NAME with doc string DOC.
This is like defvar, but NAME will not be initialized."
  `(progn
    (defvar ,name)
    (setf (documentation ',name 'variable) ,doc)))

```

```

(define-special *buffer-package*
  "Package corresponding to slime-buffer-package.

```

EVAL-FOR-EMACS binds `*buffer-package*`. Strings originating from a slime buffer are best read in this package. See also FROM-STRING and TO-STRING.")

```

(define-special *buffer-readtable*
  "Readtable associated with the current buffer")

```

```

(defmacro with-buffer-syntax ((&rest _) &body body)
  "Execute BODY with appropriate *package* and *readtable* bindings.

```

This should be used for code that is conceptionally executed in an Emacs buffer."

```

(destructuring-bind () _
  `(let ((*package* *buffer-package*))
    ;; Don't shadow *readtable* unnecessarily because that prevents
    ;; the user from assigning to it.
    (if (eq *readtable* *buffer-readtable*)
        (call-with-syntax-hooks (lambda () ,@body))
        (let ((*readtable* *buffer-readtable*)
            (call-with-syntax-hooks (lambda () ,@body)))))))

```

```

(defun from-string (string)
  "Read string in the *BUFFER-PACKAGE*"
  (with-buffer-syntax ()
    (let ((*read-suppress* nil))
      (read-from-string string))))

```

;; FIXME: deal with #\| etc. hard to do portably.

```

(defun tokenize-symbol (string)
  (let ((package (let ((pos (position #\: string)))
                    (if pos (subseq string 0 pos) nil)))
        (symbol (let ((pos (position #\: string :from-end t)))
                   (if pos (subseq string (1+ pos)) string)))
        (internp (search ":@" string)))

```

```

(values symbol package internp)))

;; FIXME: Escape chars are ignored
(defun casify (string)
  "Convert string accoring to readable-case."
  (ecase (readtable-case *readtable*)
    (:preserve string)
    (:upcase (string-upcase string))
    (:downcase (string-downcase string))
    (:invert (multiple-value-bind (lower upper) (determine-case string)
      (cond ((and lower upper) string)
            (lower (string-upcase string))
            (upper (string-downcase string))
            (t string))))))

(defun parse-symbol (string &optional (package *package*))
  "Find the symbol named STRING.
Return the symbol and a flag indicating whether the symbols was found."
  (multiple-value-bind (sname pname) (tokenize-symbol string)
    (let ((package (cond ((string= pname "") keyword-package)
                        (pname (find-package (casify pname)))
                        (t package))))
      (if package
          (find-symbol (casify sname) package)
          (values nil nil))))))

(defun parse-symbol-or-loose (string &optional (package *package*))
  (multiple-value-bind (symbol status) (parse-symbol string package)
    (if status
        (values symbol status)
        (error "Unknown symbol: ~A [in ~A]" string package))))

;; FIXME: interns the name
(defun parse-package (string)
  "Find the package named STRING.
Return the package or nil."
  (multiple-value-bind (name pos)
    (if (zerop (length string))
        (values :|| 0)
        (let ((*package* keyword-package)
            (ignore-errors (read-from-string string))))
    (if (and (or (keywordp name) (stringp name))
            (= (length string) pos))
        (find-package name))))

(defun to-string (string)
  "Write string in the *BUFFER-PACKAGE*."
  (with-buffer-syntax ()
    (prin1-to-string string)))

```

```

(defun guess-package-from-string (name &optional (default-package *package*))
  (or (and name
          (or (parse-package name)
              (find-package (string-upcase name))
              (parse-package (substitute #\ - #\! name))))
      default-package))

(defvar *readtable-alist* (default-readtable-alist)
  "An alist mapping package names to readtables.")

(defun guess-buffer-readtable (package-name &optional (default *readtable*))
  (let ((package (guess-package-from-string package-name)))
    (if package
        (or (cdr (assoc (package-name package) *readtable-alist*
                        :test #'string=))
            default)
        default)))

(defun valid-operator-symbol-p (symbol)
  "Test if SYMBOL names a function, macro, or special-operator."
  (or (fboundp symbol)
      (macro-function symbol)
      (special-operator-p symbol)))

(defun valid-operator-name-p (string)
  "Test if STRING names a function, macro, or special-operator."
  (let ((symbol (parse-symbol string)))
    (valid-operator-symbol-p symbol)))

```

9 Arglists

```

(defslimefun arglist-for-echo-area (names)
  "Return the arglist for the first function, macro, or special-op in NAMES."
  (handler-case
    (with-buffer-syntax ()
      (let ((name (find-if #'valid-operator-name-p names)))
        (if name (format-arglist-for-echo-area (parse-symbol name) name))))
    (error (cond)
            (format nil "ARGLIST: ~A" cond))))

(defun format-arglist-for-echo-area (symbol name)
  "Return SYMBOL's arglist as string for display in the echo area.
Use the string NAME as operator name."
  (let ((arglist (arglist symbol)))
    (etypecase arglist
      ((member :not-available)
       nil)
      (list
       (let ((enriched-arglist

```

```

      (if (extra-keywords symbol)
          ;; When there are extra keywords, we decode the
          ;; arglist, merge in the keywords and encode it
          ;; again.
          (let ((decoded-arglist (decode-arglist arglist))
                (enrich-decoded-arglist-with-extra-keywords
                 decoded-arglist (list symbol))
                (encode-arglist decoded-arglist))
              ;; Otherwise, just use the original arglist.
              ;; This works better for implementation-specific
              ;; lambda-list-keywords like CMUCL's &parse-body.
              arglist)))
      (arglist-to-string (cons name enriched-arglist)
                        (symbol-package symbol))))))

(defun clean-arglist (arglist)
  "Remove &whole, &environment, and &aux elements from ARGLIST."
  (cond ((null arglist) '())
        ((member (car arglist) '(&whole &environment))
         (clean-arglist (cddr arglist)))
        ((eq (car arglist) '&aux)
         '())
        (t (cons (car arglist) (clean-arglist (cdr arglist))))))

(defun arglist-to-string (arglist package)
  "Print the list ARGLIST for display in the echo area.
The argument name are printed without package qualifiers and
pretty printing of (function foo) as #'foo is suppressed."
  (setq arglist (clean-arglist arglist))
  (etypecase arglist
    (null "()")
    (cons
     (with-output-to-string (*standard-output*)
      (with-standard-io-syntax
       (let ((*package* package) (*print-case* :downcase)
             (*print-pretty* t) (*print-circle* nil) (*print-readably* nil)
             (*print-level* 10) (*print-length* 20))
         (pprint-logical-block (nil nil :prefix "(" :suffix ")")
          (loop
           (let ((arg (pop arglist)))
             (etypecase arg
              (symbol (princ arg))
              (string (princ arg))
              (cons (pprint-logical-block (nil nil :prefix "(" :suffix ")")
                (princ (car arg))
                (unless (null (cdr arg))
                  (write-char #\space))
                (pprint-fill *standard-output* (cdr arg) nil))))
              (when (null arglist) (return))
              (write-char #\space)

```

```

                (pprint-newline :fill))))))))))

(defun test-print-arglist (list string)
  (string= (arglist-to-string list (find-package :swank)) string))

;; Should work:
(progn
  (assert (test-print-arglist '(function cons) "(function cons)"))
  (assert (test-print-arglist '(quote cons) "(quote cons)"))
  (assert (test-print-arglist '(&key (function #'+)) "(&key (function #'+))"))
  (assert (test-print-arglist '(&whole x y z) "(y z)"))
  (assert (test-print-arglist '(x &aux y z) "(x)"))
  (assert (test-print-arglist '(x &environment env y) "(x y)"))
  ;; Expected failure:
  ;; (assert (test-print-arglist '(&key ((function f))) "(&key ((function f)))"))

(defun slimefun variable-desc-for-echo-area (variable-name)
  "Return a short description of VARIABLE-NAME, or NIL."
  (with-buffer-syntax ()
    (let ((sym (parse-symbol variable-name)))
      (if (and sym (boundp sym))
          (let ((*print-pretty* nil) (*print-level* 4)
              (*print-length* 10) (*print-circle* t))
            (format nil "~A => ~A" sym (symbol-value sym))))))

(defstruct (keyword-arg
            (:conc-name keyword-arg.)
            (:constructor make-keyword-arg (keyword arg-name default-arg)))
  keyword
  arg-name
  default-arg)

(defun decode-keyword-arg (arg)
  "Decode a keyword item of formal argument list.
Return three values: keyword, argument name, default arg."
  (cond ((symbolp arg)
         (make-keyword-arg (intern (symbol-name arg) keyword-package)
                           arg
                           nil))
        ((and (consp arg)
              (consp (car arg)))
         (make-keyword-arg (caar arg)
                           (cadar arg)
                           (cadr arg)))
        ((consp arg)
         (make-keyword-arg (intern (symbol-name (car arg)) keyword-package)
                           (car arg)
                           (cadr arg)))
        (t
         (error "Bad keyword item of formal argument list"))))

```

```

(defun encode-keyword-arg (arg)
  (if (eql (intern (symbol-name (keyword-arg.arg-name arg))
                  keyword-package)
          (keyword-arg.keyword arg))
      (if (keyword-arg.default-arg arg)
          (list (keyword-arg.arg-name arg)
                (keyword-arg.default-arg arg))
          (keyword-arg.arg-name arg))
      (let ((keyword/name (list (keyword-arg.arg-name arg)
                                (keyword-arg.keyword arg))))
        (if (keyword-arg.default-arg arg)
            (list keyword/name
                  (keyword-arg.default-arg arg))
            (list keyword/name))))))

(progn
  (assert (equalp (decode-keyword-arg 'x)
                  (make-keyword-arg :x 'x nil)))
  (assert (equalp (decode-keyword-arg '(x t))
                  (make-keyword-arg :x 'x t)))
  (assert (equalp (decode-keyword-arg '(:x y))
                  (make-keyword-arg :x 'y nil)))
  (assert (equalp (decode-keyword-arg '(:x y) t)
                  (make-keyword-arg :x 'y t))))

(defstruct (optional-arg
            (:conc-name optional-arg.)
            (:constructor make-optional-arg (arg-name default-arg)))
  arg-name
  default-arg)

(defun decode-optional-arg (arg)
  "Decode an optional item of a formal argument list.
Return an OPTIONAL-ARG structure."
  (etypecase arg
    (symbol (make-optional-arg arg nil))
    (list (make-optional-arg (car arg) (cadr arg)))))

(defun encode-optional-arg (optional-arg)
  (if (optional-arg.default-arg optional-arg)
      (list (optional-arg.arg-name optional-arg)
            (optional-arg.default-arg optional-arg))
      (optional-arg.arg-name optional-arg)))

(progn
  (assert (equalp (decode-optional-arg 'x)
                  (make-optional-arg 'x nil)))
  (assert (equalp (decode-optional-arg '(x t))
                  (make-optional-arg 'x t))))

```

```

(defstruct (arglist (:conc-name arglist.))
  required-args      ; list of the required arguments
  optional-args      ; list of the optional arguments
  key-p              ; whether &key appeared
  keyword-args       ; list of the keywords
  rest               ; name of the &rest or &body argument (if any)
  body-p             ; whether the rest argument is a &body
  allow-other-keys-p ; whether &allow-other-keys appeared)

(defun decode-arglist (arglist)
  "Parse the list ARGLIST and return an ARGLIST structure."
  (let ((mode nil)
        (result (make-arglist)))
    (dolist (arg arglist)
      (cond
       ((eql arg '&allow-other-keys)
        (setf (arglist.allow-other-keys-p result) t))
       ((eql arg '&key)
        (setf (arglist.key-p result) t
              mode arg))
       ((member arg lambda-list-keywords)
        (setq mode arg))
       (t
        (case mode
          (&key
           (push (decode-keyword-arg arg)
                 (arglist.keyword-args result)))
          (&optional
           (push (decode-optional-arg arg)
                 (arglist.optional-args result)))
          (&body
           (setf (arglist.body-p result) t
                 (arglist.rest result) arg))
          (&rest
           (setf (arglist.rest result) arg))
          ((nil)
           (push arg (arglist.required-args result)))
          (&whole &environment)
           (setf mode nil))))))
    (setf (arglist.required-args result)
          (nreverse (arglist.required-args result)))
          (arglist.optional-args result)
          (nreverse (arglist.optional-args result)))
          (arglist.keyword-args result)
          (nreverse (arglist.keyword-args result)))
          result))

(defun encode-arglist (decoded-arglist)
  (append (arglist.required-args decoded-arglist)

```

```

    (when (arglist.optional-args decoded-arglist)
      '(&optional))
    (mapcar #'encode-optional-arg (arglist.optional-args decoded-arglist))
    (when (arglist.key-p decoded-arglist)
      '(&key))
    (mapcar #'encode-keyword-arg (arglist.keyword-args decoded-arglist))
    (when (arglist.allow-other-keys-p decoded-arglist)
      '(&allow-other-keys))
    (cond ((not (arglist.rest decoded-arglist))
          '())
          ((arglist.body-p decoded-arglist)
           '(&body ,(arglist.rest decoded-arglist)))
          (t
           '(&rest ,(arglist.rest decoded-arglist))))))

(defun arglist-keywords (arglist)
  "Return the list of keywords in ARGLIST.
As a secondary value, return whether &allow-other-keys appears."
  (let ((decoded-arglist (decode-arglist arglist)))
    (values (arglist.keyword-args decoded-arglist)
            (arglist.allow-other-keys-p decoded-arglist))))

(defun methods-keywords (methods)
  "Collect all keywords in the arglists of METHODS.
As a secondary value, return whether &allow-other-keys appears somewhere."
  (let ((keywords '())
        (allow-other-keys nil))
    (dolist (method methods)
      (multiple-value-bind (kw aok)
        (arglist-keywords
         (swank-mop:method-lambda-list method))
        (setq keywords (remove-duplicates (append keywords kw)
                                          :key #'keyword-arg.keyword)
              allow-other-keys (or allow-other-keys aok))))
    (values keywords allow-other-keys)))

(defun generic-function-keywords (generic-function)
  "Collect all keywords in the methods of GENERIC-FUNCTION.
As a secondary value, return whether &allow-other-keys appears somewhere."
  (methods-keywords
   (swank-mop:generic-function-methods generic-function)))

(defun applicable-methods-keywords (generic-function classes)
  "Collect all keywords in the methods of GENERIC-FUNCTION that are
applicable for argument of CLASSES. As a secondary value, return
whether &allow-other-keys appears somewhere."
  (methods-keywords
   (swank-mop:compute-applicable-methods-using-classes
    generic-function classes)))

```

```

(defun arglist-to-template-string (arglist package)
  "Print the list ARGLIST for insertion as a template for a function call."
  (decoded-arglist-to-template-string
   (decode-arglist arglist) package))

(defun decoded-arglist-to-template-string (decoded-arglist package &key (prefix
  (with-output-to-string (*standard-output*)
    (with-standard-io-syntax
      (let ((*package* package) (*print-case* :downcase)
            (*print-pretty* t) (*print-circle* nil) (*print-readably* nil)
            (*print-level* 10) (*print-length* 20))
        (pprint-logical-block (nil nil :prefix prefix :suffix suffix)
          (print-decoded-arglist-as-template decoded-arglist)))))))

(defun print-decoded-arglist-as-template (decoded-arglist)
  (let ((first-p t))
    (flet ((space ()
             (unless first-p
               (write-char #\space)
               (pprint-newline :fill))
             (setq first-p nil)))
      (dolist (arg (arglist.required-args decoded-arglist))
        (space)
        (princ arg))
      (dolist (arg (arglist.optional-args decoded-arglist))
        (space)
        (format t "[~A]" (optional-arg.arg-name arg)))
      (dolist (keyword-arg (arglist.keyword-args decoded-arglist))
        (space)
        (let ((arg-name (keyword-arg.arg-name keyword-arg))
              (keyword (keyword-arg.keyword keyword-arg)))
          (format t "~W ~A"
                  (if (keywordp keyword) keyword ``',keyword)
                  arg-name)))
      (when (and (arglist.rest decoded-arglist)
                 (or (not (arglist.keyword-args decoded-arglist))
                     (arglist.allow-other-keys-p decoded-arglist)))
        (if (arglist.body-p decoded-arglist)
            (pprint-newline :mandatory)
            (space))
        (format t "~A..." (arglist.rest decoded-arglist))))
    (pprint-newline :fill))

(defgeneric extra-keywords (operator &rest args)
  (:documentation "Return a list of extra keywords of OPERATOR (a
symbol) when applied to the (unevaluated) ARGS. As a secondary value,
return whether other keys are allowed."))

(defmethod extra-keywords (operator &rest args)
  ;; default method

```

```

(declare (ignore args))
(let ((symbol-function (symbol-function operator)))
  (if (typep symbol-function 'generic-function)
      (generic-function-keywords symbol-function)
      nil)))

(defmethod extra-keywords ((operator (eql 'make-instance))
                          &rest args)
  (unless (null args)
    (let ((class-name-form (car args)))
      (when (and (listp class-name-form)
                 (= (length class-name-form) 2)
                 (eq (car class-name-form) 'quote))
        (let* ((class-name (cadr class-name-form))
               (class (find-class class-name nil)))
          (unless (swank-mop:class-finalized-p class)
            ;; Try to finalize the class, which can fail if
            ;; superclasses are not defined yet
            (handler-case (swank-mop:finalize-inheritance class)
              (program-error (c)
                (declare (ignore c))))))
          (when class
            ;; We have the case (make-instance 'CLASS ...)
            ;; with a known CLASS.
            (multiple-value-bind (slots allow-other-keys-p)
              (if (swank-mop:class-finalized-p class)
                  (values (swank-mop:class-slots class) nil)
                  (values (swank-mop:class-direct-slots class) t))
              (let ((slot-init-keywords
                    (loop for slot in slots append
                        (mapcar (lambda (initarg)
                                (make-keyword-arg
                                 initarg
                                 initarg ; FIXME
                                 (swank-mop:slot-definition-initform slot)
                                 (swank-mop:slot-definition-initargs slot))))
                          (initialize-instance-keywords
                           (applicable-methods-keywords #'initialize-instance
                                                           (list class))))
                    (return-from extra-keywords
                      (values (append slot-init-keywords
                                      initialize-instance-keywords)
                              allow-other-keys-p))))))
                (call-next-method)))
            (defun enrich-decoded-arglist-with-extra-keywords (decoded-arglist form)
              (multiple-value-bind (extra-keywords extra-aok)
                (apply #'extra-keywords form)
                ;; enrich the list of keywords with the extra keywords
                (when extra-keywords

```

```

      (setf (arglist.key-p decoded-arglist) t)
      (setf (arglist.keyword-args decoded-arglist)
            (remove-duplicates
              (append (arglist.keyword-args decoded-arglist)
                      extra-keywords)
                :key #'keyword-arg.keyword)))
      (setf (arglist.allow-other-keys-p decoded-arglist)
            (or (arglist.allow-other-keys-p decoded-arglist) extra-aok)))
      decoded-arglist)

(defun slimefun arglist-for-insertion (name)
  (with-buffer-syntax ()
    (let ((symbol (parse-symbol name)))
      (cond
        ((and symbol
              (valid-operator-name-p name))
         (let ((arglist (arglist symbol)))
           (etypecase arglist
             ((member :not-available)
              :not-available)
             (list
              (list
                (let ((decoded-arglist (decode-arglist arglist)))
                  (enrich-decoded-arglist-with-extra-keywords decoded-arglist
                                                                (list symbol))
                  (decoded-arglist-to-template-string decoded-arglist
                                                       *buffer-package*)))))))
          (t
           :not-available))))))

(defvar *remove-keywords-alist*
  '( (:test :test-not)
    (:test-not :test)))

(defun remove-actual-args (decoded-arglist actual-arglist)
  "Remove from DECODED-ARGLIST the arguments that have already been
provided in ACTUAL-ARGLIST."
  (loop while (and actual-arglist
                   (arglist.required-args decoded-arglist))
        do (progn (pop actual-arglist)
                  (pop (arglist.required-args decoded-arglist))))
  (loop while (and actual-arglist
                   (arglist.optional-args decoded-arglist))
        do (progn (pop actual-arglist)
                  (pop (arglist.optional-args decoded-arglist))))
  (loop for keyword in actual-arglist by #'cddr
        for keywords-to-remove = (cdr (assoc keyword *remove-keywords-alist*))
        do (setf (arglist.keyword-args decoded-arglist)
                  (remove-if (lambda (kw)
                              (or (eql kw keyword)
                                  (member kw keywords-to-remove))))))

```

```

                                (arglist.keyword-args decoded-arglist)
                                :key #'keyword-arg.keyword))))

(defslimefun complete-form (form-string)
  "Read FORM-STRING in the current buffer package, then complete it
  by adding a template for the missing arguments."
  (with-buffer-syntax ()
    (handler-case
      (let ((form (read-from-string form-string)))
        (when (consp form)
          (let ((operator-form (first form))
                (argument-forms (rest form)))
            (when (and (symbolp operator-form)
                      (valid-operator-symbol-p operator-form))
              (let ((arglist (arglist operator-form)))
                (etypecase arglist
                  ((member :not-available)
                   :not-available)
                  (list
                   (let ((decoded-arglist (decode-arglist arglist)))
                     (enrich-decoded-arglist-with-extra-keywords decoded-arglist
                           ;; get rid of formal args already provided
                           (remove-actual-args decoded-arglist argument-forms)
                           (return-from complete-form
                                (decoded-arglist-to-template-string decoded-arglist
                            *buffer-package*
                            :prefix ""))))))))
              :not-available)
            (reader-error (c)
              (declare (ignore c))
              :not-available))))))

```

10 Evaluation

```

(defvar *pending-continuations* '()
  "List of continuations for Emacs. (thread local)")

(defun guess-buffer-package (string)
  "Return a package for STRING.
  Fall back to the the current if no such package exists."
  (or (guess-package-from-string string nil)
      *package*))

(defun eval-for-emacs (form buffer-package id)
  "Bind *BUFFER-PACKAGE* BUFFER-PACKAGE and evaluate FORM.
  Return the result to the continuation ID.
  Errors are trapped and invoke our debugger."
  (call-with-debugger-hook
    #'swank-debugger-hook

```

```

(lambda ()
  (let (ok result)
    (unwind-protect
      (let ((*buffer-package* (guess-buffer-package buffer-package))
            (*buffer-readtable* (guess-buffer-readtable buffer-package))
            (*pending-continuations* (cons id *pending-continuations*)))
        (assert (packagep *buffer-package*))
        (assert (readtablep *buffer-readtable*))
        (setq result (eval form))
        (force-output)
        (run-hook *pre-reply-hook*)
        (setq ok t))
      (force-user-output)
      (send-to-emacs `(:return ,(current-thread)
                            ,(if ok `(:ok ,result) `(:abort))
                            ,id))))))

(defun format-values-for-echo-area (values)
  (with-buffer-syntax ()
    (let ((*print-readably* nil))
      (cond ((null values) "; No value")
            ((and (null (cdr values)) (integerp (car values)))
             (let ((i (car values)))
               (format nil "~D (#x~X, #o~O, #b~B)" i i i i)))
            (t (format nil "~{~S^, ~}" values))))))

(defunslimefun interactive-eval (string)
  (with-buffer-syntax ()
    (let ((values (multiple-value-list (eval (read-from-string string)))))
      (fresh-line)
      (force-output)
      (format-values-for-echo-area values))))

(defunslimefun eval-and-grab-output (string)
  (with-buffer-syntax ()
    (let* ((s (make-string-output-stream))
           (*standard-output* s)
           (values (multiple-value-list (eval (read-from-string string)))))
      (list (get-output-stream-string s)
            (format nil "~{~S^~%}" values))))))

(defun eval-region (string &optional package-update-p)
  "Evaluate STRING and return the result.
If PACKAGE-UPDATE-P is non-nil, and evaluation causes a package
change, then send Emacs an update."
  (unwind-protect
    (with-input-from-string (stream string)
      (let (- values)
        (loop
          (let ((form (read stream nil stream))))

```

```

        (when (eq form stream)
            (fresh-line)
            (force-output)
            (return (values values -)))
        (setq - form)
        (setq values (multiple-value-list (eval form)))
        (force-output))))))
    (when (and package-update-p (not (eq *package* *buffer-package*)))
        (send-to-emacs
         (list :new-package (package-name *package*)
               (package-string-for-prompt *package*))))))

(defun package-string-for-prompt (package)
  "Return the shortest nickname (or canonical name) of PACKAGE."
  (or (canonical-package-nickname package)
      (auto-abbreviated-package-name package)
      (shortest-package-nickname package)))

(defun canonical-package-nickname (package)
  "Return the canonical package nickname, if any, of PACKAGE."
  (cdr (assoc (package-name package) *canonical-package-nicknames*
              :test #'string=)))

(defun auto-abbreviated-package-name (package)
  "Return an abbreviated 'name' for PACKAGE."

  N.B. this is not an actual package name or nickname."
  (when *auto-abbreviate-dotted-packages*
      (let ((last-dot (position #\. (package-name package) :from-end t)))
          (when last-dot (subseq (package-name package) (1+ last-dot))))))

(defun shortest-package-nickname (package)
  "Return the shortest nickname (or canonical name) of PACKAGE."
  (loop for name in (cons (package-name package) (package-nicknames package))
        for shortest = name then (if (< (length name) (length shortest))
                                     name
                                     shortest)
        finally (return shortest)))

(defslimefun interactive-eval-region (string)
  (with-buffer-syntax ()
    (format-values-for-echo-area (eval-region string))))

(defslimefun re-evaluate-defvar (form)
  (with-buffer-syntax ()
    (let ((form (read-from-string form)))
      (destructuring-bind (dv name &optional value doc) form
        (declare (ignore value doc))
        (assert (eq dv 'defvar))
        (makunbound name))
    )))

```

```

        (prin1-to-string (eval form))))))

(defun swank-pprint (list)
  "Bind some printer variables and pretty print each object in LIST."
  (with-buffer-syntax ()
    (call-with-bindings
      *swank-pprint-bindings*
      (lambda ()
        (let ((*print-pretty* t))
          (cond ((null list) "; No value")
                (t (with-output-to-string (*standard-output*)
                    (dolist (o list)
                      (pprint o)
                      (terpri))))))))))

(defslimefun pprint-eval (string)
  (with-buffer-syntax ()
    (swank-pprint (multiple-value-list (eval (read-from-string string))))))

(defslimefun set-package (package)
  "Set *package* to PACKAGE.
Return its name and the string to use in the prompt."
  (let ((p (setq *package* (guess-package-from-string package))))
    (list (package-name p) (package-string-for-prompt p))))

(defslimefun listener-eval (string)
  (clear-user-input)
  (with-buffer-syntax ()
    (multiple-value-bind (values last-form) (eval-region string t)
      (setq +++ ++ ++ + + last-form
            *** ** ** * * (car values)
            /// // // / / values)
      (cond ((null values) "; No value")
            (t
             (format nil "~{~S~^~%~}" values))))))

(defslimefun ed-in-emacs (&optional what)
  "Edit WHAT in Emacs.

WHAT can be:
A filename (string),
A list (FILENAME LINE [COLUMN]),
A function name (symbol),
nil."
  (let ((target
        (cond ((and (listp what) (pathnamep (first what)))
               (cons (canonicalize-filename (car what)) (cdr what)))
              ((pathnamep what)
               (canonicalize-filename what))
              (t what))))))

```

```
(send-oob-to-emacs `(:ed ,target))))
```

11 Debugger

```
(defun swank-debugger-hook (condition hook)
  "Debugger function for binding *DEBUGGER-HOOK*.
Sends a message to Emacs declaring that the debugger has been entered,
then waits to handle further requests from Emacs. Eventually returns
after Emacs causes a restart to be invoked."
  (declare (ignore hook))
  (flet ((debug-it () (debug-in-emacs condition)))
    (cond (*emacs-connection*
           (debug-it))
          ((default-connection)
           (with-connection ((default-connection))
             (debug-in-emacs condition))))))

(defvar *global-debugger* t
  "Non-nil means the Swank debugger hook will be installed globally.")

(add-hook *new-connection-hook* 'install-debugger)
(defun install-debugger (connection)
  (declare (ignore connection))
  (when *global-debugger*
    (setq *debugger-hook* #'swank-debugger-hook)))
```

11.1 Debugger loop

These variables are dynamically bound during debugging.

```
(defvar *swank-debugger-condition* nil
  "The condition being debugged.")

(defvar *sldb-level* 0
  "The current level of recursive debugging.")

(defvar *sldb-initial-frames* 20
  "The initial number of backtrace frames to send to Emacs.")

(defvar *sldb-restarts* nil
  "The list of currently active restarts.")

(defvar *sldb-stepping-p* nil
  "True when during execution of a stepp command.")

(defun debug-in-emacs (condition)
  (let ((*swank-debugger-condition* condition)
        (*sldb-restarts* (compute-restarts condition))
        (*package* (or (and (boundp '*buffer-package*)
```

```

                (symbol-value '*buffer-package*))
                *package*))
    (*sldb-level* (1+ *sldb-level*))
    (*sldb-stepping-p* nil)
    (*swank-state-stack* (cons :swank-debugger-hook *swank-state-stack*)))
(force-user-output)
(call-with-bindings
 *sldb-printer-bindings*
 (lambda ()
  (call-with-debugging-environment
   (lambda () (sldb-loop *sldb-level*))))))

(defun sldb-loop (level)
  (unwind-protect
   (catch 'sldb-enter-default-debugger
    (send-to-emacs
     (list* :debug (current-thread) *sldb-level*
            (debugger-info-for-emacs 0 *sldb-initial-frames*)))
    (loop (catch 'sldb-loop-catcher
              (with-simple-restart (abort "Return to sldb level ~D." level)
                (send-to-emacs (list :debug-activate (current-thread)
                                     *sldb-level*)))
              (handler-bind ((sldb-condition #'handle-sldb-condition))
                (read-from-emacs))))))
    (send-to-emacs `(:debug-return
                    ,(current-thread) ,level ,*sldb-stepping-p*))))

(defun handle-sldb-condition (condition)
  "Handle an internal debugger condition.
Rather than recursively debug the debugger (a dangerous idea!), these
conditions are simply reported."
  (let ((real-condition (original-condition condition)))
    (send-to-emacs `(:debug-condition ,(current-thread)
                                ,(princ-to-string real-condition))))
  (throw 'sldb-loop-catcher nil))

(defun safe-condition-message (condition)
  "Safely print condition to a string, handling any errors during
printing."
  (let ((*print-pretty* t))
    (handler-case
     (format-sldb-condition condition)
    (error (cond)
            ;; Beware of recursive errors in printing, so only use the condition
            ;; if it is printable itself:
            (format nil "Unable to display error condition~@[[: ~A~]"
                    (ignore-errors (princ-to-string cond)))))))

(defun debugger-condition-for-emacs ()
  (list (safe-condition-message *swank-debugger-condition*)

```

```

      (format nil " [Condition of type ~S]"
              (type-of *swank-debugger-condition*))
      (condition-references *swank-debugger-condition*)
      (condition-extras *swank-debugger-condition*))

(defun format-restarts-for-emacs ()
  "Return a list of restarts for *swank-debugger-condition* in a
format suitable for Emacs."
  (loop for restart in *sldb-restarts*
        collect (list (princ-to-string (restart-name restart))
                      (princ-to-string restart))))

(defun frame-for-emacs (n frame)
  (let* ((label (format nil " ~2D: " n))
         (string (with-output-to-string (stream)
                  (princ label stream)
                  (print-frame frame stream))))
    (subseq string (length label))))

```

11.2 SLDB entry points

```

(defunslimefun sldb-break-with-default-debugger ()
  "Invoke the default debugger by returning from our debugger-loop."
  (throw 'sldb-enter-default-debugger nil))

(defunslimefun backtrace (start end)
  "Return a list ((I FRAME) ...) of frames from START to END.
I is an integer describing and FRAME a string."
  (loop for frame in (compute-backtrace start end)
        for i from start
        collect (list i (frame-for-emacs i frame))))

(defunslimefun debugger-info-for-emacs (start end)
  "Return debugger state, with stack frames from START to END.
The result is a list:
(condition ({restart}*) ({stack-frame}*) (cont*))
where
condition ::= (description type [extra])
restart    ::= (name description)
stack-frame ::= (number description)
extra      ::= (:references and other random things)
cont       ::= continuation
condition---a pair of strings: message, and type. If show-source is
not nil it is a frame number for which the source should be displayed.

restart---a pair of strings: restart name, and description.

stack-frame---a number from zero (the top), and a printed
representation of the frame's call.

```

continuation---the id of a pending Emacs continuation.

Below is an example return value. In this case the condition was a division by zero (multi-line description), and only one frame is being fetched (start=0, end=1).

```
((\"Arithmetic error DIVISION-BY-ZERO signalled.
Operation was KERNEL::DIVISION, operands (1 0).\"
 \"[Condition of type DIVISION-BY-ZERO]\"
 (\"ABORT\" \"Return to Slime toplevel.\")
 (\"ABORT\" \"Return to Top-Level.\"))
 ((0 \"(KERNEL::INTEGER-/-INTEGER 1 0)\")
 (4))
 (list (debugger-condition-for-emacs)
       (format-restarts-for-emacs)
       (backtrace start end)
       *pending-continuations*))

(defun nth-restart (index)
  (nth index *sldb-restarts*))

(defslimefun invoke-nth-restart (index)
  (invoke-restart-interactively (nth-restart index)))

(defslimefun sldb-abort ()
  (invoke-restart (find 'abort *sldb-restarts* :key #'restart-name)))

(defslimefun sldb-continue ()
  (continue))

(defslimefun throw-to-toplevel ()
  \"Use THROW to abort an RPC from Emacs.
If we are not evaluating an RPC then ABORT instead.\"
  (ignore-errors (throw 'slime-toplevel nil))
  ;; If we get here then there was no catch. Try aborting as a fallback.
  ;; That makes the 'q' command in SLDB safer to use with threads.
  (abort))

(defslimefun invoke-nth-restart-for-emacs (sldb-level n)
  \"Invoke the Nth available restart.
SLDB-LEVEL is the debug level when the request was made. If this
has changed, ignore the request.\"
  (when (= sldb-level *sldb-level*)
    (invoke-nth-restart n)))

(defslimefun eval-string-in-frame (string index)
  (to-string (eval-in-frame (from-string string) index)))

(defslimefun pprint-eval-string-in-frame (string index)
  (swank-pprint
```

```

      (multiple-value-list
        (eval-in-frame (from-string string) index))))

(defslimefun frame-locals-for-emacs (index)
  "Return a property list ((&key NAME ID VALUE) ...) describing
the local variables in the frame INDEX."
  (mapcar (lambda (frame-locals)
            (destructuring-bind (&key name id value) frame-locals
              (list :name (prin1-to-string name) :id id
                    :value (to-string value))))
          (frame-locals index)))

(defslimefun frame-catch-tags-for-emacs (frame-index)
  (mapcar #'to-string (frame-catch-tags frame-index)))

(defslimefun sldb-disassemble (index)
  (with-output-to-string (*standard-output*)
    (disassemble-frame index)))

(defslimefun sldb-return-from-frame (index string)
  (let ((form (from-string string)))
    (to-string (multiple-value-list (return-from-frame index form)))))

(defslimefun sldb-break (name)
  (with-buffer-syntax ()
    (sldb-break-at-start (read-from-string name))))

(defslimefun sldb-step (frame)
  (cond ((find-restart 'continue)
         (activate-stepping frame)
         (setq *sldb-stepping-p* t)
         (continue))
        (t
         (error "No continue restart."))))

```

12 Compilation Commands.

```

(defvar *compiler-notes* '()
  "List of compiler notes for the last compilation unit.")

(defun clear-compiler-notes ()
  (setf *compiler-notes* '()))

(defun canonicalize-filename (filename)
  (namestring (truename filename)))

(defslimefun compiler-notes-for-emacs ()
  "Return the list of compiler notes for the last compilation unit."
  (reverse *compiler-notes*))

```

```

(defun measure-time-interval (fn)
  "Call FN and return the first return value and the elapsed time.
The time is measured in microseconds."
  (declare (type function fn))
  (let ((before (get-internal-real-time)))
    (values
     (funcall fn)
     (* (- (get-internal-real-time) before)
        (/ 1000000 internal-time-units-per-second)))))

(defun record-note-for-condition (condition)
  "Record a note for a compiler-condition."
  (push (make-compiler-note condition) *compiler-notes*))

(defun make-compiler-note (condition)
  "Make a compiler note data structure from a compiler-condition."
  (declare (type compiler-condition condition))
  (list* :message (message condition)
         :severity (severity condition)
         :location (location condition)
         :references (references condition)
         (let ((s (short-message condition)))
           (if s (list :short-message s)))))

(defun swank-compiler (function)
  (clear-compiler-notes)
  (with-simple-restart (abort "Abort SLIME compilation.")
    (multiple-value-bind (result usecs)
      (handler-bind ((compiler-condition #'record-note-for-condition))
        (measure-time-interval function)))
    (list (to-string result)
          (format nil "~,2F" (/ usecs 1000000.0)))))

(defslimefun compile-file-for-emacs (filename load-p)
  "Compile FILENAME and, when LOAD-P, load the result.
Record compiler notes signalled as `compiler-condition's."
  (with-buffer-syntax ()
    (swank-compiler (lambda () (swank-compile-file filename load-p)))))

(defslimefun compile-string-for-emacs (string buffer position directory)
  "Compile STRING (exerpted from BUFFER at POSITION).
Record compiler notes signalled as `compiler-condition's."
  (with-buffer-syntax ()
    (swank-compiler
     (lambda ()
       (swank-compile-string string :buffer buffer :position position
                             :directory directory)))))

(defslimefun operate-on-system-for-emacs (system-name operation &rest keywords)

```

```

"Compile and load SYSTEM using ASDF.
Record compiler notes signalled as 'compiler-condition's."
  (swank-compiler
    (lambda ()
      (apply #'operate-on-system system-name operation keywords))))

(defun asdf-central-registry ()
  (when (find-package :asdf)
    (symbol-value (find-symbol (string :*central-registry*) :asdf))))

(defslimefun list-all-systems-in-central-registry ()
  "Returns a list of all systems in ASDF's central registry."
  (loop for dir in (asdf-central-registry)
        for defaults = (eval dir)
        when defaults
        nconc (mapcar #'file-namestring
                      (directory
                       (make-pathname :defaults defaults
                                       :version :newest
                                       :type "asd"
                                       :name :wild
                                       :case :local)))))

(defun file-newer-p (new-file old-file)
  "Returns true if NEW-FILE is newer than OLD-FILE."
  (> (file-write-date new-file) (file-write-date old-file)))

(defun requires-compile-p (source-file)
  (let ((fasl-file (probe-file (compile-file-pathname source-file))))
    (or (not fasl-file)
        (file-newer-p source-file fasl-file))))

(defslimefun compile-file-if-needed (filename loadp)
  (cond ((requires-compile-p filename)
         (compile-file-for-emacs filename loadp))
        (loadp
         (load (compile-file-pathname filename))
         nil)))

```

13 Loading

```

(defslimefun load-file (filename)
  (to-string (load filename)))

(defslimefun load-file-set-package (filename &optional package)
  (load-file filename)
  (if package
      (set-package package)))

```

14 Macroexpansion

```
(defun apply-macro-expander (expander string)
  (declare (type function expander))
  (with-buffer-syntax ()
    (swank-pprint (list (funcall expander (from-string string))))))

(defslimefun swank-macroexpand-1 (string)
  (apply-macro-expander #'macroexpand-1 string))

(defslimefun swank-macroexpand (string)
  (apply-macro-expander #'macroexpand string))

(defslimefun swank-macroexpand-all (string)
  (apply-macro-expander #'macroexpand-all string))

(defslimefun disassemble-symbol (name)
  (with-buffer-syntax ()
    (with-output-to-string (*standard-output*)
      (let ((*print-readably* nil))
        (disassemble (fdefinition (from-string name)))))))
```

15 Basic completion

```
(defslimefun completions (string default-package-name)
  "Return a list of completions for a symbol designator STRING.
```

The result is the list (COMPLETION-SET COMPLETED-PREFIX). COMPLETION-SET is the list of all matching completions, and COMPLETED-PREFIX is the best (partial) completion of the input string.

If STRING is package qualified the result list will also be qualified. If string is non-qualified the result strings are also not qualified and are considered relative to DEFAULT-PACKAGE-NAME.

The way symbols are matched depends on the symbol designator's format. The cases are as follows:

```
FOO          - Symbols with matching prefix and accessible in the buffer package.
PKG:FOO      - Symbols with matching prefix and external in package PKG.
PKG::FOO     - Symbols with matching prefix and accessible in package PKG."
```

```
(let ((completion-set (completion-set string default-package-name
                                     #'compound-prefix-match)))
  (list completion-set (longest-completion completion-set))))
```

```
(defslimefun simple-completions (string default-package-name)
  "Return a list of completions for a symbol designator STRING."
  (let ((completion-set (completion-set string default-package-name
```

```

                                #'prefix-match-p)))
(list completion-set (longest-common-prefix completion-set))))

```

15.1 Find completion set

```

(defun completion-set (string default-package-name matchp)
  "Return the set of completion-candidates as strings."
  (multiple-value-bind (name package-name package internal-p)
    (parse-completion-arguments string default-package-name)
    (let* ((symbols (and package
                          (find-matching-symbols name
                                                  package
                                                  (and (not internal-p)
                                                       package-name)
                                                  matchp)))
           (packs (and (not package-name)
                       (find-matching-packages name matchp)))
           (converter (output-case-converter name))
           (strings
            (mapcar converter
                    (nconc (mapcar #'symbol-name symbols) packs))))
          (format-completion-set strings internal-p package-name))))

(defun find-matching-symbols (string package external test)
  "Return a list of symbols in PACKAGE matching STRING.
TEST is called with two strings. If EXTERNAL is true, only external
symbols are returned."
  (let ((completions '())
        (converter (output-case-converter string)))
    (flet ((symbol-matches-p (symbol)
              (and (or (not external)
                      (symbol-external-p symbol package))
                   (funcall test string
                             (funcall converter (symbol-name symbol))))))
      (do-symbols (symbol package)
        (when (symbol-matches-p symbol)
          (push symbol completions))))
    (remove-duplicates completions)))

(defun symbol-external-p (symbol &optional (package (symbol-package symbol)))
  "True if SYMBOL is external in PACKAGE.
If PACKAGE is not specified, the home package of SYMBOL is used."
  (multiple-value-bind (_ status)
    (find-symbol (symbol-name symbol) (or package (symbol-package symbol)))
    (declare (ignore _))
    (eq status :external)))

(defun find-matching-packages (name matcher)
  "Return a list of package names matching NAME with MATCHER.
MATCHER is a two-argument predicate."

```

```

(let ((to-match (string-upcase name)))
  (remove-if-not (lambda (x) (funcall matcher to-match x))
    (mapcar (lambda (pkgname)
              (concatenate 'string pkgname ":"))
            (mapcar #'package-name (list-all-packages))))))

(defun parse-completion-arguments (string default-package-name)
  "Parse STRING as a symbol designator.
Return these values:
SYMBOL-NAME
PACKAGE-NAME, or nil if the designator does not include an explicit package.
PACKAGE, the package to complete in
INTERNAL-P, if the symbol is qualified with `::'."
  (multiple-value-bind (name package-name internal-p)
    (tokenize-symbol string)
    (let ((package (carefully-find-package package-name default-package-name)))
      (values name package-name package internal-p))))

(defun carefully-find-package (name default-package-name)
  "Find the package with name NAME, or DEFAULT-PACKAGE-NAME, or the
*buffer-package*. NAME and DEFAULT-PACKAGE-NAME can be nil."
  (let ((string (cond ((equal name "") "KEYWORD")
                      (t (or name default-package-name)))))
    (if string
        (guess-package-from-string string nil)
        *buffer-package*)))

```

15.2 Format completion results

We try to format results in the case as inputs. If you complete FOO then your result should include FOOBAR rather than foobar.

```

(defun format-completion-set (strings internal-p package-name)
  "Format a set of completion strings.
Returns a list of completions with package qualifiers if needed."
  (mapcar (lambda (string)
            (format-completion-result string internal-p package-name))
          (sort strings #'string<)))

(defun format-completion-result (string internal-p package-name)
  (let ((prefix (cond (internal-p (format nil "~A:" package-name))
                      (package-name (format nil "~A:" package-name))
                      (t ""))))
    (values (concatenate 'string prefix string)
            (length prefix))))

(defun output-case-converter (input)
  "Return a function to case convert strings for output.
INPUT is used to guess the preferred case."
  (ecase (readtable-case *readtable*)

```

```

(:uppercase (if (some #'lower-case-p input) #'string-downcase #'identity))
(:invert (lambda (output)
           (multiple-value-bind (lower upper) (determine-case output)
             (cond ((and lower upper) output)
                   (lower (string-upcase output))
                   (upper (string-downcase output))
                   (t output))))))
(:downcase (if (some #'upper-case-p input) #'string-upcase #'identity))
(:preserve #'identity)))

(defun determine-case (string)
  "Return two booleans LOWER and UPPER indicating whether STRING
contains lower or upper case characters."
  (values (some #'lower-case-p string)
          (some #'upper-case-p string)))

```

15.3 Compound-prefix matching

```

(defun compound-prefix-match (prefix target)
  "Return true if PREFIX is a compound-prefix of TARGET.
Viewing each of PREFIX and TARGET as a series of substrings delimited
by hyphens, if each substring of PREFIX is a prefix of the
corresponding substring in TARGET then we call PREFIX a
compound-prefix of TARGET."

```

Examples:

```

\ (compound-prefix-match "foo\" "foobar\") => t
\ (compound-prefix-match "m--b\" "multiple-value-bind\") => t
\ (compound-prefix-match "m-v-c\" "multiple-value-bind\") => NIL"
(declare (type simple-string prefix target))
(loop for ch across prefix
      with tpos = 0
      always (and (< tpos (length target))
                 (if (char= ch #\-)
                     (setf tpos (position #\_- target :start tpos))
                     (char= ch (aref target tpos))))
      do (incf tpos)))

```

```

(defun prefix-match-p (prefix string)
  "Return true if PREFIX is a prefix of STRING."
  (not (mismatch prefix string :end2 (min (length string) (length prefix)))))

```

15.4 Extending the input string by completion

```

(defun longest-completion (completions)
  "Return the longest prefix for all COMPLETIONS.
COMPLETIONS is a list of strings."
  (untokenize-completion
   (mapcar #'longest-common-prefix
           (transpose-lists (mapcar #'tokenize-completion completions)))))

```

```

(defun tokenize-completion (string)
  "Return all substrings of STRING delimited by #\-"
  (loop with end
        for start = 0 then (1+ end)
        until (> start (length string))
        do (setq end (or (position #\ - string :start start) (length string)))
        collect (subseq string start end)))

(defun untokenize-completion (tokens)
  (format nil "~{~A^~}" tokens))

(defun longest-common-prefix (strings)
  "Return the longest string that is a common prefix of STRINGS."
  (if (null strings)
      ""
      (flet ((common-prefix (s1 s2)
              (let ((diff-pos (mismatch s1 s2)))
                (if diff-pos (subseq s1 0 diff-pos) s1))))
        (reduce #'common-prefix strings))))

(defun transpose-lists (lists)
  "Turn a list-of-lists on its side.
  If the rows are of unequal length, truncate uniformly to the shortest.

  For example:
  \ (transpose-lists '((ONE TWO THREE) (1 2)))
  => ((ONE 1) (TWO 2))"
  ;; A cute function from PAIP p.574
  (if lists (apply #'mapcar #'list lists)))

```

15.5 Completion Tests

```

(defpackage :swank-completion-test
  (:use))

(let ((*readtable* (copy-readtable *readtable*))
      (p (find-package :swank-completion-test)))
  (intern "foo" p)
  (intern "Foo" p)
  (intern "FOO" p)
  (setf (readtable-case *readtable*) :invert)
  (flet ((names (prefix)
          (sort (mapcar #'symbol-name
                      (find-matching-symbols prefix p nil #'prefix-match-p))
                #'string<)))
    (assert (equal '("FOO") (names "f")))
    (assert (equal '("Foo" "foo") (names "F")))
    (assert (equal '("Foo") (names "Fo")))
    (assert (equal '("foo") (names "FO")))))

```

16 Fuzzy completion

```
(defslimefun fuzzy-completions (string default-package-name &optional limit)
  "Return an (optionally limited to LIMIT best results) list of
  fuzzy completions for a symbol designator STRING. The list will
  be sorted by score, most likely match first.
```

The result is a list of completion objects, where a completion object is:

```
(COMPLETED-STRING SCORE (&rest CHUNKS) FLAGS)
```

where a CHUNK is a description of a matched string of characters:

```
(OFFSET STRING)
```

and FLAGS is a list of keywords describing properties of the symbol.

For example, the top result for completing `"mvb"` in a package that uses COMMON-LISP would be something like:

```
(\"multiple-value-bind\" 42.391666 ((0 \"mul\") (9 \"v\") (15 \"b\"))
 (:FBOUNDP :MACRO))
```

If STRING is package qualified the result list will also be qualified. If string is non-qualified the result strings are also not qualified and are considered relative to DEFAULT-PACKAGE-NAME.

Which symbols are candidates for matching depends on the symbol designator's format. The cases are as follows:

FOO - Symbols accessible in the buffer package.

PKG:FOO - Symbols external in package PKG.

PKG::FOO - Symbols accessible in package PKG."

```
(fuzzy-completion-set string default-package-name limit))
```

```
(defun convert-fuzzy-completion-result (result converter
                                         internal-p package-name)
```

"Converts a result from the fuzzy completion core into something that emacs is expecting. Converts symbols to strings, fixes case issues, and adds information describing if the symbol is :bound, :fbound, a :class, a :macro, a :generic-function, a :special-operator, or a :package."

```
(destructuring-bind (symbol-or-name score chunks) result
```

```
(multiple-value-bind (name added-length)
```

```
(format-completion-result
```

```
(funcall converter
```

```
(if (symbolp symbol-or-name)
```

```
(symbol-name symbol-or-name)
```

```
symbol-or-name))
```

```
internal-p package-name)
```

```
(list name score
```

```
(mapcar
```

```

#' (lambda (chunk)
  ;; fix up chunk positions to account for possible
  ;; added package identifier
  (list (+ added-length (first chunk))
        (second chunk)))
chunks)
(loop for flag in '(:boundp :fboundp :generic-function
                  :class :macro :special-operator
                  :package)
  if (if (symbolp symbol-or-name)
        (case flag
          (:boundp (boundp symbol-or-name))
          (:fboundp (fboundp symbol-or-name))
          (:class (find-class symbol-or-name nil))
          (:macro (macro-function symbol-or-name))
          (:special-operator
               (special-operator-p symbol-or-name))
          (:generic-function
               (typep (ignore-errors (fdefinition symbol-or-name))
                      'generic-function)))
        (case flag
          (:package (stringp symbol-or-name)
                    ;; KLUDGE: depends on internal
                    ;; knowledge that packages are
                    ;; brought up from the bowels of
                    ;; the completion algorithm as
                    ;; strings!
                    )))
    collect flag))))

(defun fuzzy-completion-set (string default-package-name &optional limit)
  "Prepares list of completion objects, sorted by SCORE, of fuzzy
  completions of STRING in DEFAULT-PACKAGE-NAME. If LIMIT is set,
  only the top LIMIT results will be returned."
  (multiple-value-bind (name package-name package internal-p)
    (parse-completion-arguments string default-package-name)
    (let* ((symbols (and package
                          (fuzzy-find-matching-symbols name
                                                         package
                                                         (and (not internal-p)
                                                             package-name))))
           (packs (and (not package-name)
                       (fuzzy-find-matching-packages name)))
           (converter (output-case-converter name))
           (results
            (sort (mapcar #'(lambda (result)
                              (convert-fuzzy-completion-result
                               result converter internal-p package-name))
                      (nconc symbols packs))
                  #'> :key #'second))))

```

```

    (when (and limit
            (> limit 0)
            (< limit (length results)))
        (setf (cdr (nthcdr (1- limit) results)) nil))
    results)))

(defun fuzzy-find-matching-symbols (string package external)
  "Return a list of symbols in PACKAGE matching STRING using the
  fuzzy completion algorithm.  If EXTERNAL is true, only external
  symbols are returned."
  (let ((completions '())
        (converter (output-case-converter string)))
    (flet ((symbol-match (symbol)
            (and (or (not external)
                    (symbol-external-p symbol package))
                 (compute-highest-scoring-completion
                  string (funcall converter (symbol-name symbol)) #'char=))))
      (do-symbols (symbol package)
        (multiple-value-bind (result score) (symbol-match symbol)
          (when result
            (push (list symbol score result) completions))))))
    (remove-duplicates completions :key #'first)))

(defun fuzzy-find-matching-packages (name)
  "Return a list of package names matching NAME using the fuzzy
  completion algorithm."
  (let ((converter (output-case-converter name)))
    (loop for package in (list-all-packages)
          for package-name = (concatenate 'string
                                         (funcall converter
                                          (package-name package))
                                         ":")
          for (result score) = (multiple-value-list
                               (compute-highest-scoring-completion
                                name package-name #'char=))
          if result collect (list package-name score result)))

(defun slimefun fuzzy-completion-selected (original-string completion)
  "This function is called by Slime when a fuzzy completion is
  selected by the user.  It is for future expansion to make
  testing, say, a machine learning algorithm for completion scoring
  easier.

  ORIGINAL-STRING is the string the user completed from, and
  COMPLETION is the completion object (see docstring for
  SWANK:FUZZY-COMPLETIONS) corresponding to the completion that the
  user selected."
  (declare (ignore original-string completion))
  nil)

```

16.1 Fuzzy completion core

```
(defparameter *fuzzy-recursion-soft-limit* 30
  "This is a soft limit for recursion in
  RECURSIVELY-COMPUTE-MOST-COMPLETIONS. Without this limit,
  completing a string such as \"ZZZZZ\" with a symbol named
  \"ZZZZZZZZZZZZZZZZZZZZZZ\" will result in explosive recursion to
  find all the ways it can match.
```

Most natural language searches and symbols do not have this problem -- this is only here as a safeguard.")

```
(defun compute-highest-scoring-completion (short full test)
  "Finds the highest scoring way to complete the abbreviation
  SHORT onto the string FULL, using TEST as a equality function for
  letters. Returns two values: The first being the completion
  chunks of the high scorer, and the second being the score."
  (let* ((scored-results
          (mapcar #'(lambda (result)
                     (cons (score-completion result short full) result))
                 (compute-most-completions short full test)))
        (winner (first (sort scored-results #'> :key #'first))))
    (values (rest winner) (first winner))))

(defun compute-most-completions (short full test)
  "Finds most possible ways to complete FULL with the letters in SHORT.
  Calls RECURSIVELY-COMPUTE-MOST-COMPLETIONS recursively. Returns
  a list of (&rest CHUNKS), where each CHUNKS is a description of
  how a completion matches."
  (let ((*all-chunks* nil))
    (declare (special *all-chunks*))
    (recursively-compute-most-completions short full test 0 0 nil nil nil t)
    *all-chunks*))
```

```
(defun recursively-compute-most-completions
  (short full test
   short-index initial-full-index
   chunks current-chunk current-chunk-pos
   recurse-p)
  "Recursively (if RECURSE-P is true) find /most/ possible ways
  to fuzzily map the letters in SHORT onto FULL, with TEST being a
  function to determine if two letters match.
```

A chunk is a list of elements that have matched consecutively. When consecutive matches stop, it is coerced into a string, paired with the starting position of the chunk, and pushed onto CHUNKS.

Whenever a letter matches, if RECURSE-P is true, RECURSIVELY-COMPUTE-MOST-COMPLETIONS calls itself with a position

one index ahead, to find other possibly higher scoring possibilities. If there are less than *FUZZY-RECURSION-SOFT-LIMIT* results in *ALL-CHUNKS* currently, this call will also recurse.

Once a word has been completely matched, the chunks are pushed onto the special variable *ALL-CHUNKS* and the function returns."

```
(declare (special *all-chunks*))
(flet ((short-cur ()
  "Returns the next letter from the abbreviation, or NIL
  if all have been used."
  (if (= short-index (length short))
      nil
      (aref short short-index)))
  (add-to-chunk (char pos)
  "Adds the CHAR at POS in FULL to the current chunk,
  marking the start position if it is empty."
  (unless current-chunk
    (setf current-chunk-pos pos))
  (push char current-chunk))
  (collect-chunk ()
  "Collects the current chunk to CHUNKS and prepares for
  a new chunk."
  (when current-chunk
    (push (list current-chunk-pos
                (coerce (reverse current-chunk) 'string)) chunks)
    (setf current-chunk nil
          current-chunk-pos nil))))
;; If there's an outstanding chunk coming in collect it. Since
;; we're recursively called on skipping an input character, the
;; chunk can't possibly continue on.
(when current-chunk (collect-chunk))
(do ((pos initial-full-index (1+ pos)))
  ((= pos (length full)))
  (let ((cur-char (aref full pos)))
    (if (and (short-cur)
             (funcall test cur-char (short-cur)))
        (progn
          (when recurse-p
            ;; Try other possibilities, limiting insanely deep
            ;; recursion somewhat.
            (recursively-compute-most-completions
             short full test short-index (1+ pos)
             chunks current-chunk current-chunk-pos
             (not (> (length *all-chunks*)
                     *fuzzy-recursion-soft-limit*))))
          (incf short-index)
          (add-to-chunk cur-char pos))
        (collect-chunk))))
(collect-chunk)
```

```

;; If we've exhausted the short characters we have a match.
(if (short-cur)
    nil
    (let ((rev-chunks (reverse chunks)))
        (push rev-chunks *all-chunks*)
        rev-chunks))))

```

16.2 Fuzzy completion scoring

```

(defparameter *fuzzy-completion-symbol-prefixes* "+-%&?<"
  "Letters that are likely to be at the beginning of a symbol.
  Letters found after one of these prefixes will be scored as if
  they were at the beginning of ths symbol.")
(defparameter *fuzzy-completion-symbol-suffixes* "+->"
  "Letters that are likely to be at the end of a symbol.
  Letters found before one of these suffixes will be scored as if
  they were at the end of the symbol.")
(defparameter *fuzzy-completion-word-separators* "-/."
  "Letters that separate different words in symbols. Letters
  after one of these symbols will be scores more highly than other
  letters.")

(defun score-completion (completion short full)
  "Scores the completion chunks COMPLETION as a completion from
  the abbreviation SHORT to the full string FULL. COMPLETION is a
  list like:
  ((0 \"mul\") (9 \"v\") (15 \"b\"))
  Which, if SHORT were \"mulvb\" and full were \"multiple-value-bind\",
  would indicate that it completed as such (completed letters
  capitalized):
  MULtiple-Value-Bind

```

Letters are given scores based on their position in the string. Letters at the beginning of a string or after a prefix letter at the beginning of a string are scored highest. Letters after a word separator such as #\ are scored next highest. Letters at the end of a string or before a suffix letter at the end of a string are scored medium, and letters anywhere else are scored low.

If a letter is directly after another matched letter, and its intrinsic value in that position is less than a percentage of the previous letter's value, it will use that percentage instead.

Finally, a small scaling factor is applied to favor shorter matches, all other things being equal."

```

(labels ((at-beginning-p (pos)
          (= pos 0))
         (after-prefix-p (pos)
          (and (= pos 1)

```

```

        (find (aref full 0) *fuzzy-completion-symbol-prefixes*))
(word-separator-p (pos)
  (find (aref full pos) *fuzzy-completion-word-separators*))
(after-word-separator-p (pos)
  (find (aref full (1- pos)) *fuzzy-completion-word-separators*))
(at-end-p (pos)
  (= pos (1- (length full))))
(before-suffix-p (pos)
  (and (= pos (- (length full) 2))
        (find (aref full (1- (length full)))
                *fuzzy-completion-symbol-suffixes*)))
(score-or-percentage-of-previous (base-score pos chunk-pos)
  (if (zerop chunk-pos)
      base-score
      (max base-score
            (* (score-char (1- pos) (1- chunk-pos)) 0.85))))
(score-char (pos chunk-pos)
  (score-or-percentage-of-previous
   (cond ((at-beginning-p pos) 10)
         ((after-prefix-p pos) 10)
         ((word-separator-p pos) 1)
         ((after-word-separator-p pos) 8)
         ((at-end-p pos) 6)
         ((before-suffix-p pos) 6)
         (t 1))
   pos chunk-pos))
(score-chunk (chunk)
  (loop for chunk-pos below (length (second chunk))
        for pos from (first chunk)
        summing (score-char pos chunk-pos)))
(let* ((chunk-scores (mapcar #'score-chunk completion))
      (length-score (/ 10.0 (1+ (- (length full) (length short))))))
  (values
   (+ (reduce #'+ chunk-scores) length-score)
   (list (mapcar #'list chunk-scores completion) length-score))))

(defun highlight-completion (completion full)
  "Given a chunk definition COMPLETION and the string FULL,
HIGHLIGHT-COMPLETION will create a string that demonstrates where
the completion matched in the string. Matches will be
capitalized, while the rest of the string will be lower-case."
  (let ((highlit (nstring-downcase (copy-seq full))))
    (dolist (chunk completion)
      (setf highlit (nstring-upcase highlit
                                   :start (first chunk)
                                   :end (+ (first chunk)
                                           (length (second chunk))))))
    highlit))

(defun format-fuzzy-completions (winners)

```

```

"Given a list of completion objects such as on returned by
FUZZY-COMPLETIONS, format the list into user-readable output."
(let ((max-len
      (loop for winner in winners maximizing (length (first winner))))
      (loop for (sym score result) in winners do
        (format t "~&~VA score ~8,2F ~A"
                 max-len (highlight-completion result sym) score result))))

```

17 Documentation

```

(defslimefun apropos-list-for-emacs (name &optional external-only
                                    case-sensitive package)
  "Make an apropos search for Emacs.
The result is a list of property lists."
  (let ((package (if package
                     (or (find-package (string-to-package-designator package))
                         (error "No such package: ~S" package))))
        (mapcan (listify #'briefly-describe-symbol-for-emacs)
                 (sort (remove-duplicates
                       (apropos-symbols name external-only case-sensitive package))
                       #'present-symbol-before-p))))

(defun string-to-package-designator (string)
  "Return a package designator made from STRING.
Uses READ to case-convert STRING."
  (let ((*package* *swank-io-package*))
    (read-from-string string)))

(defun briefly-describe-symbol-for-emacs (symbol)
  "Return a property list describing SYMBOL.
Like 'describe-symbol-for-emacs' but with at most one line per item."
  (flet ((first-line (string)
          (let ((pos (position #\newline string)))
            (if (null pos) string (subseq string 0 pos))))
        (let ((desc (map-if #'stringp #'first-line
                           (describe-symbol-for-emacs symbol))))
          (if desc
              (list* :designator (to-string symbol) desc))))))

(defun map-if (test fn &rest lists)
  "Like (mapcar FN . LISTS) but only call FN on objects satisfying TEST.
Example:
\ (map-if #'oddp #'- '(1 2 3 4 5)) => (-1 2 -3 4 -5)"
  (declare (type function test fn))
  (apply #'mapcar
         (lambda (x) (if (funcall test x) (funcall fn x) x))
         lists))

(defun listify (f)

```

```

"Return a function like F, but which returns any non-null value
wrapped in a list."
(declare (type function f))
(lambda (x)
  (let ((y (funcall f x)))
    (and y (list y))))

(defun present-symbol-before-p (a b)
  "Return true if A belongs before B in a printed summary of symbols.
Sorted alphabetically by package name and then symbol name, except
that symbols accessible in the current package go first."
  (flet ((accessible (s)
          (find-symbol (symbol-name s) *buffer-package*)))
    (cond ((and (accessible a) (accessible b))
           (string< (symbol-name a) (symbol-name b)))
          ((accessible a) t)
          ((accessible b) nil)
          (t
           (string< (package-name (symbol-package a))
                    (package-name (symbol-package b)))))))

(let ((regex-hash (make-hash-table :test #'equal)))
  (defun compiled-regex (regex-string)
    (or (gethash regex-string regex-hash)
        (setf (gethash regex-string regex-hash)
              (if (zerop (length regex-string))
                  (lambda (s) (check-type s string) t)
                  (compile nil (nregex:regex-compile regex-string))))))

(defun apropos-matcher (string case-sensitive package external-only)
  (let* ((case-modifier (if case-sensitive #'string #'string-upcase))
        (regex (compiled-regex (funcall case-modifier string))))
    (lambda (symbol)
      (and (not (keywordp symbol))
           (if package (eq (symbol-package symbol) package) t)
           (if external-only (symbol-external-p symbol) t)
           (funcall regex (funcall case-modifier symbol))))))

(defun apropos-symbols (string external-only case-sensitive package)
  (let ((result '()))
    (matchp (apropos-matcher string case-sensitive package external-only))
    (with-package-iterator (next (or package (list-all-packages))
                               :external :internal)
      (loop
        (multiple-value-bind (morep symbol) (next)
          (cond ((not morep)
                 (return))
                ((funcall matchp symbol)
                 (push symbol result))))))
    result))

```

```

(defun call-with-describe-settings (fn)
  (let ((*print-readably* nil))
    (funcall fn)))

(defmacro with-describe-settings ((&rest _) &body body)
  (declare (ignore _))
  `(call-with-describe-settings (lambda () ,@body)))

(defun describe-to-string (object)
  (with-describe-settings ()
    (with-output-to-string (*standard-output*)
      (describe object))))

(defslimefun describe-symbol (symbol-name)
  (with-buffer-syntax ()
    (describe-to-string (parse-symbol-or-lose symbol-name))))

(defslimefun describe-function (name)
  (with-buffer-syntax ()
    (let ((symbol (parse-symbol-or-lose name)))
      (describe-to-string (or (macro-function symbol)
                              (symbol-function symbol))))))

(defslimefun describe-definition-for-emacs (name kind)
  (with-buffer-syntax ()
    (with-describe-settings ()
      (with-output-to-string (*standard-output*)
        (describe-definition (parse-symbol-or-lose name) kind))))))

(defslimefun documentation-symbol (symbol-name &optional default)
  (with-buffer-syntax ()
    (multiple-value-bind (sym foundp) (parse-symbol symbol-name)
      (if foundp
          (let ((vdoc (documentation sym 'variable))
                (fdoc (documentation sym 'function)))
            (or (and (or vdoc fdoc)
                    (concatenate 'string
                                 fdoc
                                 (and vdoc fdoc '(#\Newline #\Newline))
                                 vdoc))
                default))
          default))))))

```

18 Package Commands

```

(defslimefun list-all-package-names (&optional include-nicknames)
  "Return a list of all package names.
Include the nicknames if INCLUDE-NICKNAMES is true."

```

```
(loop for package in (list-all-packages)
      collect (package-name package)
      when include-nicknames append (package-nicknames package)))
```

19 Tracing

```
;; Use eval for the sake of portability...
(defun tracedp (fspec)
  (member fspec (eval `(trace))))

(defslimefun swank-toggle-trace (spec-string)
  (let ((spec (from-string spec-string)))
    (cond ((consp spec) ; handle complicated cases in the backend
           (toggle-trace spec))
          ((tracedp spec)
           (eval `(untrace ,spec))
           (format nil "~S is now untraced." spec))
          (t
           (eval `(trace ,spec))
           (format nil "~S is now traced." spec)))))

(defslimefun untrace-all ()
  (untrace))
```

20 Undefing

```
(defslimefun undefine-function (fname-string)
  (let ((fname (from-string fname-string)))
    (format nil "~S" (fmakunbound fname))))
```

21 Profiling

```
((defun profiledp (fspec)
  (member fspec (profiled-functions)))

(defslimefun toggle-profile-fdefinition (fname-string)
  (let ((fname (from-string fname-string)))
    (cond ((profiledp fname)
           (unprofile fname)
           (format nil "~S is now unprofiled." fname))
          (t
           (profile fname)
           (format nil "~S is now profiled." fname)))))
```

22 Source Locations

```
(defslimefun find-definitions-for-emacs (name)
```

```

"Return a list ((DSPEC LOCATION) ...) of definitions for NAME.
DSPEC is a string and LOCATION a source location. NAME is a string."
(multiple-value-bind (sexp error)
  (ignore-errors (values (from-string name))))
(cond (error '())
      (t (loop for (dspec loc) in (find-definitions sexp)
                collect (list (to-string dspec) loc))))))

(defun alistify (list key test)
  "Partition the elements of LIST into an alist. KEY extracts the key
from an element and TEST is used to compare keys."
  (declare (type function key))
  (let ((alist '()))
    (dolist (e list)
      (let* ((k (funcall key e))
             (probe (assoc k alist :test test)))
        (if probe
            (push e (cdr probe))
            (push (cons k (list e)) alist))))
    alist))

(defun location-position< (pos1 pos2)
  (cond ((and (position-p pos1) (position-p pos2))
        (< (position-pos pos1)
            (position-pos pos2)))
        (t nil)))

(defun partition (list test key)
  (declare (type function test key))
  (loop for e in list
        if (funcall test (funcall key e)) collect e into yes
        else collect e into no
        finally (return (values yes no))))

(defstruct (xref (:conc-name xref.)
                (:type list))
  dspec location)

(defun location-valid-p (location)
  (eq (car location) :location))

(defun xref-buffer (xref)
  (location-buffer (xref.location xref)))

(defun xref-position (xref)
  (location-buffer (xref.location xref)))

(defun group-xrefs (xrefs)
  "Group XREFS, a list of the form ((DSPEC LOCATION) ...) by location.
The result is a list of the form ((LOCATION . ((DSPEC . LOCATION) ...)) ...)."

```

```

(multiple-value-bind (resolved errors)
  (partition xrefs #'location-valid-p #'xref.location)
  (let ((alist (alistify resolved #'xref-buffer #'equal)))
    (append
      (loop for (buffer . list) in alist
        collect (cons (second buffer)
          (mapcar (lambda (xref)
            (cons (to-string (xref.dspec xref))
              (xref.location xref)))
            (sort list #'location-position<
              :key #'xref-position))))))
      (if errors
        (list (cons "Unresolved"
          (mapcar (lambda (xref)
            (cons (to-string (xref.dspec xref))
              (xref.location xref)))
            errors))))))))))

(defslimefun xref (type symbol-name)
  (let ((symbol (parse-symbol-or-lose symbol-name *buffer-package*)))
    (group-xrefs
      (ecase type
        (:calls (who-calls symbol))
        (:calls-who (calls-who symbol))
        (:references (who-references symbol))
        (:binds (who-binds symbol))
        (:sets (who-sets symbol))
        (:macroexpands (who-macroexpands symbol))
        (:specializes (who-specializes symbol))
        (:callers (list-callers symbol))
        (:callees (list-callees symbol))))))

```

23 Inspecting

```

(defun common-separated-spec (list &optional (callback (lambda (v)
  '(:value ,v))))
  (butlast
    (loop
      for i in list
      collect (funcall callback i)
      collect ", ")))

(defun inspector-princ (list)
  "Like princ-to-string, but don't rewrite (function foo) as #'foo.
Do NOT pass circular lists to this function."
  (let ((*print-pprint-dispatch* (copy-pprint-dispatch)))
    (set-pprint-dispatch '(cons (member function)) nil)
    (princ-to-string list)))

```

```

(defmethod inspect-for-emacs ((object cons) inspector)
  (declare (ignore inspector))
  (if (consp (cdr object))
      (inspect-for-emacs-list object)
      (inspect-for-emacs-simple-cons object)))

(defun inspect-for-emacs-simple-cons (cons)
  (values "A cons cell."
        (label-value-line*
         ('car (car cons))
         ('cdr (cdr cons)))))

(defun inspect-for-emacs-list (list)
  (let ((maxlen 40))
    (multiple-value-bind (length tail) (safe-length list)
      (flet ((frob (title list)
              (let ((lines
                    (do ((i 0 (1+ i))
                        (l list (cdr l))
                        (a '() (cons (label-value-line i (car l)) a)))
                    ((not (consp l))
                     (let ((a (if (null l)
                                   a
                                   (cons (label-value-line :tail l) a))))
                      (reduce #'append (reverse a) :from-end t))))))
              (values title (append '("Elements:" (:newline)) lines))))))

    (cond ((not length) ; circular
           (frob "A circular list."
                 (cons (car list)
                       (ldiff (cdr list) list))))
          ((and (<= length maxlen) (not tail))
           (frob "A proper list." list))
          (tail
           (frob "An improper list." list))
          (t
           (frob "A proper list." list))))))

;; (inspect-for-emacs-list '#1=(a #1# . #1# ))

(defun safe-length (list)
  "Similar to 'list-length', but avoid errors on improper lists.
Return two values: the length of the list and the last cdr.
NIL is returned if the list is circular."
  (do ((n 0 (+ n 2)) ;Counter.
      (fast list (caddr fast)) ;Fast pointer: leaps by 2.
      (slow list (cdr slow))) ;Slow pointer: leaps by 1.
      (nil)
    (cond ((null fast) (return (values n nil)))
          ((not (consp fast)) (return (values n fast))))))

```

```

((null (cdr fast)) (return (values (1+ n) (cdr fast))))
((and (eq fast slow) (> n 0)) (return nil))
((not (consp (cdr fast))) (return (values (1+ n) (cdr fast))))))

(defmethod inspect-for-emacs ((ht hash-table) inspector)
  (declare (ignore inspector))
  (values "A hash table."
    (append
      (label-value-line*
        ("Count" (hash-table-count ht))
        ("Size" (hash-table-size ht))
        ("Test" (hash-table-test ht))
        ("Rehash size" (hash-table-rehash-size ht))
        ("Rehash threshold" (hash-table-rehash-threshold ht)))
      `("Contents: " (:newline))
      (loop for key being the hash-keys of ht
            for value being the hash-values of ht
            append `(:value ,key) " = " (:value ,value) (:newline))))))

(defmethod inspect-for-emacs ((array array) inspector)
  (declare (ignore inspector))
  (values "An array."
    (append
      (label-value-line*
        ("Dimensions" (array-dimensions array))
        ("Its element type is" (array-element-type array))
        ("Total size" (array-total-size array))
        ("Adjustable" (adjustable-array-p array)))
      (when (array-has-fill-pointer-p array)
        (label-value-line "Fill pointer" (fill-pointer array)))
      `("Contents:" (:newline))
      (loop for i below (array-total-size array)
            append (label-value-line i (row-major-aref array i))))))

(defmethod inspect-for-emacs ((char character) inspector)
  (declare (ignore inspector))
  (values "A character."
    (append
      (label-value-line*
        ("Char code" (char-code char))
        ("Lower cased" (char-downcase char))
        ("Upper cased" (char-upcase char)))
      (if (get-macro-character char)
        `("In the current readtable ("
          (:value ,*readtable*) ") it is a macro character: "
          (:value ,(get-macro-character char))))))

(defun docstring-ispec (label object kind)
  "Return a inspector spec if OBJECT has a docstring of of kind KIND."
  (let ((docstring (documentation object kind)))

```

```

(cond ((not docstring) nil)
      ((< (+ (length label) (length docstring))
          75)
        (list label ": " docstring '(:newline)))
      (t
        (list label ": " '(:newline) " " docstring '(:newline))))))

(defmethod inspect-for-emacs ((symbol symbol) inspector)
  (declare (ignore inspector))
  (let ((package (symbol-package symbol)))
    (multiple-value-bind (_symbol status)
      (and package (find-symbol (string symbol) package))
      (declare (ignore _symbol))
      (values
       "A symbol."
       (append
        (label-value-line "Its name is" (symbol-name symbol))
        ;;
        ;; Value
        (cond ((boundp symbol)
               (label-value-line (if (constantp symbol)
                                     "It is a constant of value"
                                     "It is a global variable bound to")
                                (symbol-value symbol)))
              (t '("It is unbound." (:newline))))
        (docstring-ispec "Documentation" symbol 'variable)
        (multiple-value-bind (expansion definedp) (macroexpand symbol)
          (if definedp
              (label-value-line "It is a symbol macro with expansion"
                                expansion)))
        ;;
        ;; Function
        (if (fboundp symbol)
            (append (if (macro-function symbol)
                        `("It a macro with macro-function: "
                          (:value ,(macro-function symbol)))
                        `("It is a function: "
                          (:value ,(symbol-function symbol))))
                    `(" " (:action "[make funbound]"
                                   ,(lambda () (fmakunbound symbol))))
                    `(:newline)))
            `("It has no function value." (:newline)))
        (docstring-ispec "Function Documentation" symbol 'function)
        (if (compiler-macro-function symbol)
            (label-value-line "It also names the compiler macro"
                              (compiler-macro-function symbol)))
        (docstring-ispec "Compiler Macro Documentation"
                          symbol 'compiler-macro)
        ;;
        ;; Package

```

```

      (if package
        `("It is " ,(string-downcase (string status))
          " to the package: "
          (:value ,package ,(package-name package))
          ,@(if (eq :internal status)
              `(:action " [export it]"
                ,(lambda () (export symbol package))))))
          (:newline))
        `("It is a non-interned symbol." (:newline)))
    ;;
    ;; Plist
    (label-value-line "Property list" (symbol-plist symbol))
    ;;
    ;; Class
    (if (find-class symbol nil)
      `("It names the class "
        (:value ,(find-class symbol) ,(string symbol))
        (:action " [remove]"
          ,(lambda () (setf (find-class symbol) nil)))
        (:newline)))
      ;;
      ;; More package
      (if (find-package symbol)
        (label-value-line "It names the package" (find-package symbol)))
      ))))

(defmethod inspect-for-emacs ((f function) inspector)
  (declare (ignore inspector))
  (values "A function."
    (append
      (label-value-line "Name" (function-name f))
      `("Its argument list is: "
        ,(inspector-princ (arglist f)) (:newline))
      (docstring-ispec "Documentation" f t)
      (if (function-lambda-expression f)
        (label-value-line "Lambda Expression"
          (function-lambda-expression f))))))

(defun method-specializers-for-inspect (method)
  "Return a \"pretty\" list of the method's specializers. Normal
specializers are replaced by the name of the class, eql
specializers are replaced by `(eql ,object).\"
  (mapcar (lambda (spec)
    (typecase spec
      (swank-mop:eql-specializer
        `(eql ,(swank-mop:eql-specializer-object spec)))
      (t (swank-mop:class-name spec))))
    (swank-mop:method-specializers method)))

(defun method-for-inspect-value (method)

```

```

>Returns a \"pretty\" list describing METHOD. The first element
of the list is the name of generic-function method is
specialized on, the second element is the method qualifiers,
the rest of the list is the method's specializers (as per
method-specializers-for-inspect).\"
(append (list (swank-mop:generic-function-name
              (swank-mop:method-generic-function method)))
        (swank-mop:method-qualifiers method)
        (method-specializers-for-inspect method)))

(defmethod inspect-for-emacs ((o standard-object) inspector)
  (declare (ignore inspector))
  (values "An object."
        `("Class: " (:value ,(class-of o))
          (:newline)
          "Slots:" (:newline)
          ,@(loop
              with direct-slots = (swank-mop:class-direct-slots (class-of o))
              for slot in (swank-mop:class-slots (class-of o))
              for slot-def = (or (find-if (lambda (a)
                                           ;; find the direct slot
                                           ;; with the same name
                                           ;; as SLOT (an
                                           ;; effective slot).
                                           (eql (swank-mop:slot-definition-
                                                (swank-mop:slot-definition-
                                                direct-slots)
                                                slot)
                                               slot)
                                         collect `(:value ,slot-def ,(inspector-princ (swank-mop:slot-
                                               collect " = "
                                               if (slot-boundp o (swank-mop:slot-definition-name slot-def))
                                                 collect `(:value ,(slot-value o (swank-mop:slot-definition-n
                                                 else
                                                 collect "#<unbound>\"
                                                 collect `(:newline))))))

(defvar *gf-method-getter* 'methods-by-applicability
  "This function is called to get the methods of a generic function.
The default returns the method sorted by applicability.
See 'methods-by-applicability'."))

(defun specializer< (specializer1 specializer2)
  "Return true if SPECIALIZER1 is more specific than SPECIALIZER2."
  (let ((s1 specializer1) (s2 specializer2) )
    (cond ((typep s1 'swank-mop:eql-specializer)
           (not (typep s2 'swank-mop:eql-specializer)))
          (t
           (flet ((cpl (class)
                   (and (swank-mop:class-finalized-p class)
                        (swank-mop:class-precedence-list class))))
             (and (swank-mop:class-finalized-p class)
                  (swank-mop:class-precedence-list class))))))

```

```

        (member s2 (cpl s1)))))))))

(defun methods-by-applicability (gf)
  "Return methods ordered by most specific argument types.

`method-specializer<' is used for sorting."
  ;; FIXME: argument-precedence-order and qualifiers are ignored.
  (let ((methods (copy-list (swank-mop:generic-function-methods gf))))
    (labels ((method< (meth1 meth2)
              (loop for s1 in (swank-mop:method-specializers meth1)
                    for s2 in (swank-mop:method-specializers meth2)
                    do (cond ((specializer< s2 s1) (return nil))
                              ((specializer< s1 s2) (return t))))))
      (stable-sort methods #'method<))))

(defun abbrev-doc (doc &optional (maxlen 80))
  "Return the first sentence of DOC, but not more than MAXLEN characters."
  (subseq doc 0 (min (1+ (or (position #\. doc) (1- maxlen)))
                    maxlen
                    (length doc))))

(defmethod inspect-for-emacs ((gf standard-generic-function) inspector)
  (declare (ignore inspector))
  (flet ((lv (label value) (label-value-line label value)))
    (values
     "A generic function."
     (append
      (lv "Name" (swank-mop:generic-function-name gf))
      (lv "Arguments" (swank-mop:generic-function-lambda-list gf))
      (docstring-ispec "Documentation" gf t)
      (lv "Method class" (swank-mop:generic-function-method-class gf))
      (lv "Method combination"
          (swank-mop:generic-function-method-combination gf))
      `("Methods: " (:newline))
      (loop for method in (funcall *gf-method-getter* gf) append
            `((:value ,method ,(inspector-princ
                               ;; drop the name of the GF
                               (cdr (method-for-inspect-value method))))
              (:action " [remove method]"
                      ,(let ((m method)) ; LOOP reassigns method
                          (lambda ()
                            (remove-method gf m))))
              (:newline)))))))))

(defmethod inspect-for-emacs ((method standard-method) inspector)
  (declare (ignore inspector))
  (values "A method."
         `("Method defined on the generic function "
           (:value ,(swank-mop:method-generic-function method)
                   ,(inspector-princ

```

```

                (swank-mop:generic-function-name
                  (swank-mop:method-generic-function method))))
(:newline)
,@(docstring-ispec "Documentation" method t)
"Lambda List: " (:value ,(swank-mop:method-lambda-list method))
(:newline)
"Specializers: " (:value ,(swank-mop:method-specializers method)
                        ,(inspector-princ (method-specializers-for
(:newline)
"Qualifiers: " (:value ,(swank-mop:method-qualifiers method))
(:newline)
"Method function: " (:value ,(swank-mop:method-function method))))))

(defmethod inspect-for-emacs ((class standard-class) inspector)
  (declare (ignore inspector))
  (values "A class."
    `("Name: " (:value ,(class-name class))
      (:newline)
      "Super classes: "
      ,@(common-seperated-spec (swank-mop:class-direct-superclasses class)
      (:newline)
      "Direct Slots: "
      ,@(common-seperated-spec
        (swank-mop:class-direct-slots class)
        (lambda (slot)
          `(:value ,slot ,(inspector-princ (swank-mop:slot-definition-na
(:newline)
      "Effective Slots: "
      ,@(if (swank-mop:class-finalized-p class)
        (common-seperated-spec
          (swank-mop:class-slots class)
          (lambda (slot)
            `(:value ,slot ,(inspector-princ
              (swank-mop:slot-definition-name slot))))))
        `("#<N/A (class not finalized)>"))
      (:newline)
      ,@(when (documentation class t)
        `("Documentation:" (:newline) ,(documentation class t) (:newlin
      "Sub classes: "
      ,@(common-seperated-spec (swank-mop:class-direct-subclasses class)
        (lambda (sub)
          `(:value ,sub ,(inspector-princ (class-n
(:newline)
      "Precedence List: "
      ,@(if (swank-mop:class-finalized-p class)
        (common-seperated-spec (swank-mop:class-precedence-list class)
          (lambda (class)
            `(:value ,class ,(inspector-princ (c
        `("#<N/A (class not finalized)>"))
      (:newline)

```

```

,@(when (swank-mop:specializer-direct-methods class)
  `("It is used as a direct specializer in the following methods:"
    ,@(loop
      for method in (sort (copy-list (swank-mop:specializer-direct-methods class))
                          #'string< :key (lambda (x)
                                           (symbol-name
                                            (let ((name (swank-mop:specializer-direct-method-name x))
                                                  (swank-mop:specializer-direct-method-name x))
                                              (if (symbolp name)
                                                  name))))))
      collect " "
      collect `(:value ,method ,(inspector-princ (method-for-inheritance method)))
      collect '(:newline)
      if (documentation method t)
      collect " Documentation: " and
      collect (abbrev-doc (documentation method t)) and
      collect '(:newline))))
"Prototype: " ,(if (swank-mop:class-finalized-p class)
  `(:value ,(swank-mop:class-prototype class))
  `("#<N/A (class not finalized)>"))))

(defmethod inspect-for-emacs ((slot swank-mop:standard-slot-definition) inspector)
  (declare (ignore inspector))
  (values "A slot."
    `("Name: " (:value ,(swank-mop:slot-definition-name slot))
      (:newline)
      ,@(when (swank-mop:slot-definition-documentation slot)
        `("Documentation:" (:newline)
          (:value ,(swank-mop:slot-definition-documentation slot))
          (:newline)))
      "Init args: " (:value ,(swank-mop:slot-definition-initargs slot))
      "Init form: " ,(if (swank-mop:slot-definition-initfunction slot)
        `(:value ,(swank-mop:slot-definition-initform slot))
        "#<unspecified>") (:newline)
      "Init function: " (:value ,(swank-mop:slot-definition-initfunction slot))
      (:newline))))

(defmethod inspect-for-emacs ((package package) inspector)
  (declare (ignore inspector))
  (let ((internal-symbols '())
        (external-symbols '()))
    (do-symbols (sym package)
      (when (eq package (symbol-package sym))
        (push sym internal-symbols)
        (multiple-value-bind (symbol status)
          (find-symbol (symbol-name sym) package)
          (declare (ignore symbol))
          (when (eql :external status)
            (push sym external-symbols))))))
    (setf internal-symbols (sort internal-symbols #'string-lessp)
          external-symbols (sort external-symbols #'string-lessp)))

```

```

(values "A package."
  `("Name: " (:value ,(package-name package))
    (:newline)
    "Nick names: " ,@(common-seperated-spec (sort (package-nicknames
    (:newline)
    ,@(when (documentation package t)
      `("Documentation:" (:newline)
        ,(documentation package t) (:newline)))
    "Use list: " ,@(common-seperated-spec (sort (package-use-list pack
      (lambda (pack)
        `(:value ,pack ,(inspecto
    (:newline)
    "Used by list: " ,@(common-seperated-spec (sort (package-used-by-
      (lambda (pack)
        `(:value ,pack ,(insp
    (:newline)
    ,(if (null external-symbols)
      "0 external symbols."
      `(:value ,external-symbols ,(format nil "~D external symbol~
    (:newline)
    ,(if (null internal-symbols)
      "0 internal symbols."
      `(:value ,internal-symbols ,(format nil "~D internal symbol~
    (:newline)
    ,(if (null (package-shadowing-symbols package))
      "0 shadowed symbols."
      `(:value ,(package-shadowing-symbols package)
        ,(format nil "~D shadowed symbol~:P." (length (pack

(defmethod inspect-for-emacs ((pathname pathname) inspector)
  (declare (ignore inspector))
  (values (if (wild-pathname-p pathname)
    "A wild pathname."
    "A pathname.")
    (append (label-value-line*
      ("Namestring" (namestring pathname))
      ("Host" (pathname-host pathname))
      ("Device" (pathname-device pathname))
      ("Directory" (pathname-directory pathname))
      ("Name" (pathname-name pathname))
      ("Type" (pathname-type pathname))
      ("Version" (pathname-version pathname)))
      (unless (or (wild-pathname-p pathname)
        (not (probe-file pathname)))
        (label-value-line "Truename" (truename pathname))))))

(defmethod inspect-for-emacs ((pathname logical-pathname) inspector)
  (declare (ignore inspector))
  (values "A logical pathname."
    (append

```

```

(label-value-line*
 ("Namestring" (namestring pathname))
 ("Physical pathname: " (translate-logical-pathname pathname)))
`("Host: " (pathname-host pathname)
  " (" (:value ,(logical-pathname-translations
              (pathname-host pathname)))
      "other translations)"
  (:newline))
(label-value-line*
 ("Directory" (pathname-directory pathname))
 ("Name" (pathname-name pathname))
 ("Type" (pathname-type pathname))
 ("Version" (pathname-version pathname))
 ("Truename" (if (not (wild-pathname-p pathname))
                 (probe-file pathname))))))

(defmethod inspect-for-emacs ((n number) inspector)
  (declare (ignore inspector))
  (values "A number." `("Value: " ,(princ-to-string n))))

(defmethod inspect-for-emacs ((i integer) inspector)
  (declare (ignore inspector))
  (values "A number."
    (append
     `((, (format nil "Value: ~D = #x~X = #o~O = #b~,,' ,8:B = ~E"
                  i i i i i)
       (:newline))
      (if (< -1 i char-code-limit)
          (label-value-line "Corresponding character" (code-char i))
          (label-value-line "Length" (integer-length i)))
      (ignore-errors
       (list "As time: "
             (multiple-value-bind (sec min hour date month year)
               (decode-universal-time i)
               (format nil "~4,'0D-~2,'0D-~2,'0DT~2,'0D:~2,'0D:~2,'0DZ"
                       year month date hour min sec))))))

(defmethod inspect-for-emacs ((c complex) inspector)
  (declare (ignore inspector))
  (values "A complex number."
    (label-value-line*
     ("Real part" (realpart c))
     ("Imaginary part" (imagpart c)))))

(defmethod inspect-for-emacs ((r ratio) inspector)
  (declare (ignore inspector))
  (values "A non-integer ratio."
    (label-value-line*
     ("Numerator" (numerator r))
     ("Denominator" (denominator r)))

```

```

        ("As float" (float r))))))

(defmethod inspect-for-emacs ((f float) inspector)
  (declare (ignore inspector))
  (multiple-value-bind (significand exponent sign) (decode-float f)
    (values "A floating point number."
            (append
              `("Scientific: " ,(format nil "~E" f) (:newline)
                "Decoded: "
                (:value ,sign) " * "
                (:value ,significand) " * "
                (:value ,(float-radix f)) "^" (:value ,exponent) (:newline))
              (label-value-line "Digits" (float-digits f))
              (label-value-line "Precision" (float-precision f))))))

(defvar *inspectee*)
(defvar *inspectee-parts*)
(defvar *inspectee-actions*)
(defvar *inspector-stack* '())
(defvar *inspector-history* (make-array 10 :adjustable t :fill-pointer 0))
(declare (type vector *inspector-history*))
(defvar *inspect-length* 30)

(defun reset-inspector ()
  (setq *inspectee* nil
        *inspector-stack* nil
        *inspectee-parts* (make-array 10 :adjustable t :fill-pointer 0)
        *inspectee-actions* (make-array 10 :adjustable t :fill-pointer 0)
        *inspector-history* (make-array 10 :adjustable t :fill-pointer 0)))

(defslimefun init-inspector (string)
  (with-buffer-syntax ()
    (reset-inspector)
    (inspect-object (eval (read-from-string string)))))

(defun print-part-to-string (value)
  (let ((string (to-string value))
        (pos (position value *inspector-history*)))
    (if pos
        (format nil "#~D=~A" pos string)
        string)))

(defun inspector-content-for-emacs (specs)
  (loop for part in specs collect
    (etypecase part
      (null ; XXX encourages sloppy programming
       nil)
      (string part)
      (cons (destructure-case part
              (:newline)

```

```

        (string #\newline))
        ( (:value obj &optional str)
          (value-part-for-emacs obj str))
        ( (:action label lambda)
          (action-part-for-emacs label lambda))))))

(defun assign-index (object vector)
  (let ((index (fill-pointer vector)))
    (vector-push-extend object vector)
    index))

(defun value-part-for-emacs (object string)
  (list :value
        (or string (print-part-to-string object))
        (assign-index object *inspectee-parts*)))

(defun action-part-for-emacs (label lambda)
  (list :action label (assign-index lambda *inspectee-actions*)))

(defun inspect-object (object &optional (inspector (make-default-inspector)))
  (push (setq *inspectee* object) *inspector-stack*)
  (unless (find object *inspector-history*)
    (vector-push-extend object *inspector-history*))
  (let ((*print-pretty* nil) ; print everything in the same line
        (*print-circle* t)
        (*print-readably* nil))
    (multiple-value-bind (title content)
      (inspect-for-emacs object inspector)
      (list :title title
            :type (to-string (type-of object))
            :content (inspector-content-for-emacs content))))))

(defslimefun inspector-nth-part (index)
  (aref *inspectee-parts* index))

(defslimefun inspect-nth-part (index)
  (with-buffer-syntax ()
    (inspect-object (inspector-nth-part index))))

(defslimefun inspector-call-nth-action (index)
  (funcall (aref *inspectee-actions* index)
    (inspect-object (pop *inspector-stack*)))

(defun inspector-pop ()
  "Drop the inspector stack and inspect the second element. Return
nil if there's no second element."
  (with-buffer-syntax ()
    (cond ((cdr *inspector-stack*)
           (pop *inspector-stack*)
           (inspect-object (pop *inspector-stack*)))
          (t nil))))

```

```

        (t nil))))))

(defslimefun inspector-next ()
  "Inspect the next element in the *inspector-history*."
  (with-buffer-syntax ()
    (let ((position (position *inspectee* *inspector-history*)))
      (cond ((= (1+ position) (length *inspector-history*))
             nil)
            (t (inspect-object (aref *inspector-history* (1+ position))))))))))

(defslimefun quit-inspector ()
  (reset-inspector)
  nil)

(defslimefun describe-inspectee ()
  "Describe the currently inspected object."
  (with-buffer-syntax ()
    (describe-to-string *inspectee*)))

(defslimefun inspect-in-frame (string index)
  (with-buffer-syntax ()
    (reset-inspector)
    (inspect-object (eval-in-frame (from-string string) index))))

(defslimefun inspect-current-condition ()
  (with-buffer-syntax ()
    (reset-inspector)
    (inspect-object *swank-debugger-condition*)))

(defslimefun inspect-frame-var (frame var)
  (with-buffer-syntax ()
    (reset-inspector)
    (inspect-object (frame-var-value frame var))))

```

24 Thread listing

```

(defvar *thread-list* ()
  "List of threads displayed in Emacs. We don't care about
synchronization issues (yet). There can only be one thread listing at
a time.")

(defslimefun list-threads ()
  "Return a list ((NAME DESCRIPTION) ...) of all threads."
  (setq *thread-list* (all-threads))
  (loop for thread in *thread-list*
        collect (list (thread-name thread)
                      (thread-status thread)
                      (thread-id thread))))

```

```

(defslimefun quit-thread-browser ()
  (setq *thread-list* nil))

(defun nth-thread (index)
  (nth index *thread-list*))

(defslimefun debug-nth-thread (index)
  (let ((connection *emacs-connection*))
    (interrupt-thread (nth-thread index)
      (lambda ()
        (with-connection (connection)
          (simple-break))))))

(defslimefun kill-nth-thread (index)
  (kill-thread (nth-thread index)))

(defslimefun start-swank-server-in-thread (index port-file-name)
  "Interrupt the INDEXth thread and make it start a swank server.
The server port is written to PORT-FILE-NAME."
  (interrupt-thread (nth-thread index)
    (lambda ()
      (start-server port-file-name :style nil))))

```

25 Class browser

```

(defun mop-helper (class-name fn)
  (let ((class (find-class class-name nil)))
    (if class
        (mapcar (lambda (x) (to-string (class-name x)))
                (funcall fn class))))))

(defslimefun mop (type symbol-name)
  "Return info about classes using mop.

When type is:
  :subclasses - return the list of subclasses of class.
  :superclasses - return the list of superclasses of class."
  (let ((symbol (parse-symbol symbol-name *buffer-package*)))
    (ecase type
      (:subclasses
       (mop-helper symbol #'swank-mop:class-direct-subclasses))
      (:superclasses
       (mop-helper symbol #'swank-mop:class-direct-superclasses)))))

```

26 Automatically synchronized state

Here we add hooks to push updates of relevant information to Emacs.

26.1 *FEATURES*

```
(defun sync-features-to-emacs ()
  "Update Emacs if any relevant Lisp state has changed."
  ;; FIXME: *slime-features* should be connection-local
  (unless (eq *slime-features* *features*)
    (setq *slime-features* *features*)
    (send-to-emacs (list :new-features (features-for-emacs)))))

(defun features-for-emacs ()
  "Return `*slime-features*' in a format suitable to send it to Emacs."
  *slime-features*)

(add-hook *pre-reply-hook* 'sync-features-to-emacs)
```

26.2 Indentation of macros

This code decides how macros should be indented (based on their arglists) and tells Emacs. A per-connection cache is used to avoid sending redundant information to Emacs – we just say what’s changed since last time.

The strategy is to scan all symbols, pick out the macros, and look for &body-arguments.

```
(defvar *configure-emacs-indentation* t
  "When true, automatically send indentation information to Emacs
after each command.")

(defslimefun update-indentation-information ()
  (perform-indentation-update *emacs-connection* t))

;; This function is for *PRE-REPLY-HOOK*.
(defun sync-indentation-to-emacs ()
  "Send any indentation updates to Emacs via CONNECTION."
  (when *configure-emacs-indentation*
    (let ((fullp (need-full-indentation-update-p *emacs-connection*)))
      (perform-indentation-update *emacs-connection* fullp))))

(defun need-full-indentation-update-p (connection)
  "Return true if the whole indentation cache should be updated.
This is a heuristic to avoid scanning all symbols all the time:
instead, we only do a full scan if the set of packages has changed."
  (set-difference (list-all-packages)
                  (connection.indentation-cache-packages connection)))

(defun perform-indentation-update (connection force)
  "Update the indentation cache in CONNECTION and update Emacs.
If FORCE is true then start again without considering the old cache."
  (let ((cache (connection.indentation-cache connection)))
    (when force (clrhash cache))
    (let ((delta (update-indentation/delta-for-emacs cache force))))
```

```

      (setf (connection.indentation-cache-packages connection)
            (list-all-packages))
      (unless (null delta)
        (send-to-emacs (list :indentation-update delta))))))

(defun update-indentation/delta-for-emacs (cache &optional force)
  "Update the cache and return the changes in a (SYMBOL . INDENT) list.
  If FORCE is true then check all symbols, otherwise only check symbols
  belonging to the buffer package."
  (let ((alist '()))
    (flet ((consider (symbol)
            (let ((indent (symbol-indentation symbol)))
              (when indent
                (unless (equal (gethash symbol cache) indent)
                  (setf (gethash symbol cache) indent)
                  (push (cons (string-downcase symbol) indent) alist)))))))
      (if force
          (do-all-symbols (symbol)
            (consider symbol))
          (do-symbols (symbol *buffer-package*)
            (when (eq (symbol-package symbol) *buffer-package*)
              (consider symbol))))))
    alist))

(defun package-names (package)
  "Return the name and all nicknames of PACKAGE in a list."
  (cons (package-name package) (package-nicknames package)))

(defun cl-symbol-p (symbol)
  "Is SYMBOL a symbol in the COMMON-LISP package?"
  (eq (symbol-package symbol) cl-package))

(defun known-to-emacs-p (symbol)
  "Return true if Emacs has special rules for indenting SYMBOL."
  (cl-symbol-p symbol))

(defun symbol-indentation (symbol)
  "Return a form describing the indentation of SYMBOL.
  The form is to be used as the 'common-lisp-indent-function' property
  in Emacs."
  (if (and (macro-function symbol)
           (not (known-to-emacs-p symbol)))
      (let ((arglist (arglist symbol)))
        (etypecase arglist
          ((member :not-available)
           nil)
          (list
           (list
            (macro-indentation arglist))))))
      nil))

```

