ILISP User Manual

A GNU Emacs Interface for Interacting with Lisp
Edition 0.14, September 1999
For ILISP Version 5.9.1

**by Todd Kaufmann, Chris McConnell, Ivan Vazquez,
Marco Antoniotti, Rick Campbell
and Paolo Amoroso**

This is edition 0.14 of the *ILISP User Manual* for ILISP Version 5.9.1, September 1999.

How to get the latest ILISP distribution.

ILISP is "free"; this means that everyone is free to use it and free to redistribute it on a free basis. ILISP is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of ILISP that they might get from you. The precise conditions appears following this section.

The easiest way to get a copy of ILISP is from someone else who has it. You need not ask for permission to do so, or tell any one else; just copy it.

If you do start using the package, please send mail to `majordomo@cons.org` to subscribe to the ILISP mailing list. (A message with `subscribe ilisp` in the body should suffice.)

Please send bugs, questions or suggestions for discussion to `ilisp@cons.org`. Message sent there will be distributed to all subscribers. To report a bug you can also switch to the buffer where the problem occurs, execute the `M-x ilisp-bug` command and follow the displayed instructions. See the file '`HISTORY`' for a list of known bugs and problems.

FTP and Web directions

You can get the distribution file, `ilisp-5.9.tar.gz` (`ilisp-5.9.zip`) via anonymous FTP or HTML at the following addresses:

- `ftp://ftp2.cons.org/pub/languages/lisp/ilisp/`, or
- `http://www2.cons.org:8000/ftp-area/ilisp/`

If you use a tty ftp client, just log in as 'anonymous'.

Please report any problems to the mailing list `ilisp@cons.org`.

Unpack and install:

% gzip -dc ilisp-5.9.tar.gz | tar xf -

or

% unzip ilisp-5.9.zip

See Chapter 1 [Installation], page 4.

Acknowledgements

ILISP replaces the standard inferior Lisp mode. ILISP is based on comint mode and derived from a number of different interfaces including Symbolics, CMU Common Lisp, and Thinking Machines.

There are many people that have taken the time to report bugs, make suggestions and even better send code to fix bugs or implement new features.

Thanks to Ben Hyde, Ben Wing, Bjorn Victor, Brian Dennis, Christian Lynbech, Christof Ullwer, Christopher Hoover, Chuck Irvine, Dan Pierson, David Bakash, David Braunegg, David Duff, David Gadbois, Erik Naggum, Frank Ritter, Fred White, Guido Bosch, Hans Chalupsky, Jamie Zawinski, Jason Trenouth, Jeffrey Mark Siskind, Jim Healy, Kimball Collins, Larry Hunter, Larry Stead, Mark Kantrowitz, Martin Atzmueller, Marty Hall, Matsuo Yoshihiro, Michael Ernst, Michael Kashket, Neil Smithline, Paolo Amoroso, Paul Fuqua (for the CMU-CL GC display code) Qiegang Long, Richard Harris, Richard Stallman, Rick Campbell, Robert P. Goldman, Scott Fahlman, Thomas M. Breuel, Tim Bradshaw, Tom Emerson, Yusuf Pisan, and many others for bug reports, suggestions and code. Our apologies to anyone we may have forgotten.

Special thanks to Todd Kaufmann for the texinfo file, work on bridge, epoch-pop and for really exercising everything.

Please send bug reports, fixes and extensions to `ilisp@cons.org` so that they can be merged into the master source.

```
--Chris McConnell                     1991-03-18
--Ivan Vazquez                        1993-06-27
--Marco Antoniotti and Rick Campbell  1996-10-25
--Marco Antoniotti and Paolo Amoroso  1999-08-19
```

Introduction

ILISP is an interface from GNU Emacs to an inferior Lisp. It has the following features:

- Runs under Emacs-18, Emacs-19, Emacs-20, and XEmacs-19.
- Support for multiple Common Lisp (including Allegro, CLISP and CMU), XLisp and Scheme dialects on multiple machines even at the same time.
- Dynamically sized pop-up windows that can be buried and scrolled from any window.
- Packages are properly handled including the distinction between exported and internal symbols.
- Synchronous, asynchronous or batch eval and compile of files, regions, definitions and sexps with optional switching and automatic calling.
- Arglist, documentation, describe, inspect and macroexpand.
- Completion of filename components and Lisp symbols including partial matches.
- Find source both with and without help from the inferior Lisp, including CLOS methods, multiple definitions and multiple files.
- Edit the callers of a function with and without help from the inferior Lisp.
- Trace/untrace a function.
- *M-q* ("Fill-paragraph") works properly on paragraphs in comments, strings and code.
- Find unbalanced parentheses.
- Super brackets.
- Handles editing, entering and indenting full Lisp expressions.
- Next, previous, and similar history mechanism compatible with comint.
- Handles Lisp errors.
- Uniform interface to Lisp debuggers.
- Result histories are maintained in the inferior Lisp.
- Does not create spurious symbols and handles case issues.
- Online manuals for ILISP and Common Lisp.

1 How to install ILISP

Installation of ILISP and some initialization of your computing environment are described in this chapter. Please read the following sections carefully before getting started with ILISP.

If ILISP has already been installed at your location, you can probably skip ahead to "Autoloading". See Section 1.3 [Autoloading], page 9.

Copy the ILISP distribution archive, e.g. `ilisp-X.Y.tar.gz`, to the location where you would like to install it. Next extract the archive (See [FTP and Web directions], page 1). You may need root privileges to perform these operations.

1.1 Configuration and compilation

Some configuration needs to be done before compiling the Emacs Lisp files that comprise ILISP. Start with the '`Makefile`' file, in the section after the comment `Various variables` (you can safely ignore the variables for configuring packaging and distribution, which are meant for maintainers).

First, set the `EMACS` variable to be the pathname of the Emacs you will be using ILISP with. This is the Emacs that will be used to compile ILISP with.

If your Emacs supports the `easymenu` package, it is possible to make ILISP add to Lisp mode buffers and buffers with inferior Lisp processes, or to Scheme mode buffers and buffers with inferior Scheme pProcesses, an `Ilisp` menu with all available commands. To enable this feature, set to `t` the variable `enable-ilisp-cl-easy-menu-p` in '`ilisp-def.el`' for the Common Lisp dialects, and `enable-ilisp-scheme-easy-menu-p` for Scheme dialects. Setting these variables also causes the default `Lisp` menu to be removed before displaying the `Ilisp` one.

See the file '`INSTALLATION`' for additional configuration options for specific Lisp dialects. '`README.CL-HS`' includes a list of known problems with CLISP by Haible and Stoll. Please note that the configuration and installation information provided in the file may be out of date.

Run `make` or `make compile` to build ILISP from source. Ignore any compilation warnings unless they result in ILISP not compiling completely.

For reducing the Emacs startup time you may run `make loadfile`. This concatenates all '`.elc`' (the compiled Emacs Lisp files) into an '`ilisp-all.elc`' file and removes the '`*.elc`' files. So your Emacs can load one single compiled file faster than a bunch of smaller compiled files.

You should then copy relevant sections of '`ilisp.emacs`' to your '`.emacs`' or to the system-wide '`default.el`' file, depending on who will be using ILISP.

You should add the directory where all of the ILISP Emacs Lisp files reside to your `load-path`. There is an example of this in '`ilisp.emacs`'.

As an alternative you could set up a '`.ilisp`' which contains the appropriate portions of '`ilisp.emacs`', in order to avoid cluttering too much '`.emacs`' or '`default.el`'.

The first time a dialect is started, the interface files will complain about not being compiled, just ignore the message. Once a Lisp dialect is started up, you should execute the command `ilisp-compile-inits` which will compile the '`*.lisp`' files and write them to the same directory as the ILISP files.

The binary files should have a unique extension for each different combination of architecture and Lisp dialect. You will need to change `ilisp-init-binary-extension` and `ilisp-init-binary-command` to get additional extensions. The binary for each different architecture should be different.

If you want to build the interface files into a Lisp world, you will also need to set `ilisp-load-inits` to `nil` in the same place that you change `ilisp-program` to load the Lisp world.

There is an `ilisp-site-hook` for initializing site specific stuff like program locations when ILISP is first loaded. You may want to define appropriate autoloads in your system Emacs start up file. See Section 1.3 [Autoloading], page 9.

Example site init:

```
;;; CMU site
(setq ilisp-site-hook
      '(lambda ()
         (setq ilisp-motd "CMU ILISP V%s")
         (setq expand-symlinks-rfs-exists t)
         (setq allegro-program "/usr/local/acl5/lisp")
         (setq lucid-program "/usr/misc/.lucid/bin/lisp")))
```

Kent Pitman and The Harlequin Group Ltd. have made publicly available on the Web the Common Lisp HyperSpec, an HTML version of the full text of the ANSI Common Lisp specification. Daniel Barlow, Stephen Carney and Erik Naggum independently developed Emacs Lisp packages for looking up Lisp symbols in the HyperSpec and displaying the relevant sections with a Web browser. ILISP includes all of them in the '`extra`' directory of the distribution tree. By default ILISP uses Naggum's package. If you want to use one of the others, see the comments at the beginning of the corresponding files.

The '`ilisp.emacs`' file provides sample instructions for making Naggum's package access a local copy of the HyperSpec. See Section 1.3 [Autoloading], page 9. Since the package relies on the `browse-url` Emacs package, make sure that the latter is properly configured.

Previous versions of ILISP provided commands for accessing the online Common Lisp documentation shipped with Franz Inc.'s Allegro CL product (`fi:clman` module). The public availability of the HyperSpec, and the inclusion since version 5.9 of ILISP of the `hyperspec` packages, make access to the Franz documentation no longer necessary. So by default ILISP does not load the `fi:clman` module, but if you still want to use its commands set the `ilisp-use-fi-clman-interface-p` to `t` in '`ilisp-def.el`'.

The ILISP documentation consists of a user manual and a reference card. Both of them are in the '`docs`' directory of the distribution tree.

The generation of GNU Info, DVI, PostScript and HTML versions of the manual from the Texinfo source is controlled by the '`Makefile`' in the '`docs`' directory. Run `make docs` or just `make` to generate all of the formats. If you are interested in only some of them then issue the appropriate command: `make info` for GNU Info, `make dvi` for DVI, `make ps` for PostScript and `make html` for HTML. To remove the intermediate files produced during the generation of DVI output you can run `make tmpclean`.

The ILISP reference card is available in DVI and PostScript formats together with the TeX source. Check the comments at the beginning of the TeX source if you need to generate a version with a different number of columns.

1.2 Files of ILISP

The files you need to use ilisp are:

'bridge.el'
: Process to process communication.

'comint-ipc.el'
: Extensions for sending commands and getting results.

'comint-v18.el'
: The basic comint abstraction. You only need this if running Emacs-18.

'docs/'
: Directory containing the documentation.

'extra/'
: Directory containing contributed packages.

'ilcompat.el'
: Compatibility code between FSF-18, FSF-19, FSF-20, LEmacs-19 and XEmacs.

'ild.el'
: Uniform interface to Lisp debuggers.

'ilfsf18.el'
: Support for GNU Emacs v18.

'ilfsf19.el'
: Support for GNU Emacs v19.

'ilfsf20.el'
: Support for GNU Emacs v20.

'ilisp-acl.el'
: Allegro Common Lisp dialect definition.

'ilisp-aut.el'
: Autoload definitions.

'ilisp-bat.el'
: ILISP batch code module.

'ilisp-bug.el'
: ILISP bug submittal code.

'ilisp-chs.el'
: Haible and Stoll's CLISP Common Lisp dialect definition.

'ilisp-cl.el'
: Common-Lisp dialect definition.

'ilisp-cl-easy-menu.el'
: ILISP menu definition (Common Lisp dialects) for the Easymenu package.

'ilisp-cmt.el'
: Comint related code/setup.

'ilisp-cmu.el'
: CMU CL Common Lisp dialect definition.

'ilisp-cpat.el'
: Compatability between GNU Emacs 18, 19, and Lucid Emacs 19.

'ilisp-def.el'
: Variable definitions.

'`ilisp-dia.el`'
: Code for customizing dialects

'`ilisp-doc.el`'
: ILISP mode documenation.

'`ilisp-el.el`'
: Emacs Lisp additions.

'`ilisp-ext.el`'
: Standalone lisp-mode extensions.

'`ilisp-hi.el`'
: High level interface code.

'`ilisp-hlw.el`'
: Harlequin LispWorks Common Lisp dialect definition.

'`ilisp-hnd.el`'
: Error handling.

'`ilisp-ind.el`'
: Indentation code.

'`ilisp-inp.el`'
: Buffer input module.

'`ilisp-kcl.el`'
: Kyoto Common Lisp dialect definition.

'`ilisp-key.el`'
: Keymap setups, including `ilisp-lispm-bindings`.

'`ilisp-kil.el`'
: Interface to reset/kill/abort inferior Lisp.

'`ilisp-low.el`'
: Low level interface code.

'`ilisp-luc.el`'
: Lucid/Liquid Common Lisp dialect definition.

'`ilisp-mak.el`'
: Code for supporting the ILISP build process.

'`ilisp-menu.el`'
: Menu definitions.

'`ilisp-mnb.el`'
: Menubar definition code.

'`ilisp-mod.el`'
: ILISP mode definition.

'`ilisp-mov.el`'
: Buffer-point movement code.

'ilisp-out.el'
: Output handling, include typeout window (a popper replacement).

'ilisp-prc.el'
: Process handling code.

'ilisp-prn.el'
: Parenthesis handling.

'ilisp-rng.el'
: Match ring code.

'ilisp-s2c.el'
: Scheme->C Scheme dialect definition.

'ilisp-sch.el'
: Scheme dialects definitions.

'ilisp-scheme-easy-menu.el'
: ILISP menu definition (Scheme dialects) for the Easymenu package.

'ilisp-snd.el'
: ilisp-send definitions and associated code.

'ilisp-src.el'
: ILISP source code module.

'ilisp-sym.el'
: ILISP symbol handling.

'ilisp-utl.el'
: Misc. utilities.

'ilisp-val.el'
: Buffer value interface.

'ilisp-xfr.el'
: Transfer between Lisp <-> Emacs code.

'ilisp-xls.el'
: XLisp and XLisp-Stat dialects definitions.

'ilisp.el'
: File to be loaded, loads in all necessary parts of ILISP.

'ilisp.emacs'
: File with sample '.emacs' code for ILISP.

'ilisp.texi'
: Texinfo file for ILISP.

'illuc19.el'
: Support for Lucid Emacs v19.

'ilxemacs.el'
: Support for XEmacs.

'*.lisp'
: ILISP support code. Each dialect will have one of these files.

1.3 How to define autoload entries

A complete example of things you may want to add to your `.emacs` can be found in the in the file '`ilisp.emacs`' in the `ilisp-directory` what follows is that file.

```
;;; This file shows examples of some of the things you might want to
;;; do to install or customize ILISP.  You may not want to include all
;;; of them in your .emacs.  For example, the default key binding
;;; prefix for ILISP is C-z and this file changes the default prefix
;;; to C-c. For more information on things that can be changed, see
;;; the file ilisp.el.


;;; If ilisp lives in some non-standard directory, you must tell
;;; Emacs where to get it. This may or may not be necessary.

(setq load-path (cons (expand-file-name "~jones/emacs/ilisp/")
                      load-path))


;;; If you always want partial minibuffer completion

(require 'completer)

;;; If want TMC completion then you will have to Ftp it yourself
;;; from think.com It's become to flaky for me to deal with. -- Ivan

; (load "completion")
; (initialize-completions)


;;; If you want to redefine typeout-window keys.

; (add-hook 'ilisp-load-hook
;   '(lambda ()
;      (define-key global-map "\C-c1" 'ilisp-bury-output)
;      (define-key global-map "\C-cv" 'ilisp-scroll-output)
;      (define-key global-map "\C-cg" 'ilisp-grow-output)))


;;; Autoload based on your Lisp. You only really need the one you
;;; use. If called with a prefix, you will be prompted for a
;;; buffer and program.

(autoload 'run-ilisp   "ilisp" "Select a new inferior Lisp." t)

(autoload 'common-lisp "ilisp" "Inferior generic Common Lisp." t)

;; Franz
(autoload 'allegro     "ilisp" "Inferior Allegro Common Lisp." t)
```

```
;; Lucid
; (autoload 'lucid     "ilisp" "Inferior Lucid Common Lisp." t)

;; Harlequin
; (autoload lispworks  "ilisp"
;   "Inferior Harlequin Common Lisp (LispWorks)." t)
; (autoload harlequin  "ilisp"
;   "Inferior Harlequin Common Lisp (LispWorks)." t)
; (autoload pulcinella "ilisp"
;   "Inferior Harlequin Common Lisp (LispWorks)." t)
;; Italian "Commedia dell'Arte" twist.

;; CMUCL
(autoload 'cmulisp     "ilisp" "Inferior CMU Common Lisp." t)

;; CLISP (Bruno Haible and Michael Stoll)
(autoload 'clisp-hs   "ilisp"
  "Inferior Haible/Stoll CLISP Common Lisp." t)

;; KCL dialects
; (autoload 'kcl       "ilisp"
;   "Inferior Kyoto Common Lisp." t)
; (autoload 'akcl      "ilisp"
;   "Inferior Austin Kyoto Common Lisp." t)
; (autoload 'ibcl "ilisp"
;   "Ibuki Common Lisp." t)
; (autoload 'gcl "ilisp"
;   "Inferior GNU Common Lisp." t)
; (autoload 'ecl "ilisp"
;   "Inferior EcoLisp." t)

;; XLisp
; (autoload 'xlisp      "ilisp" "Inferior XLisp." t)
; (autoload 'xlispstat  "ilisp" "Inferior XLisp-Stat." t)

;; Scheme
; (autoload 'scheme     "ilisp" "Inferior generic Scheme." t)
; (autoload 'oaklisp    "ilisp" "Inferior Oaklisp Scheme." t)
; (autoload 'scm        "ilisp" "Inferior SCM Scheme." t)
; (autoload 'chez       "ilisp" "Inferior Chez Scheme." t)
; (autoload 'stk        "ilisp" "Inferior STk Scheme." t)
; (autoload 'snow       "ilisp" "Inferior STk Scheme without Tk." t)
; (autoload 'guile      "ilisp" "Inferior GUILE Scheme." t)


;;; Define where Lisp programs are found. (This may already have
;;; been done at your site).
```

```
(setq allegro-program "/usr/local/acl5/lisp")

; (setq lucid-program "/usr/misc/.lucid/bin/lisp")

(setq clisp-hs-program "clisp -I")

; (setq lispworks-program
;       "/somewhere/in/the/directory/tree/lispworks")

(setq cmulisp-program
      "/usr/local/lib/cmucl/bin/lisp")

;; If you are interested in maintaining CMUCL or compiling it
;; from source then set this to where the source files are.
; (setq cmulisp-local-source-directory
;       "/usr/robotics/shared/cmu-cl/17e/")

; (setq akcl-program "kcl")
; (setq gcl-program "gcl")
; (setq ecl-program "ecl")

; (setq xlisp-program "xlisp")
; (setq xlispstat-program "xlispstat")

; (setq scm-program "scm -i")
; (setq chez-program "petite")
; (setq stk-program "stk -interactive")
; (setq snow-program "snow -interactive")
; (setq guile-program "guile")


;;; This makes reading a Lisp or Scheme file load in ILISP.

(set-default 'auto-mode-alist
     (append '(("\\.lisp$" . lisp-mode)
                 ("\\.lsp$" . lisp-mode)
                 ("\\.cl$" . lisp-mode))
               auto-mode-alist))

(setq lisp-mode-hook '(lambda () (require 'ilisp)))

(set-default 'auto-mode-alist
             (append '(("\\.scm$" . scheme-mode)
                         ("\\.ss$" . scheme-mode)
                         ("\\.stk$" . scheme-mode)
                         ("\\.stklos$" . scheme-mode))
                       auto-mode-alist))
```

```
(setq scheme-mode-hook '(lambda () (require 'ilisp)))


;;; Configuration of Erik Naggum's HyperSpec access package.

;; If you have a local copy of the HyperSpec, set its path here.
; (setq common-lisp-hyperspec-root
;       "file:/home/joe/HyperSpec/")


;;; Sample load hook

(add-hook 'ilisp-load-hook
          '(lambda ()
             ;; Change default key prefix to C-c
             (setq ilisp-prefix "\C-c")

             ;; Set a keybinding for the COMMON-LISP-HYPERSPEC command
             (defkey-ilisp "" 'common-lisp-hyperspec)

             ;; Make sure that you don't keep popping up the 'inferior
             ;; Lisp' buffer window when this is already visible in
             ;; another frame. Actually this variable has more impact
             ;; than that. Watch out.
             ; (setq pop-up-frames t)

             (message "Running ilisp-load-hook")
             ;; Define LispMachine-like key bindings, too.
             ; (ilisp-lispm-bindings) Sample initialization hook.

             ;; Set the inferior Lisp directory to the directory of
             ;; the buffer that spawned it on the first prompt.
             (add-hook 'ilisp-init-hook
                       '(lambda ()
                          (default-directory-lisp ilisp-last-buffer)))
             ))
```

2 How to run a Lisp process using ILISP

To start a Lisp use *M-x run-ilisp*, or a specific dialect like *M-x allegro*. If called with a prefix you will be prompted for a buffer name and a program to run. The default buffer name is the name of the dialect. The default program for a dialect will be the value of DIALECT-program or the value of ilisp-program inherited from a less specific dialect. If there are multiple Lisp's, use the dialect name or *M-x select-ilisp* (*C-Z S*) to select the current ILISP buffer.

Entry into ILISP mode runs the hooks on `comint-mode-hook` and `ilisp-mode-hook` and then DIALECT-hooks specific to Lisp dialects in the nesting order above. Many dialects call `ilisp-load-init` in their dialect setup.

These are the currently supported dialects.

- `allegro` Allegro Common Lisp from Franz Inc.
- `akcl` Austin Kyoto Common Lisp, the U. Texas derivative.
- `chez` Chez Scheme by Cadence Research Systems.
- `clisp-hs` CLISP by Haible and Stoll.
- `cmucl` CMU Common Lisp, the major development platform for ILISP so far.
- `common-lisp` Generic Common Lisp.
- `ecl` EcoLisp, the Embeddable Common Lisp by Beppe Attardi. A derivative of KCL and AKCL.
- `gcl` GNU Common Lisp, the official GNU release. A derivative of AKCL.
- `guile` GUILE Scheme by the GNU Project.
- `ibcl` Ibuki Common Lisp, derived from KCL.
- `kcl` Kyoto Common Lisp, original version.
- `liquid` Liquid Common Lisp, the successor of Lucid Common Lisp supported by Harlequin Ltd.
- `lispworks` LispWorks Common Lisp from Harlequin Ltd.
- `lucid` Lucid Common Lisp, currently supported by Harlequin Ltd.
- `oaklisp` Oaklisp scheme.
- `scheme` Generic Scheme.
- `scm` SCM Scheme by Aubrey Jeffer.
- `snow` Snow, STk Scheme without supoort for the Tk toolkit.
- `stk` STk scheme by Erick Gallesio.
- `xlisp` XLisp by David Betz.
- `xlispstat` XLisp-Stat, a derivative of XLisp for statistical computations.

Support for Scheme and XLisp dialects is experimental and your feedback is welcome. The '`ilisp-s2c.el`' file contains a first cut at defining the Scheme->C dialect, but it is neither compiled nor loaded by ILISP. See '`scheme2c.mail`' for a trick for making Scheme->C work with ILISP.

To define a new dialect, See Section 6.1 [Defining new dialects], page 29 and See Chapter 5 [Customization], page 27. If anyone figures out support for other dialects, I would be happy to include it in future releases. See Chapter 6 [Dialects], page 29.

The currently supported dialects are listed below so that the indentation correponds to the hierarchical relationship between dialects:

```
common-lisp
  allegro
  clisp-hs
  cmulisp
  kcl
    akcl
      gcl
      ecl
    ibcl
  lispworks
  lucid
    liquid
scheme
  chez
  guile (yep! Here it comes. Still incomplete though)
  oaklisp
  Scheme->C (still "in fieri")
  scm
  snow
  stk
xlisp
  xlispstat
```

3 Buffers used by ILISP, and their commands

dialect
: The Lisp listener buffer. Forms can be entered in this buffer in, and they will be sent to Lisp when you hit return if the form is complete. This buffer is in ilisp-mode, which is built on top of comint-mode, and all comint commands such as history mechanism and job control are available.

lisp-mode-buffers
: A buffer is assumed to contain Lisp source code if its major mode is in the list `lisp-source-modes`. If it's loaded into a buffer that is in one of these major modes, it's considered a Lisp source file by `find-file-lisp`, `load-file-lisp` and `compile-file-lisp`. Used by these commands to determine defaults.

`*Completions*`
: Used for listing completions of symbols or files by the completion commands. See Section 4.13 [Completion], page 25.

`*Aborted Commands*`
: See Section 4.10 [Interrupts], page 22.

`*Errors*`
`*Output*`
`*Error Output*`
: used to pop-up results and errors from the inferior Lisp.

`*ilisp-send*`
: Buffer containing the last form sent to the inferior Lisp.

`*Edit-Definitions*`
`*All-Callers*`
: See Section 4.6 [Source code commands], page 20.

`*Last-Changes*`
`*Changed-Definitions*`
: See Section 4.7 [Batch commands], page 21.

3.1 Typeout windows

All ILISP output is funneled through the function which is bound to the hook `ilisp-display-output-function`. The function gets a single argument, a string, and should make that output visible to the user somehow.

One possible choice for output display is `ilisp-display-output-in-typeout-window`, which pops up a window at the top of the current screen which is just large enough to display the output. This window can be "remotely controlled" by the commands `ilisp-scroll-output`, `ilisp-bury-output`, and `ilisp-grow-output`.

Unlike the old popper facility, the ilisp typeout window facility does not trounce on any existing Emacs functions or on any common key bindings, like *C-x o*.

Other built-in functions which might be useful as values for `ilisp-display-output-function` include `ilisp-display-output-default`, `ilisp-display-output-adaptively`, `ilisp-display-output-in-lisp-listener`, `ilisp-display-output-in-temp-buffer`, and `ilisp-display-output-in-typeout-window`.

The default display function is `ilisp-display-output-default`, which obeys the `lisp-no-popper` variable.

Users are encouraged to write their own output display functions to get the exact desired behavior, displaying on a private Emacs screen, in a pop-up dialog box, or whetever.

C-Z 1 (ilisp-bury-output)
: deletes and buries the typeout output window.

C-Z v (ilisp-scroll-output)
: scrolls the output window if it is showing, otherwise does nothing. If it is called with a negative prefix, it will scroll backwards.

C-Z G (ilisp-grow-output)
: will grow the output window if showing by the prefix number of lines.

An alternative to typeout windows is to always have the inferior Lisp buffer visible and have all output go there. Setting `lisp-no-popper` to `t` will cause all output to go to the inferior Lisp buffer. Setting `lisp-no-popper` to `'message` will make output of one line go to the message window. Setting `comint-always-scroll` to `t` will cause process output to always be visible. If a command gets an error, you will be left in the break loop.

3.2 Switching buffers

Commands to make switching between buffers easier.

C-Z b (switch-to-lisp)
: will pop to the current ILISP buffer or if already in an ILISP buffer, it will return to the buffer that last switched to an ILISP buffer. With a prefix, it will also go to the end of the buffer. If you do not want it to pop, set `pop-up-windows` to nil.

M-C-l (previous-buffer-lisp)
: will switch to the last visited buffer in the current window or the Nth previous buffer with a prefix.

4 ILISP Commands

Most of these key bindings work in both Lisp Mode and ILISP mode. There are a few additional and-go bindings found in Lisp Mode.

4.1 Eval and compile functions

In Lisp, the major unit of interest is a form, which is anything between two matching parentheses. Some of the commands here also refer to "defun," which is a list that starts at the left margin in a Lisp buffer, or after a prompt in the ILISP buffer. These commands refer to the "defun" that contains the point.

"A call" refers to a reference to a function call for a function or macro, or a reference to a variable. Commands which "insert a call" in the ILISP buffer will bring up the last command which matches it or else will insert a template for a call.

When an eval is done of a single form matching `ilisp-defvar-regexp` the corresponding symbol will be unbound and the value assigned again.

When you send a form to Lisp, the status light will reflect the progress of the command. In a Lisp mode buffer the light will reflect the status of the currently selected inferior Lisp unless `lisp-show-status` is nil. The very first inferior Lisp command executed may send some forms to initialize the inferior Lisp. If you want to find out what command is currently running, use the command *C-Z s* (status-lisp). If you call it with a prefix, the pending commands will be displayed as well.

Note that in this table as elsewhere, the key *C-Z* (ilisp-prefix) is used as a prefix character for ILISP commands, though this may be changed. For a full list of key-bindings, use *M-x describe-mode* or *M-x describe-bindings* while in an ILISP-mode buffer.

The eval/compile commands verify that their expressions are balanced and then send the form to the inferior Lisp. If called with a positive prefix, the result of the operation will be inserted into the buffer after the form that was just sent.

For commands which operate on a region, the result of the compile or eval is the last form in the region.

The '`and-go`' versions will perform the operation and then immediately switch to the ILISP buffer where you will see the results of executing your form. If `eval-defun-and-go-lisp` or `compile-defun-and-go-lisp` is called with a prefix, a call for the form will be inserted as well.

C-Z
: The prefix-key for most ILISP commands. This can be changed by setting the variable `ilisp-prefix`.

RET (return-ilisp)
: In ILISP-mode buffer, sends the current form to lisp if complete, otherwise creates a new line and indents. If you edit old input, the input will be copied to the end of the buffer first and then sent.

C-] (close-and-send-lisp)
: Closes the current sexp, indents it, and then sends it to the current inferior Lisp.

LFD (newline-and-indent-lisp)
: Insert a new line and then indent to the appropriate level. If called at the end of the inferior Lisp buffer and an sexp, the sexp will be sent to the inferior Lisp without a trailing newline.

C-Z e *(eval-defun-lisp)*
M-C-x (eval-defun-lisp)
C-Z C-e (eval-defun-and-go-lisp)
Send the defun to Lisp.

C-Z r *(eval-region-lisp)*
C-Z C-r (eval-region-and-go-lisp)

C-Z n *(eval-next-sexp-lisp)*
C-Z C-n (eval-next-sexp-and-go-lisp)

C-Z c *(compile-defun-lisp)*

C-Z C-c (compile-defun-lisp-and-go)
When `compile-defun-lisp` is called in an inferior Lisp buffer with no current form, the last form typed to the top-level will be compiled.

C-Z w *(compile-region-lisp)*
C-Z C-w (compile-region-and-go-lisp)

If any of the forms contain an interactive command, then the command will never return. To get out of this state, you need to use `abort-commands-lisp` (*C-Z g*). If `lisp-wait-p` is t, then EMACS will display the result of the command in the minibuffer or a pop-up window. If `lisp-wait-p` is `nil`, (the default) the send is done asynchronously and the results will be brought up only if there is more than one line or there is an error. In this case, you will be given the option of ignoring the error, keeping it in another buffer or keeping it and aborting all pending sends. If there is not a command already running in the inferior Lisp, you can preserve the break loop. If called with a negative prefix, the sense of `lisp-wait-p` will be inverted for the next command.

4.2 Documentation functions

`describe-lisp`, `inspect-lisp`, `arglist-lisp`, and `documentation-lisp` switch whether they prompt for a response or use a default when called with a negative prefix. If they are prompting, there is completion through the inferior Lisp by using *TAB* or *M-TAB*. When entering an expression in the minibuffer, all of the normal ILISP commands like `arglist-lisp` also work.

Commands that work on a function will use the nearest previous function symbol. This is either a symbol after a '`#'`' or the symbol at the start of the current list.

The `fi:clman` and `fi:clman-apropos` commands for accessing the Franz Allegro CL documentation are not enabled by default. See Section 1.1 [Configuration and compilation], page 4.

C-Z `a (arglist-lisp)`
Return the arglist of the current function. With a numeric prefix, the leading paren will be removed and the arglist will be inserted into the buffer.

C-Z `d (documentation-lisp)`
Infers whether function or variable documentation is desired. With a negative prefix, you can specify the type of documentation as well. With a positive prefix the documentation of the current function call is inserted into the buffer.

C-Z `i (describe-lisp)`
Describe the previous sexp (it is evaluated). If there is no previous sexp and if called from inside an ILISP buffer, the previous result will be described.

C-Z `i` `(describe-lisp)`
: Describe the previous sexp (it is evaluated). If there is no previous sexp and if called from inside an ILISP buffer, the previous result will be described.

C-Z `I` `(inspect-lisp)`
: Switch to the current inferor Lisp and inspect the previous sexp (it is evaluated). If there is no previous sexp and if called from inside an ILISP buffer, the previous result will be inspected.

C-Z `H` `(hyperspec-lookup)`
: Look up a standard symbol in the Common Lisp HyperSpec and display the relevant section.

C-Z `D` `(fi:clman)`
C-Z `A` `(fi:clman-apropos)`
: If the Franz online Common Lisp manual is available, get information on a specific symbol. `fi:clman-apropos` will get information apropos a specific string. Some of the documentation is specific to the Allegro dialect, but most of it is for standard Common Lisp.

4.3 Macroexpansion

C-Z M (macroexpand-lisp)
C-Z m (macroexpand-1-lisp)
: These commands apply to the next sexp. If called with a positive numeric prefix, the result of the macroexpansion will be inserted into the buffer. With a negative prefix, prompts for expression to expand.

4.4 Tracing functions

C-Z t (trace-defun-lisp)
: traces the current defun. When called with a numeric prefix the function will be untraced. When called with negative prefix, prompts for function to be traced.

C-Z C-t (trace-defun-lisp-break)
: traces the current defun and enters the debugger whenever that function is invoked. When called with a numeric prefix the function will be untraced. When called with negative prefix, prompts for function to be traced.

4.5 Package Commands

The first time an inferior Lisp mode command is executed in a Lisp Mode buffer, the package will be determined by using the regular expression `ilisp-package-regexp` to find a package sexp and then passing that sexp to the inferior Lisp through `ilisp-package-command`. For the '`common-lisp`' dialect, this will find the first `(in-package PACKAGE)` form in the file. A buffer's package will be displayed in the mode line. If a buffer has no specification, forms will be evaluated in the current inferior Lisp package.

Buffer package caching can be turned off by setting the variable `lisp-dont-cache-package` to `T`. This will force ILISP to search for the closest previous `ilisp-package-regexp` in the buffer each time an inferior Lisp mode command is executed.

C-Z `p` `(package-lisp)`
: Show the current package of the inferior Lisp.

C-Z `P` `(set-package-lisp)`
: Set the inferior Lisp package to the current buffer's package or with a prefix to a manually entered package.

`M-x set-buffer-package-lisp`
: Set the buffer's package from the buffer. If it is called with a prefix, the package can be set manually.

4.6 Source Code Commands

The following commands all deal with finding things in source code. The first time that one of these commands is used, there may be some delay while the source module is loaded. When searching files, the first applicable rule is used:

- try the inferior Lisp,
- try a tags file if defined,
- try all buffers in one of `lisp-source-modes` or all files defined using `lisp-directory`.

`M-x lisp-directory` defines a set of files to be searched by the source code commands. It prompts for a directory and sets the source files to be those in the directory that match entries in `auto-mode-alist` for modes in `lisp-source-modes`. With a positive prefix, the files are appended. With a negative prefix, all current buffers that are in one of `lisp-source-modes` will be searched. This is also what happens by default. Using this command stops using a tags file.

`edit-definitions-lisp`, `who-calls-lisp`, and `edit-callers-lisp` will switch whether they prompt for a response or use a default when called with a negative prefix. If they are prompting, there is completion through the inferior Lisp by using *TAB* or *M-TAB*. When entering an expression in the minibuffer, all of the normal ILISP commands like *arglist-lisp* also work.

`edit-definitions-lisp` (*M-.*) will find a particular type of definition for a symbol. It tries to use the rules described above. The files to be searched are listed in the buffer `*Edit-Definitions*`. If `lisp-edit-files` is nil, no search will be done if not found through the inferior Lisp. The variable `ilisp-locator` contains a function that when given the name and type should be able to find the appropriate definition in the file. There is often a flag to cause your Lisp to record source files that you will need to set in the initialization file for your Lisp. The variable is `*record-source-files*` in both allegro and lucid. Once a definition has been found, `next-definition-lisp` (*M-,*) will find the next definition (or the previous definition with a prefix).

`edit-callers-lisp` (*C-Z ^*) will generate a list of all of the callers of a function in the current inferior Lisp and edit the first caller using `edit-definitions-lisp`. Each successive call to `next-caller-lisp` (*M-`*) will edit the next caller (or the previous caller with a prefix). The list is stored in the buffer `*All-Callers*`. You can also look at the callers by doing *M-x who-calls-lisp*.

`search-lisp` (*M-?*) will search the current tags files, `lisp-directory` files or buffers in one of `lisp-source-modes` for a string or a regular expression when called with a prefix. `next-definition-lisp` (*M-,*) will find the next definition (or the previous definition with a prefix).

`replace-lisp` (*M-"*) will replace a string (or a regexp with a prefix) in the current tags files, `lisp-directory` files or buffers in one of `lisp-source-modes`.

Here is a summary of the above commands (behavior when given prefix argument is given in parentheses):

`M-x lisp-directory`
: Define a set of files to be used by the source code commands.

`M-. (edit-definitions-lisp)`
: Find definition of a symbol.

`M-, (next-definition-lisp)`
: Find next (previous) definition.

C-Z `^ (edit-callers-lisp)`
: Find all callers of a function, and edit the first.

`` M-` (next-caller-lisp) ``
: Edit next (previous) caller of function set by `edit-callers-lisp`.

`M-x who-calls-lisp`
: List all the callers of a function.

`M-? (search-lisp)`
: Search for string (regular expression) in current tags, `lisp-directory` files or buffers. Use `next-definition-lisp` to find next occurence.

`M-" (replace-lisp)`
: Replace a string (regular expression) in files.

4.7 Batch commands

The following commands all deal with making a number of changes all at once. The first time one of these commands is used, there may be some delay as the module is loaded. The eval/compile versions of these commands are always executed asynchronously.

`mark-change-lisp` (*C-Z SPC*) marks the current defun as being changed. A prefix causes it to be unmarked. `clear-changes-lisp` (*C-Z * 0*) will clear all of the changes. `list-changes-lisp` (*C-Z * l*) will show the forms currently marked.

`eval-changes-lisp` (*C-Z * e*), or `compile-changes-lisp` (*C-Z * c*) will evaluate or compile these changes as appropriate. If called with a positive prefix, the changes will be kept. If there is an error, the process will stop and show the error and all remaining changes will remain in the list. All of the results will be kept in the buffer `*Last-Changes*`.
Summary:

C-Z `SPC (mark-change-lisp)`
: Mark (unmark) current defun as changed.

C-Z `* e (eval-changes-lisp)`
C-Z `* c (compile-changes-lisp)`
: Call with a positive prefix to keep changes.

C-Z `* 0 (clear-changes-lisp)`

C-Z `* l (list-changes-lisp)`

4.8 Files and directories

File commands in Lisp source-mode buffers keep track of the last used directory and file. If the point is on a string, that will be the default if the file exists. If the buffer is one of `lisp-source-modes`, the buffer file will be the default. Otherwise, the last file used in a lisp-source-mode will be used.

C-x C-f (find-file-lisp)
: will find a file. If it is in a string, that will be used as the default if it matches an existing file. Symbolic links are expanded so that different references to the same file will end up with the same buffer.

C-Z l (load-file-lisp)
: will load a file into the inferior Lisp. You will be given the opportunity to save the buffer if it has changed and to compile the file if the compiled version is older than the current version.

C-Z k (compile-file-lisp)
: will compile a file in the current inferior Lisp.

C-Z ! (default-directory-lisp)
: sets the default inferior Lisp directory to the directory of the current buffer. If called in an inferior Lisp buffer, it sets the Emacs `default-directory` to the Lisp default directory.

4.9 Switching between interactive and raw keyboard modes

There are two keyboard modes for interacting with the inferior Lisp, “interactive” and “raw”. Normally you are in interactive mode where keys are interpreted as commands to EMACS and nothing is sent to the inferior Lisp unless a specific command does so. In raw mode, all characters are passed directly to the inferior Lisp without any interpretation as EMACS commands. Keys will not be echoed unless ilisp-raw-echo is T.

Raw mode can be turned on interactively by the command `raw-keys-ilisp` (*C-Z #*) and will continue until you type *C-G*. Raw mode can also be turned on/off by inferior Lisp functions if the command `io-bridge-ilisp` (M-x io-bridge-ilisp) has been executed in the inferior Lisp either interactively or on a hook. To turn on raw mode, a function should print ^[1^] and to turn it off should print ^[0^]. An example in Common Lisp would be:

```
(progn (format t "œ1Æ") (print (read-char)) (format t "œ0Æ"))
```

4.10 Interrupts, aborts, and errors

If you want to abort the last command you can use *C-g*.

If you want to abort all commands, you should use the command `abort-commands-lisp` (*C-Z g*). Commands that are aborted will be put in the buffer `*Aborted Commands*` so that you can see what was aborted. If you want to abort the currently running top-level command, use `interrupt-subjob-ilisp` (*C-c C-c*). As a last resort, *M-x panic-lisp* will reset the ILISP state without affecting the inferior Lisp so that you can see what is happening.

`delete-char-or-pop-ilisp` (*C-d*) will delete prefix characters unless you are at the end of an ILISP buffer in which case it will pop one level in the break loop.

`reset-ilisp`, (*C-Z z*) will reset the current inferior Lisp's top-level so that it will no longer be in a break loop.

Summary:

C-c C-c (interrupt-subjob-ilisp)
: Send a keyboard interrupt signal to lisp.

C-Z g (abort-commands-lisp)
: Abort all running or unsent commands.

M-x panic-lisp (panic-lisp)
: Reset the ILISP process state.

C-Z z (reset-ilisp)
: Reset Lisp to top-level.

C-d (delete-char-or-pop-ilisp)
: If at end of buffer, pop a level in break loop.

If `lisp-wait-p` is `nil` (the default), all sends are done asynchronously and the results will be brought up only if there is more than one line or there is an error. In case, you will be given the option of ignoring the error, keeping it in another buffer or keeping it and aborting all pending sends. If there is not a command already running in the inferior Lisp, you can preserve the break loop. If called with a negative prefix, the sense of `lisp-wait-p` will be inverted for the next command.

4.11 Interface to Lisp debuggers

ILD is an interface to Lisp debuggers, currently the ones of the AKCL, Allegro, CLISP, CMU CL and Lucid Common Lisp dialects (see '`ild.mail`'). It uses a standard set of single-keystroke commands to interface to a variety of different debuggers and is vaguely modelled after the Symbolics debugger. It provides two key advantages: single keystrokes for moving up and down the stack, and a uniform interface to different debuggers.

Not all debugger commands are available in all implementations. Some are, but further work is needed. These are noted in the code (see the dialect definition files). If you know how to fix them please contact the ILISP maintainer.

Here is a list of the available ILD commands:

M-a (ild-abort)
: Abort.

M-c (ild-continue)
: Continue.

M-C-n (ild-next)
: Next stack frame (with numeric argument n, move to the next n-th stack frame).

M-C-p (ild-previous)
: Previous stack frame (with numeric argument n, move to the previous n-th stack frame).

C-c < (ild-top)
: Top stack frame.

C-c > (ild-bottom)
: Bottom stack frame.

M-b (ild-backtrace)
: Backtrace.

M-C-d (ild-locals)
: Display all local variables.

M-C-l (ild-local)
: Display a particular local variable (with numeric argument *n*, display the *n*-th local variable).

C-c r (ild-return)
: Return.

M-C-r (ild-retry)
: Retry.

C-x t (ild-trap-on-exit)
: Trap on exit.

C-c L (select-lisp)
: Select Lisp interaction buffer.

C-z C-s (slow-lisp)
: Set compiler options for maximal debuggability.

C-z C-f (fast-lisp)
: Set compiler options for fastest but least debuggable code.

4.12 Command history

ILISP mode is built on top of `comint-mode`, the general command-interpreter-in-a-buffer mode. As such, it inherits many commands and features from this, including a command history mechanism.

Each ILISP buffer has a command history associated with it. Commands that do not match `ilisp-filter-regexp` and that are longer than `ilisp-filter-length` and that do not match the immediately prior command will be added to this history.

M-n (comint-next-input)
M-p (comint-previous-input)
: Cycle through the input history.

M-s (comint-previous-similar-input)
: Cycle through input that has the string typed so far as a prefix.

M-N (comint-psearch-input)
: Search forwards for prompt.

M-P (comint-msearch-input)
Search backwards for prompt.

C-c R (comint-msearch-input-matching)
Search backwards for occurrence of prompt followed by string which is prompted for (*not* a regular expression).

See `comint-mode` documentation for more information on '`comint`' commands.

4.13 Completion

Commands to reduce number of keystrokes.

M-TAB (complete-lisp)
will try to complete the previous symbol in the current inferior Lisp. Partial completion is supported unless `ilisp-prefix-match` is set to `t`. (If you set it to `t`, inferior Lisp completions will be faster.) With partial completion, '`p--n`' would complete to '`position-if-not`' in Common Lisp. If the symbol follows a left paren or a '`#'`', only symbols with function cells will be considered. If the symbol starts with a '`*`' or you call with a positive prefix all possible completions will be considered. Only external symbols are considered if there is a package qualification with only one colon. The first time you try to complete a string the longest common substring will be inserted and the cursor will be left on the point of ambiguity. If you try to complete again, you can see the possible completions. If you are in a string, then filename completion will be done instead. And if you try to complete a filename twice, you will see a list of possible completions. Filename components are completed individually, so '`/u/mi/`' could expand to '`/usr/misc/`'. If you complete with a negative prefix, the most recent completion (symbol or filename) will be undone.

M-RET (complete)
will complete the current symbol to the most recently seen symbol in Emacs that matches what you have typed so far. Executing it repeatedly will cycle through potential matches. This is from the TMC completion package and there may be some delay as it is initially loaded.

4.14 Miscellany

Indentation, parenthesis balancing, movement and comment commands.

TAB (indent-line-ilisp)
indents for Lisp. With prefix, shifts rest of expression rigidly with the current line.

M-C-q (indent-sexp-ilisp)
will indent each line in the next sexp.

M-q (reindent-lisp)
will reindent the current paragraph if in a comment or string. Otherwise it will close the containing defun and reindent it.

C-a (bol-ilisp)
: will go after the prompt as defined by `comint-prompt-regexp` or `ilisp-other-prompt` or to the left margin with a prefix.

DEL (backward-delete-char-untabify)
: converts tabs to spaces as it moves back.

C-Z ; (comment-region-lisp)
: will put prefix copies of `comment-start` before and `comment-end`'s after the lines in region. To uncomment a region, use a minus prefix.

C-Z) (find-unbalanced-lisp)
: will find unbalanced parens in the current buffer. When called with a prefix it will look in the current region.

] (close-all-lisp)
: will close all outstanding parens back to the containing form, or a previous left bracket which will be converted to a left parens. If there are too many parens, they will be deleted unless there is text between the last paren and the end of the defun. If called with a prefix, all open left brackets will be closed.

5 ILISP Customization

Starting a dialect runs the hooks on `comint-mode-hook` and `ilisp-mode-hook` and then *DIALECT*`-hooks` specific to dialects in the nesting order below.

```
common-lisp
  allegro
  clisp-hs
  cmulisp
  kcl
    akcl
      gcl
      ecl
    ibcl
  lispworks
  lucid
    liquid
scheme
  chez
  guile (yep! Here it comes. Still incomplete though)
  oaklisp
  Scheme->C (still "in fieri")
  scm
  snow
  stk
xlisp
  xlispstat
```

On the very first prompt in the inferior Lisp, the hooks on `ilisp-init-hook` are run. For more information on creating a new dialect or variables to set in hooks, see '`ilisp.el`'.

ILISP Mode Hooks:

`ilisp-site-hook`
: Executed when file is loaded

`ilisp-load-hook`
: Executed when file is loaded

`ilisp-mode-hook`
: Executed when an ilisp buffer is created

`ilisp-init-hook`
: Executed after inferior Lisp is initialized and the first prompt is seen.

DIALECT`-hook`
: Executed when dialect is set

Variables you might want to set in a hook or dialect:

`ilisp-prefix`
: Keys to prefix ilisp key bindings

`ilisp-program`
: Program to start for inferior Lisp

`ilisp-motd`
: String printed on startup with version

`lisp-wait-p`
: Set to T for synchronous sends

`ilisp-handle-errors`
: Set to T for ilisp to handle errors from the underlying Lisp.

`ilisp-display-output-function`
: The name of a function which displays ILISP output.

`lisp-no-popper`
: Set to T to have all output in inferior Lisp

`lisp-show-status`
: Set to nil to stop showing process status

`ilisp-prefix-match`
: Set to T if you do not want partial completion

`ilisp-filter-regexp`
: Input history filter

`ilisp-filter-length`
: Input history minimum length

`ilisp-other-prompt`
: Prompt for non- top-level read-eval print loops

6 Dialects

A *dialect* of Lisp is a specific implementation. For the parts of Common Lisp which are well specified, they are usually the same. For the parts that are not (debugger, top-level loop, etc.), there is usually the same functionality but different commands.

ILISP provides the means to specify these differences so that the ILISP commands will use the specific command peculiar to an implementation, but still offer the same behavior with the same interface.

6.1 Defining new dialects

To define a new dialect use the macro `defdialect`. For examples, look at the dialect definitions in '`ilisp-acl.el`', '`ilisp-cmu.el`', '`ilisp-kcl.el`', '`ilisp-luc`'. There are hooks and variables for almost anything that you are likely to need to change. The relationship between dialects is hierarchical with the root values being defined in `setup-ilisp`. For a new dialect, you only need to change the variables that are different than in the parent dialect.

6.2 Writing new commands

Basic tools for creating new commands:

`deflocal` Define a new buffer local variable.

`ilisp-dialect`
: List of dialect types. For specific dialect clauses.

`lisp-symbol`
: Create a symbol.

`lisp-symbol-name`
: Return a symbol's name

`lisp-symbol-delimiter`
: Return a symbol's qualification

`lisp-symbol-package`
: Return a symbol's package

`lisp-string-to-symbol`
: Convert string to symbol

`lisp-symbol-to-string`
: Convert symbol to string

`lisp-buffer-symbol`
: Convert symbol to string qualified for buffer

`lisp-previous-symbol`
: Return previous symbol

`lisp-previous-sexp`
: Return previous sexp

`lisp-def-name`
: Return name of current definition

`lisp-function-name`
: Return previous function symbol

`ilisp-read`
: Read an sexp with completion, arglist, etc

`ilisp-read-symbol`
: Read a symbol or list with completion

`ilisp-completing-read`
: Read from choices or list with completion

Notes:

- Special commands like arglist should use `ilisp-send` to send a message to the inferior Lisp.
- Eval/compile commands should use `eval-region-lisp` or `compile-region-lisp`.

Concept Index

Key Index

]

C

D

L

M

Command Index

Commands available via *M-x* prefix.

Variable Index

Variables and hooks of ILISP.

Function Index

Internal functions of ILISP which can be used to write new commands.

C

D

E

I

L

Table of Contents