Vanilla Lisp Shell (VLS)

Table of Contents

The Vanilla Lisp Shell (VLS) is designed to provide an Emacs interface to a Lisp process that works basically the same way for every flavor of Lisp.

1 Introduction

The Vanilla Lisp Shell (VLS) is designed to provide an Emacs interface to a Lisp process that from the user's perspective works basically the same way for every flavor of Lisp. For example `M-RET` evaluates an expression and `C-c C-b` produces a back-trace regardless of the type of Lisp. VLS will work with any Lisp specification such as Common Lisp or Scheme and any Lisp implementation such as Allegro Common Lisp or CMU Common Lisp.

This flexibility is achieved by Lisp type specifics files that have a simple syntax for associating common symbols with specific Lisp command strings. Those common symbol values are then used by VLS commands in forming a dialog with a specific Lisp process. VLS provides a comprehensive set of type specifics files based on current Lisp implementations, but the user can have his own set of type specifics files and edit them for customized effects.

VLS tries as much as possible to make the VLS commands work exactly the same way whether in the Lisp shell buffer or in a Lisp source code file buffer. This philosophy allows the Lisp shell buffer to work more like a free form scratch pad rather than a sequential prompt enter paradigm; although the user may operate that way also if that is what they are accustomed to.

VLS tries to be as intelligent as possible. For example when evaluating a form in a Common Lisp file VLS will search the source file and automatically put the Lisp process in the correct package before evaluating the form.

Along with the expected Lisp shell capabilities VLS also provides sophisticated Lisp tools. One such tool is generalized source code instrumenting. Conditional breakpoints are just one example of a source code instrument. VLS provides a small useful set of instruments and a facility to make it easy for the user to add custom instruments.

2 Notation Conventions

VLS commands are presented using the following form

`Command:` *command-name* `Keys:` *key-sequence* `Action:` *description*

where the user can execute the command with M-x *command-name* or with the default key sequence *key-sequence*. And *description* is a description of what the command does. Most of these commands are covered in the Vanilla Commands chapter, See Chapter 5 [Vanilla Commands], page 8.

Some VLS commands will have a universal action that has the same *description* for all Lisp types but more specific meanings or specific meanings for prefix arguments for a particular Lisp type. These are covered under the specific Lisp section under Other Flavored Commands, See Chapter 6 [Other Flavored Commands], page 13. These specific descriptions are meant to augment the universal command form above using the specific form

`Command:` *command-name* `Lisp Type Specific Actions:` *description*

where *description* in this case describes the specific Lisp actions of *command-name*.

At a few points in this document we refer to the operating system variables `$LISPDIR` and `$VLSLIBDIR`. Please note that these variables are *not* set up by VLS or required to be set up by the user. They are only used in this document to designate two important directories.

The variable `$LISPDIR` designates the directory where the VLS program has been installed. And the variable `$VLSLIBDIR` designates the directory where the VLS `types` directory is stored. For example the default designated values for these under Unix are

```
LISPDIR = /usr/local/share/emacs/site-lisp
VLSLIBDIR = /usr/local/lib/vls
```

All Lisps have the concept of `nil`, more or less. Common Lisp does in fact were `nil` means a constant symbol denoting false, the empty list and is interpreted as the end of a list in a cons that has a `cdr` of `nil`. Some Lisps do not have the concept of `nil` by default. Scheme for example does not. In Scheme the closest thing to `nil` is the constant `#f` and the empty list. When `nil` is used in this document, regardless of the Lisp type, please substitute the obvious meaning. Unfortunately it would have been too tedious to use a specific Lisp meanings in all places where `nil` occurs in VLS documents and Elisp commands.

3 Getting Started

This chapter gets you started. Most likely if you are reading this document VLS is already installed and you do not need to read the section on Installing VLS. The Lisp Modes section is only important if you are using a non-lisp-mode Lisp such as Scheme.

The Sample Lisp Shell Command section, See Section 3.3 [Sample Lisp Shell Command], page 5, will show you how to use the sample VLS Lisp shell command `vlsc`. You might actually want to use this sample Lisp shell command as your permanent command. Other varieties of Lisp shell commands are also possible with VLS, See Chapter 9 [Creating VLS Shell Commands], page 31.

3.1 Installing VLS

To make VLS unpack it in some directory with

```
tar -zxvf tar-file
```

where *tar-file* is the VLS tar file with the `tgz` file extension. This will create a `vls` sub-directory there. Change to the `vls` sub-directory and type either

```
./configure
```

or

```
./configure --prefix=PATH
```

By default the first will prepare to install VLS under the default directory `/usr/local`. The second will prepare to install VLS under a directory PATH that you provide.

Then type

```
make
```

To install the made VLS system depending on the operating system and installation PATH you may need to log in as a super user. Then type

```
make install
```

The key files that get installed are

```
PATH/share/emacs/site-lisp/vls.el
PATH/share/emacs/site-lisp/vls.elc
PATH/lib/vls/types
PATH/lib/vls/extra
PATH/info/vls.info
PATH/doc/vls/vls.html
```

where PATH is the installation directory.

To get started you should read the VLS user document. A quick start can be tried by just reading the Getting Started chapter. The `vls.html` file can be read in a web browser. So for example if the above PATH was `/usr/local` you can use the URL

```
file:/usr/local/doc/vls/vls.html
```

Or if you want to read the VLS user document in Emacs with the `info` command, while in Emacs type

```
C-u C-h i /usr/local/info/vls
```

You can also get a PostScript copy of the VLS document by changing to the directory where you typed `./configure` and then change to the sub-directory `doc` and type

```
make ps
```

which will create a PostScript version of the VLS user document called

```
vls.ps
```

3.2 Lisp Modes

If your Lisp files use `lisp-mode` then you *do not* need to read this section. If while visiting one of your Lisp code files you look at the variable `major-mode` (C-h v major-mode) and it reports its value as `lisp-mode` then your are using `lisp-mode`.

However if your Lisps uses a Lisp mode other than `lisp-mode`, for example Scheme uses `scheme-mode`, then it is important to VLS that this be known before VLS is loaded. This is so that VLS can set its default key bindings into the user's preferred mode. To make this happen easily VLS provides a variable `vls-modes` that is a list of the viable lisp mode that VLS should use. But default it is simply the list

```
(lisp-mode)
```

If you intend to use other than the `lisp-mode` with VLS then you need to set `vls-modes` *before* you load VLS. For example if you intended to use Scheme and Common Lisp your need to set in your `~/.emacs` file

```
(setq vls-modes '(lisp-mode scheme-mode))
```

and then load and run VLS.

3.3 Sample Lisp Shell Command

VLS provides a sample Lisp shell command called `vlsc`. We call it a "sample" since there are many possible kinds of lisp shell commands possible. But the `vlsc` Lisp shell command described in this section may be good enough for your permanent Lisp shell command. For a new VLS user it is highly recommended that you start off with `vlsc` and consider more interesting things later, See Chapter 9 [Creating VLS Shell Commands], page 31.

To understand what to substitute for the variables `$LISPDIR` and `$VLSLIBDIR` used in this section see the section on Notation Conventions, See Chapter 2 [Notation Conventions], page 3. You will want to put the following in your `~/.emacs` file

```
; Autoload vlsc command.
(autoload 'vlsc "$LISPDIR/vls")

; Make C-z a prefix key.  Note that suspend-emacs is also on C-xC-z.
(if (not (keymapp (key-binding "\C-z"))) (global-unset-key "\C-z"))

; Bind vlsc command to keys.
(global-set-key "\C-z\C-l" 'vlsc)
```

The `autoload` will allow VLS to be loaded when you invoke the `vlsc` command and the `global-set-key` will bind the `vlsc` command to the keys `C-zC-l` If you want to bind it to something else change the `\C-z`s and `\C-z\C-l` to something else.

The `vlsc` command picks a specific Lisp shell to run from a list of shell specs. These shell specs are in a variable `vlsc-shells` that the user must set. Each of these shell specs in turn is a list of three items: a Lisp shell id, a Lisp executable and a Lisp types specifics file. For example if addition to the above you put in your `~/.emacs`

```
; Set my list of shells for vlsc command.
(setq vlsc-shells
      '((allegro "lisp" "$VLSLIBDIR/types/allegro.el")
        (allegro-1 "lisp" "$VLSLIBDIR/types/allegro.el")
        (cmucl "cmucl" "$VLSLIBDIR/types/cmucl.el")))
```

the first entry says create a shell called `allegro` with a buffer called `"*allegro*"` that then runs the executable `lisp` and sets up that particular shell based on the contents of the Lisp types file `types/allegro.el`. And the same for `allegro-1` and `cmucl`. If you want to run more or different Lisps edit the above with differently named shell ids, Lisp executables and their associated Lisp type file. If you look in the `$VLSLIBDIR/types` directory you should find one compatible with your Lisp. If not you will have to create your own Lisp type file, See Chapter 8 [Type Specifics Files], page 22.

Then when you rerun Emacs you should be able to type the keys you bound to `vlsc` and it should invoke the first shell you gave in `vlsc-shells`. Typing that subsequently will return to that established shell buffer. A numeric prefix argument N given to the `vlsc` command will invoke the Nth shell. A `C-u` prefix argument will list the Lisp shells invoked by prefix argument N where N = 0 is the same as no prefix argument.

4 Current Lisp

To use VLS it is important to understand the concept of the Current Lisp and a Lisp Shell Buffer. When you run a Lisp Shell using VLS it creates a Lisp shell buffer, runs a Lisp executable as a process and then associates that Lisp process with that buffer. It also makes that buffer's Lisp process the Current Lisp. Whenever VLS sends any Lisp expressions it always sends them to the Current Lisp. So for example if you were to evaluate an expression in any Lisp file buffer it would send the expression to the Current Lisp process and would pop-up the Lisp Shell buffer associated with the Current Lisp process and print the returned value in that buffer.

If there are multiple VLS Lisp Shell buffers only one will be associated with the Current Lisp process. There are three ways to make a VLS Lisp Shell the current Lisp: 1. When you invoke that Lisp Shell for the first time. 2. While in a Lisp Shell buffer you execute any VLS command. 3. You manually make the Current Lisp Shell buffer the Current Lisp with the `vls-make-current` command, See Section 5.2 [Shell Buffer Commands], page 9.

Note that when you invoke a Lisp Shell it will only become the Current Lisp on the initial invocation. All subsequent invocations will simply switch to that Lisp Shell buffer. This is so that you can easily switch between different Lisp process buffers to examine contents without worrying about changing the Current Lisp.

5 Vanilla Commands

This chapter describes Emacs VLS commands that work the same way regardless of the Lisp specification or implementation. In some rare cases a specific Lisp may completely change the semantics of the vanilla command, See Section 8.4.2 [Mechanism Specifics Parameters], page 26. The Emacs documentation of such commands will reflect accurately what it does and it usually does what you would expect the vanilla command to do in that special Lisp type case. In some common cases where the vanilla command for a specific Lisp just differs in a small way, such as a slightly different meaning of a command prefix argument, those command differences are covered in another chapter, See Chapter 6 [Other Flavored Commands], page 13. In all cases the Emacs documentation (`C-h d` or `describe-function`) for any VLS command should accurately reflect what the command does for the current Lisp process.

The Instrumentation commands are in this same category of vanilla commands but deserve a separate chapter, See Chapter 7 [Instrumenting Code], page 15.

5.1 Eval Commands

When we talk about "eval" commands we mean commands that send some data to the Lisp process for interpretation. A common case of this is sending a Lisp s-expression to the Lisp process read-eval-print loop. And in all cases when the Lisp process returns a value or it wants to print some text, the Lisp shell will print the value or values or text in the Lisp shell buffer.

The basic Lisp read-eval-print loop is simple and the only thing that it can evaluate is a Lisp s-expression. But modern Lisps are not this simple and allow other kinds of interpretation. For example non-s-expression commands to a debugger are now vogue. These kinds of modern divergences from the old basic Lisp read-eval-print loop creates two interesting issues for the modern Lisp shell. The first is that there are many different classes of things to evaluate or interpret: s-expression, a line, a region, a letter, a word ... etc. The second issue is that it opens up the possibility of different kinds of evaluation paradigms. For example one paradigms might be to operate the Lisp shell buffer as something like a prompt-enter-return paradigm of a Unix shell. Another paradigm might be to operate the Lisp shell buffer as a free form scratch pad. VLS attacks deals with these issues by giving the user all possible choices rather than imposing a single style.

The default settings are more inclined to more of a free form scratch pad paradigm. But VLS makes it easy for the user to set up things they way they like it. The way that VLS does this is to have a one eval command `vls-eval` that is easily customized. Basically the way that `vls-eval` works is to execute one of a collection of evals commands that VLS or the user provides. The user can easily change these evals commands in his `~/.emacs` file.

First we describe `vls-eval` and then each of the commands that by default it can execute.

Command: `vls-eval` Keys: `M-RET` Action: Execute the 0th (first) eval command on the list `vls-evals`. By default this is `vls-eval-previous`. A numeric prefix argument N will execute the Nth eval command in `vls-evals`. A `C-u` prefix argument will list all of the eval commands in `vls-evals`.

Variable: `vls-evals` Meaning: Contains a list of eval commands to be executed by `vls-eval`. The user can set this list to their choosing by setting it in their `~/.emacs` file. For example

```
(setq vls-evals '(vls-send-input my-special-eval vls-eval-previous))
```

would cause `vls-eval` to execute the `vls-send-input` command. And with a prefix argument of 1 it would execute a command called `my-special-eval`.

To have `vls-eval` execute a command with a specific prefix argument just specify a list form in `vls-evals` that is similar to a function call with arguments. For example if in the previous example `my-special-eval` does an even more special thing with a prefix argument of 9 you could do something like

```
(setq vls-evals '(my-special-eval vls-eval-previous (my-special-eval 9)))
```

Then `vls-eval` with no prefix argument would execute `my-special-eval` and `vls-eval` with a prefix argument of 2 would execute `my-special-eval` as if given a prefix argument of 9.

Command: `vls-eval-previous` Action: Send the previous s-expression from point to the current Lisp process. This is the default evaluator for `vls-eval` and supports the idea of a free form scratch pad paradigm for talking to the Lisp process because any expression or sub-expression in the Lisp shell buffer, Lisp file buffer or any Lisp buffer can be evaluated with this command. To send Lisp data that is not on the form of a s-expression use `vls-send-input` or `vls-send-region`.

Command: `vls-send-input` Keys: `C-c RET` Action: Send the last entered input into the Lisp shell buffer to the current Lisp process. Note that this command is not intended to work on other points in the buffer other than the last entered data and *only* works in a VLS shell buffer. It is also useful for sending non-s-expression data to the current Lisp process from the current Lisp shell buffer.

Command: `vls-send-region` Action: Send the Emacs region (from point ro mark) to the current Lisp process.

Command: `vls-eval-definition` Keys: `C-M-x` Action: Send the top level definition that point is in in to the current Lisp process.

5.2 Shell Buffer Commands

This section describes the VLS commands related to the Lisp Shell Buffer itself and associated Lisp process.

Command: `vls-exit-lisp` Keys: `C-c !` Action: Exits the Lisp process associated with the current Lisp shell buffer and asks to delete the buffer.

Command: `vls-make-current` Keys: `C-c C-x c` Action: Makes the current buffer, if it is a VLS Lisp Shell buffer, the Current Lisp shell buffer and process.

Command: `vls-interrupt-lisp-process` Keys: `C-c C-c` Action: Interrupts the current Lisp process and usually causes an immediate exception stack break in the current Lisp buffer so that the user can examine the state of the Lisp process.

Command: `vls-kill-last-output` Keys: `C-c C-o` Action: Kills all output from the Lisp process since the last manual eval in the current Lisp Shell Buffer. The output is

placed onto the top of the Emacs kill stack so that you can yank it back with the `yank` command.

Command: `vls-yank-last-input` Keys: `C-c C-y` Action: Yanks the last input set to the current Lisp process at point. Successively repeating this command will yank previous inputs from the input ring of the current Lisp process. Each Lisp shell has its own input ring.

Variable: `vls-input-ring-size` Meaning: Indicates the maximum size of Lisp shell input rings. The default setting is 16.

Command: `vls-focus-process` Keys: `C-c C-x f` Action: Some Lisps have a multiprocessing capability. For the ones that do there is usually an available list of processes. A VLS Lisp Shell can only be focused on one of those processes at at time. What it means to be focused on a Lisp process in a VLS Lisp shell buffer is that when you send s-expressions or other data to the Lisp process to be evaluated the focused process is the one that will receive the s-expressions or data and respond. In other terminology the focused process is the active Lisp Listener. This command focuses the current Lisp shell on the first process in the current Lisp process list. A numeric prefix argument N focuses on the Nth Lisp process. A `C-u` prefix argument prints the list in the Lisp shell buffer so that you know which is which.

Command: `vls-reload-specifics` Keys: `"C-c C-x R"` Action: Reloads the Lisp specifics file associated with current Lisp Shell buffer. This is useful when you want to modify a Lisp specifics file and want it to take effect without rerunning the Lisp shell.

5.3 Lisp File Commands

Most VLS Lisp commands work the same way whether in a Lisp file buffer or in a Lisp shell buffer. The commands in this section have a slightly different meaning if the current buffer does not have a file associated with it; in that case one will be prompted for. This provides a convenient way to operate on a prompted Lisp file by going to a VLS Lisp shell buffer and executing any one of these commands.

Command: `vls-compile-file` Keys: `C-c c` Action: The current buffer's Lisp file is compiled in the current Lisp process. A `C-u` prefix argument will evaluate and compile just the current definition that point is in.

Command: `vls-load-source` Keys: `C-c l` Action: The current buffer's file Lisp source is loaded into the current Lisp process. A `C-u` prefix argument causes the result of each expression to be printed in the current Lisp buffer as it is loaded.

Command: `vls-load-source-or-binary` Keys: `C-c C-l` Action: The current buffer's file Lisp source or binary file is loaded into the current Lisp process. Which ever is loaded depends on the rules for the current Lisp for what gets loaded when a file minus it's extension is specified to be loaded. A `C-u` prefix argument causes the result of each expression to be printed in the current Lisp buffer as it is loaded.

5.4 Information Commands

This section lists the VLS commands for getting information from the current Lisp process or VLS program.

Command: `vls-apropos` Keys: `C-c i a` Action: In Lisps that have an apropos facility do an apropos on the symbol that point is on.

Command: `vls-describe` Keys: `C-c i d` Action: Describe the symbol of the current Lisp process that point is on in the current buffer.

Command: `vls-help` Keys: `C-c h` Action: Prints out a list of Lisp special mode commands in the current Lisp shell buffer. Typically this list is useful when debugging but in some Lisps can contain other helpful commands.

Command `vls-version` Keys: `C-c i v` Action: Show the currently running version of VLS.

5.5 Package Commands

Some Lisps, such as Common Lisp, have symbol package systems. This section only applies to those kinds of Lisps.

When evaluating expressions in a Lisp file with any VLS eval command VLS automatically searches the file for an expression that declares what package a following expression should be evaluated in. For example in Common Lisp this is an `in-package` expression. If for some reason you don't want that to happen you can narrow the Lisp file buffer to just the expression or region being evaluated.

VLS also provides the following commands for easily setting and inquiring what the symbol current package is.

Command: `vls-what-package` Keys: `C-c C-p` Action: Reports the current Lisp current package in the current Lisp shell buffer.

Command: `vls-in-package` Keys: `C-c p` Action: Searches the current buffer backwards for an `in-package` expression and sends to the current Lisp process. A numeric prefix argument N searches for Nth previous `in-package` expression. If N is negative it searches forward for the -Nth. A `C-u` prefix argument pushes the previous `in-package` expression on the Emacs kill ring.

5.6 Debugging Commands

Most Lisps have the capability to examine the execution call stack when a program encounters an error or certain exceptions. There are usually a whole raft of stack based functions specific to each Lisp that can usually be printed with the `vls-help` command. VLS provides some of the more important ones as vanilla commands. VLS also provides some of the specific stack functionality that work differently for each specific Lisp, See Chapter 6 [Other Flavored Commands], page 13.

The `vls-up-stack` and `vls-down-stack` commands move up and down the run time stack frames. What it means to position to a stack frame is to be in the same call and variable binding environment associated with that stack frame. The direction `"down"` means earlier stack frames and `"up"` means later stack frames.

Command: `vls-back-trace` Keys: `C-c C-b` Action: Prints out a backtrace of the run time stack in the current Lisp shell buffer. For specific Lisp type prefix argument functionality, See Chapter 6 [Other Flavored Commands], page 13.

Command: `vls-current-frame` Keys: `C-c C-f` Action: Prints out information about the current stack frame in the current Lisp shell buffer.

Command: `vls-down-stack` Keys: `C-c C-d` Action: Move down the run time stack one stack frame in the current Lisp shell. A numeric prefix argument N means move down N frames. A `C-u` prefix argument means move to bottom of stack.

Command: `vls-reset-listener` Keys: `C-c C-r` Action: Reset the Lisp listeners to the top level listener. So for example no matter how nested into break listeners this command will unwind all of those to the top level listener. If given a numeric prefix argument N it will reset only N listeners. So for example if you are in a break listener and then create another error, a 1 prefix argument here will return you to the previous break listener.

Command: `vls-return-frame` Keys: `C-c r` Action: Return `nil` from the current stack frame and continue execution. With a `C-u` prefix argument enter a value to return and continue.

Command: `vls-step` Keys: `C-c C-s` Action: For Lisps that have a code stepper facility and when in a stepping mode this command steps one expression. A numeric prefix argument N steps over the next N expressions. A `C-u` prefix argument evaluates the current expression in non-stepping mode.

Command: `vls-up-stack` Keys: `C-c C-u` Action: Move up the run time stack one stack frame in the current Lisp shell. A numeric prefix argument N means move up N frames. A `C-u` prefix argument means move to top of stack.

Command: `vls-var-value` Keys: `C-c C-v` Action: For Lisps that have the capability to probe the local value of a variable symbolically this command will allow examining the variable binding. Usually this is done in some binding environment. An example would be when in a break listener examining the value of some variable within the scope of where the code broke. After executing this command Emacs prompts for a variable name in the mini-buffer and then value of that variable is then echoed in the current Lisp shell buffer. In some Lisps just entering a RET to the variable name prompt will do something special like print the local lexical environment.

Command: `vls-what-error` Keys: `C-c C-e` Action: Prints the current error in the current Lisp shell buffer. The current error was due to evaluating the last Lisp expression that was in error.

6 Other Flavored Commands

This chapter describes Emacs commands to a Lisp process that are specific to various flavors of Lisp. They are of two varieties. Those commands that are strictly unique to a specific Lisp type and those that are common to all Lisp types but have some differences. Most of the second variety are differences in the effect of prefix arguments to common commands. These differences are specified in this document as an extension to the vanilla command using notation form "Lisp Type Specific Actions:" instead of the "Actions:" forms which are used in the vanilla commands section, See Chapter 5 [Vanilla Commands], page 8.

VLS Lisp commands depend on the Lisp type specifics files, See Chapter 8 [Type Specifics Files], page 22. If a specific Lisp type does not provide a way to provide the full semantics of a vanilla command then a command specifics parameter might be missing for that Lisp type. For example some Lisps do not have multiprocessing capabilities so the multiprocessing VLS commands would not work. When executing such a command an error message will be issued saying that it did not find a command string for the parameter. The missing parameter command string may be due the author of the Lisp specifics files not knowing how or not get to defining that parameter. The user is encouraged to try and define such a parameter.

6.1 Allegro Flavored Commands

Command: `vls-back-trace` Lisp Type Specific Actions: A positive numeric prefix argument N prints the N stack frames around the current stack frame. With a `C-u` prefix argument prints all significant stack frames. With a 0 numeric prefix argument prints all stack frames.

6.2 Clisp Flavored Commands

Command: `vls-back-trace` Lisp Type Specific Actions: A positive numeric prefix argument N between 1 and 5 sets mode to mode-N then back trace. The mode holds for all following `vls-back-trace` without prefix arguments. Initially the mode is mode-4 so a prefix argument of 4 will put you back to the initial mode. A `C-u` prefix argument prints maximum stack frames (same as mode-1) and a 0 prefix argument prints just the apply stack frames (same as mode-5).

Command: `vls-return-frame` Lisp Type Specific Actions: Clisp overrides the vanilla semantics of `vls-return-frame` such that instead of with a `C-u` prefix argument prompting the user for a new value and `nil` otherwise, always prompts the user for a return/continue value in the current Lisp buffer.

6.3 CMU CL Flavored Commands

Command: `vls-back-trace` Lisp Type Specific Actions: A positive prefix argument N prints N stack frames down from the current stack frame and including the current.

6.4 GCL Flavored Commands

Command: `vls-back-trace` Lisp Type Specific Actions: A positive numeric prefix argument N prints N stack frames from top of stack. A `C-u` prefix argument prints a short version of the stack frames.

6.5 LispWorks Flavored Commands

Command: `vls-back-trace` Lisp Type Specific Actions: A positive numeric prefix argument N prints N stack frames down from current stack frame. A 0 prefix argument print all objects found in the current stack frame. A `C-u` prefix argument prints a very detailed version of all stack frames.

6.6 MIT Scheme Flavored Commands

Command: `vls-return-frame` Lisp Type Specific Actions: MIT Scheme overrides the vanilla semantics of `vls-return-frame` such that instead of with a `C-u` prefix argument prompting the user for a new value and `nil` otherwise, always prompts the user for a return/continue value in the current Lisp buffer.

6.7 Scheme Flavored Commands

This section applies to all Scheme types.

Command: `vls-help` Lisp Type Specific Actions: Scheme implementations tend to not put the user into a debugger when an error or exception is encountered. Instead the user is expected to change this behavior or explicitly put himself in the debugger with a call to `debug`. To simulate the vanilla behavior the packaged VLS Lisp type specifics files debugging commands automatically put the user into the debugger, do the action indicated by the VLS command, like for example `vls-back-trace`, and then exits the debugger. Scheme Lisp types override the vanilla command for `vls-help` to work differently from these kinds of VLS debugging commands. It puts the user in the debugger and prints the help list but does not exit the debugger. The rationale here is that if the user is asking for the Scheme help commands that only work in the debugger it is better to leave him there to try one.

7 Instrumenting Code

VLS provides a facility for making it easy to instrument your Lisp code. Code instrumentation help to both understand Lisp code and understand what is going on in difficult to understand code bugs. Most Lisp specifications provide tracing facilities. For example Common Lisp specifies a `trace` function. These tracing facilities are quite powerful and elegant but they are usually done only at the functional interface and show the outline of function input and output arguments over time. Sometimes this is not sufficient. A case in point is that when you trace a function and find that the arguments are not what you would expect. In one sense Code instrumentation takes over where code tracing leaves off and helps to get a clear visualization of what is happening inside the code. In fact you can use VLS code instrumentation to do selective tracing.

VLS provides a given set of instruments to choose from, See Section 7.3 [Given Instruments], page 17. But it is easy for you to modify or add to this set, See Section 7.4 [Creating Instruments], page 20. The section on Given Instruments not only explains the given instruments but serves as an example of using instruments.

7.1 Instrument Basics

VLS instruments code in Emacs by placing an instrument at point (see Emacs point and mark) inside of a definition in a Lisp file buffer. A VLS instrument is any s-expression. And you could do exactly what VLS does yourself by making a copy of a definition and inserting such s-expressions and reevaluating the definition. But VLS automates this process for you to make code instrumentation faster, less tedious and less error prone.

VLS maintains a copy of such instrumented definitions and tracks the instruments you assert so that you can easily instrument, re-instrument and edit the existing instruments. It also provides a list of common instruments for you to choose from when you are creating instruments and makes it easy to add your own instruments to this list. There is some science to code instrumentation that makes it appropriate for VLS to provide a given list of useful instruments. For example a conditional breakpoint is an example of such an instrument.

VLS code instruments are a form of template that allows an s-expression to have slot fillers depending on the code context in which the instrument is inserted. When you choose an instrument at the current point in a buffer VLS pops up an edit buffer with just the slot filled instrument for your examination. At this point you can accept the automatically provided instrument or edit the instrument to make it do more complex things. When the instrument passes your inspection you hit `C-cC-c` and the instrument along with all other previous instruments are inserted in the copy of the definition and the instrumented definition copy is reinstalled in the current Lisp process.

When a definition is reinstalled by VLS in this way the user has the option of installing just the interpreted version, the compiled version or have VSL ask for the option. You can do this by setting the variable `vls-compile-instrumented` in your `~/.emacs` file or by toggling it with the `vls-instrument-toggle-compile` command, See Section 7.2 [Instrument Commands], page 16.

7.2 Instrument Commands

This section contains the VLS commands for applying and using the given or user specified instruments. VLS instrument commands that operate on single definitions work by the user placing point inside or after the source definition and then executing the command. When we say "definition containing point" in this section we mean the definition that point is inside or after but not inside a subsequent s-expression. After doing this VLS automatically determines the parameters for the definition and effect of the command.

Command: `vls-instrument` Keys: `C-c SPC SPC` Action: Choose an instrument and apply at point. The user chooses an instrument from the list of instruments specified for the current Lisp and applies that instrument at point in a definition in the current Lisp buffer. A list of the specified instruments are displayed and the user chooses a number for the instrument to be inserted at point. Just a RET for this choice will choose the 0th instrument. After the instrument is accepted by the user in the VLS edit buffer the definition is installed with all existing instruments for that definition. If the definition has already been instrumented and the definition has changed from when the definition was instrumented `vls-instrument` will automatically clear all the old instruments and only include this current instrument.

Variable: `vls-compile-instrumented` Meaning: Allows you to control what happens when instrumented or uninstrumented code is installed. After instrumenting a definition, nil means don't ask don't compile (the default), t means don't ask just compile, and 'ask means ask to compile.

Command: `vls-instrument-toggle-compile` Keys: `C-c SPC t` Action: Toggle the variable `vls-compile-instrumented` between compile and do not compile instrumented code during its installation, A `C-u` prefix argument means to set the variable to ask to compile during the install. This command does not require point to be placed inside any definition.

Command: `vls-instrument-clear` Keys: `C-c SPC c` Action: Clear all instruments of the definition containing point and reinstalls from that source containing point without the instruments.

Command: `vls-instrument-clear-all` Keys: `C-c SPC C` Action: Clear all instruments associated with current Lisp process and reinstall all pre-instrumented definitions. Note that this slightly different from applying `vls-instrument-clear` to all instrumented definitions since `vls-instrument-clear` installs the definition containing point and `vls-instrument-clear-all` installs the definitions that was saved before it was instrumented. What this implies is that if any definition is edited since it was instrumented it should be cleared with `vls-instrument-clear`. Or else if such an edited definition was cleared through `vls-instrument-clear-all` the pre-edited definition will be the installed version until the edited version is reinstalled manually.

Command: `vls-instrument-display` Keys: `C-c SPC d` Action: Displays the instruments of definition containing point in a VLS buffer. Each instrument in this buffer is preceded by a comment indicating that it is an instrument. This command is useful when a the user makes an error in instrumenting the definition and wants to see what is going on in the instrumented code.

Command: `vls-instrument-display-all` Keys: `C-c SPC D` Action: Show all instrumented definitions and related information.

Command: `vls-instrument-edit` Keys: `C-c SPC e` Action: Edit the instrument closest to point in current definition. If more than one instrument is closest to point then this command will ask which one you want. This allows you to edit a previously inserted instrument. VLS brings up the same `*vls*` edit buffer that it did when you executed `vls-instrument` for the original instrument. If you clear this edit buffer below the header and accept, it results in the particular instrument being removed.

Command: `vls-instrument-again` Keys: `C-c SPC r` Action: Reinstall instruments of definition containing point. If for some reason the definition containing point was redefined this command allows you to reinstall the instrumented definition. If the definition containing point was modified `vls-instrument-again` will detect this and give the option of doing nothing or reinstalling the old definition with its instruments.

Command: `vls-instrument-view-variables` Keys: `C-c SPC v` Action: View any variables set by instruments of definition containing point. Some instruments are capable of collecting results in global variables a points in your code. VLS keeps track of these variables and this command will print the contents of such variables in the current Lisp buffer.

Command: `vls-instrument-restart-variables` Keys: `C-c SPC V` Action: Restart any variables set by instruments of definition containing point. Some instruments are capable of collecting results in global variables at points in your code. Usually these are lists or counters of such results at points in time. These variables will keep growing from one test run of your code to the next. This command allows such variables to be reset. VLS tries to do the correct thing here, for example if the accumulated variable has an integer in it it will reset it to 0, if it has a list it will reset it to nil, ... etc.

7.3 Given Instruments

To illustrate the given instruments we will use a Common Lisp example. Those non Common Lisp users will have to use their imagination here. Suppose that we have a Lisp buffer with the following code for factorial

```
(in-package test)

(defun fact (x)
  (if (= x 0) 1
    (* x (fact (1- x)))))
```

Lets say that point is just before the `if` expression. When you execute the command `vls-instrument` (`C-c SPC SPC`) described in the previous section you get the selection prompt

```
VLS Instruments
 0 = Breakpoint
 1 = Wrap selected instrument before
 2 = Wrap selected instrument after
 3 = Print variables
 4 = Capture variables
 5 = Count passes
```

and are asked to enter a number associated with one of these listed instruments. And lets say that you choose to enter the "Breakpoint" instrument. VLS would then pop-up in a `*vls*` buffer something like the following

```
Instrumenting definition: fact
    ---- Edit or enter below and type C-cC-c when done ----
(cond (t (break "test fact 1")))
```

Notice first of all that this is a conditional breakpoint with a predicate of true by default which in effect makes it an unconditional breakpoint if you accept the default. To make it a conditional breakpoint replace the true predicate with any other predicate. When you are happy with the contents of the `*vls*` buffer hit C-cC-c and the instrument will be registered and installed into the current Lisp process. At the same time VLS will pop-up a new `*vsl*` buffer with the definition and all of the current instruments in place. Each instrument will be clearly delineated with a Lisp comment.

```
Instrumented definition:

(defun fact (x)

  ;; *Instrument*
  (cond (t (break "test fact 1")))

  (if (= x 0) 1
    (* x (fact (1- x)))))
```

Now when you run the factorial function in the current Lisp shell buffer it should break at the point you placed this instrument.

We will skip the "Wrap" instruments for now and go to an explanation of the "Capture variables" since this will help explain a large number of the instrument commands and the "Wrap" instruments. First clear all the existing instruments by keeping the point inside the source of the definition `fact` with the `vls-instrument-clear` command (`C-c SPC c`). This clears out all the instruments in the `fact` function and reinstalls `fact` from the source that point is in.

In this next example we want to insert a "Capture variables" instrument so that we can run a test case and then later view the variable `x` each time that it comes into this section of the code. So position point just before the inner `fact` recursive call expression. But note that if we put an instrument at this point, just before the inner recursive `fact` call expression, the instrument itself would be an expression inside the times expression `(* x (fact (1- x)))`. This would produce an undesirable result since whatever the instrument returned would become a factor in the times expression. This is what the "Wrap" instruments are for.

In this case we would choose the "Wrap selected instrument before". What this would do is wrap an expression around the instrument followed by the `(fact (1- x))` expression such that the effect of the instrument would occur first but the result of the `(fact (1- x))` expression would be returned and passed as an argument to the times expression.[1].

[1] The astute user might ask why if VSL says that it tries to do the most intelligent thing that in this case it should automatically wrap the instrument. But the even more astute user would further notice that because of the flexibility requirement of VLS the user may actually want an instrument with a return value to be a factor. VLS takes the position that flexibility is more important in this case than trying to do what some might consider the "right" thing.

Whenever you select a "Wrap" type of instrument VLS will ask for the another instrument to be wrapped with the following expression in the Lisp buffer after point. In this case we want to give it the "Capture variables" instrument. After selecting the "Capture variables" instrument, this particular instrument will continually ask for variable names until you just type a RET by itself. So do this by entering `x RET RET`. At this point you should see something like the following in a `*vls*` buffer

```
Instrumenting definition: fact
    ---- Edit or enter below and type C-cC-c when done ----
(progn
  (cond (t (unless (boundp '/fact-capture-1) (setq /fact-capture-1 nil))
           (push (list (list 'x x)) /fact-capture-1)))
  (fact (1- x)))
```

There are a three things to notice about the features of instruments, which is why we chose this example. The first is to notice that the `(fact (1- x))` expression is wrapped in the `progn` with the instrument. This provides the wrapping effect that we have been discussing. The second thing is to notice that much like the conditional breakpoint this instrument is embedded in a conditional which in the same way allows a conditional collection of variable values.

And finally notice the introduction of the `/fact-capture-1` global variable that this instrument introduces to collect the value of the variable `x`. This global variable is specific case of VLS instrument variables used for various purposes such as collecting data. A VLS instrument variable is guaranteed to be unique among VLS instrument variables and unique among all global variables if none other have the VLS global debug variables prefix, See Section 8.4.4 [Variable Specifics Parameters], page 27. The default prefix is `"/"` but the user can easily change this prefix if for some reason the users application has other global variables with this prefix.

Then evaluate in the Lisp shell buffer

```
(fact 5)
```

it should return `120` in the Lisp shell buffer and then it should have collected the values in the VLS global instrument variable. To see this execute the command `vls-instrument-view-variables` (`C-c SPC v`). VLS should then print in the Lisp shell buffer something like

```
Instrument global variable: /fact-capture-1 =
(((X 1)) ((X 2)) ((X 3)) ((X 4)) ((X 5)))
```

It is important to note that the variable values in the list are captured in the reverse order that they occur. So in the above `x` first took on the value 5 and its last value was 1.

It is also important to note that instead of just entering the variable `x` we could have entered any number or variables or s-expressions. For example, if to the instrument prompt above for variables to capture you entered `x RET (+ x 1) RET RET`, then the result of `vls-instrument-view-variables` would produce something like

```
Instrument global variable: /fact-capture-1 =
(((X 1) ((+ X 1) 2)) ((X 2) ((+ X 1) 3)) ((X 3) ((+ X 1) 4))
 ((X 4) ((+ X 1) 5)) ((X 5) ((+ X 1) 6)))
```

VLS keeps track of such registered VLS instrument variables and the command `vls-instrument-view-variables` will print all such variables collected for the definition that

point is in. Furthermore such instrument variables under an instrument like "Capture variables" will keep accumulating on top of previous runs. If you don't want this behavior at some point then execute the command `vls-instrument-restart-variables` (`C-c SPC V`) while point is inside of the definition. VLS tries to do the smart thing with this command to reset the instrument variable. For example so that if the instrument variable is collecting a list of things it will reset the list to `nil`, if the instrument variable is an integer, it will reset the instrument variable to 0.

If you wanted the effect of the instrument to take place after the expression following point, the `(fact (1- x))` expression in our example above, then you would use the "Wrap selected instrument after" instrument. All else works the same as the "Wrap selected instrument before" including having the wrapped expression returning the value of the expression following point.

At any time you can examine the existing instruments in your instrumented definition with the command `vls-instrument-display` (`C-c SPC d`). It displays the instrumented definition with the instruments clearly delineated with Lisp comments in a `*vls*` buffer.

Also at any time you can edit the instruments in a definition using `vls-instrument-edit` (`C-c SPC e`). The way editing instruments works is that where ever point is it, will edit the closest instrument to where point is. After bringing up the `*vls*` buffer with the instrument instance to be edited everything from there on is exactly equivalent to creating an instrument in the first place. If you want to remove an instrument you can just clear the `*vls` buffer below the header that says "Edit or enter below" and then hit C-cC-c and that instrument will be removed; all other existing instruments will remain.

The "Print variables" instrument is the same as the "Capture variables" but instead of capturing the variables and their values it simply prints them out as the instrument instance is invoked.

The "Count passes" instrument is very simple and simply counts the number of passes through a point in the code in an instrument global variable.

7.4 Creating Instruments

It is easy to create your own instruments. You might need to know some Elisp depending on the sophistication of the instrument. You can look at the Lisp type specifics files at the existing ones for examples. In the VLS directory the file `types/elisp/cl.el` is one example that defines a variable `vls-cl-instruments` for Common Lisp instruments. The Common Lisp type specifics file `types/cl.el` then defines `vls-cl-instruments` as the value of the types parameter `v:instruments`. So in the case of Common Lisp you can add to this variable list of instruments or set it to your own list in your `~/.emacs` file.

If you want to create your own instruments it's a good idea to look at the given instruments as examples. However, this section give a more formal presentation for exactness.

Each Lisp type has an Lisp specifics file parameter called `v:instruments`, as we mentioned about, that defines a variable name. That variable in turn contains a list of instruments where each instrument is a list of the form

(*description template*)

Where *description* is a string describing the instrument that will appear in the selection list when the user executes the `vls-instrument` command. And *template* is an Elisp form

that returns an initial instrument Elisp string that will appear in the `*vls*` buffer when the user selects it through the `vls-instrument` command.

In the most general sense a *template* is any Elisp form that returns a string. It could be as simple as just an Elisp literal string. More typically the *template* is an Elisp `format` expression with `%s` fillers in the its first argument string. These `%s` fillers are usually filled with data based on the context of where the user is inserting the instrument in his code. For example it may be filled with a string that includes the name the definition that is being instrumented. Here is an example of the Breakpoint Instrument from the Given Instruments for Common Lisp

```
("Breakpoint"
  (format "(cond (t (break \"%s\")))" (vlsi-def-unique)))
```

The `"Breakpoint"` string is the *description* and will appear in the VLS Instruments selection list. The `format` expression is, as we explained above, as way to generate an initial instrument string instance by substituting the `%s` with the value returned by the function `vlsi-def-unique`. This function will return return a unique string that also contains the identity of the definition that is being instrumented. For a list of all provided helper functions see the next section, See Section 7.5 [Instrument Helpers], page 21.

7.5 Instrument Helpers

The previous section, See Section 7.4 [Creating Instruments], page 20, we described using `vlsi-def-unique` which is one example of a number of VLS instrument helper functions given by VLS. They all have the prefix `vlsi-`. Of course you can write your own helper substitution functions. Following are a list of the VLS helper functions and what they return.

Function: `vlsi-def-unique` Action: Return a unique string containing the identity of the definition that point is in.

Function: `vlsi-gvar-unique` Arguments: `descriptor` Action: Return a string containing the identity of the definition that point is in and the given `descriptor` string prefixed by the `v:global-debug-variable-prefix` of the current Lisp. The returned string represents a unique instrument global variable.

Function: `vlsi-get-var-pairs` Action: Prompts the user for variables and returns list of variable value pairs of the current Lisp as a string. It is a list or pairs of the form in effect

```
('symbol symbol)
```

where is `symbol` is the name of a variable that the user enters to the prompt. The effect is that each pair when evaluated in the current Lisp will result in the variable name as the first of the pair and the value of the variable as the second of the pair. It does not check what the user enters so no just variable but s-expressions and anything else can be entered .

Function: `vlsi-next-sexp` Action: Returns the next s-expression forward from point as a string.

Function: `vlsi-select-instrument` Action: The user selects an instrument as usual and this functions returns the composed instrument instance of that instrument. This is used by instruments that consume other instruments, like the `"Wrap"` instruments.

8 Type Specifics Files

As mentioned up front VLS gets its flexibility through the use of Lisp type specifics files. This chapter expands on the syntax and semantics of these files.

The reader should note that since Lisp type specifics files can be modified or substituted by the user the command descriptions in this document depend on the original VLS packaged Lisp type specifics files.

8.1 Given Specifics Files

VLS provides a set of Lisp type specifics files in the VLS installed directory under the subdirectory `types`. These files all have the `.el` file extension but only for the purpose of Elisp formatting, they are *not* Elisp files. For example `types/allegro.el`. Note however that some specifics files actually load some Lisp specific Elisp code. Such Elisp code is under the directory `types/elisp`, for example `types/elisp/allegro.el`.

8.2 Format of Specifics Files

Lisp type specifics files are *not* Elisp source files. They are files that are strictly interpreted by VLS when running a VLS Lisp shell. The only reason for giving them an `.el` extension is so that they will be in Elisp mode for formatting. And they sometimes do contain some Elisp code. And entry in a Lisp type specifics file can one of four things:

1. A string
2. A list who's first entry is another list
3. A list who's first entry is a symbol
4. An Elisp comment

Anything else, for the time being, is ignored.

In the case of (1.), a string is interpreted as an absolute or relative path name of another Lisp type specifics file to be interpreted by VLS. In the case of (2.), a list whose first element is a list then it is interpreted as a list of Lisp specifics parameters to be interpreted by VLS. In the case of (3.), a list whose first element is a symbol the list is considered to be an Elisp s-expression and it is just evaluated by Elisp. In the case of (4.), Elisp comments are just ignored.

8.3 Specifics Parameter Forms

As mentioned in the previous section a Lisp Specifics file with an entry of a list who's first entry is a list is interpreted by VLS as a list of Lisp specifics parameters. Each sublist entry is a list of the form

(*parameter-symbol parameter-value*)

Where *parameter-symbol* is the symbol referenced by the VLS code to get a specific Lisp parameter value to support some VLS generalized action. This is the key to the flexibility of VLS and what allows VLS to provide a "vanilla" Lisp shell capability, that is the capability to adapt to any flavor of Lisp without changing the way things work in VLS. The *parameter-value* then can be one of five things

1. An Elisp specifics string
2. An Elisp s-expression that returns a specifics string
3. An Elisp list of specifics strings
4. An Elisp s-expression that returns a list of specifics strings
5. Any Elisp value

In the first four cases above the semantics of a "specifics string" is an Elisp string that VLS can send to the current List process where it will be interpreted as a legitimate input. In the case of (3.) and (4.) a list of specifics strings means that VLS should send that list sequentially to the current Lisp process.

In the case of (2.) and (4.) VLS will evaluate the Elisp s-expression for either a specifics string or a list of specifics strings. And in the case of (2.) and (4.) the Elisp s-expression can reference the locally bound variable `arg` which is bound to the raw Emacs prefix argument (nil, a number or list of a number). In some cases VLS will dispatch on the Emacs prefix argument but in others the specifics parameter is expected to do that dispatching in its Elisp code. In some cases where a VLS command will dispatch on the prefix argument or does not need the prefix argument it may pass some other value in the `arg` local variable. Also if there needs to be more that one argument the rest are passed in a local variable `args` which is a list of the rest of the arguments. This document will indicate how each parameter must interpret the `arg` and `args` variable, See Section 8.4.1 [Command Specifics Parameters], page 24.

The cases (1.) through (4.) above are meant for the command parameters classified in the previous section that by convention have a `c:` prefix. The cases of (1.) and (2.) also apply to the Lisp probe specifics parameters with a prefix of `p:`. Lisp probes should not use cases (3.) or (4.) since a single return value is required for probes. The case of (5.) is for variables with a prefix of `v:` and mechanism with a prefix of `m:`.

In all cases of specifics parameters recursion can be used. That is to say, a *parameter-value* can refer to a previously established *parameter-value* recursively. And any such recursively referenced parameters must have been defined otherwise the *parameter-value* will be an Elisp `nil`. The reference is done by invoking the VLS function `vls-spec`

Function: `vls-spec` Meaning: Given a *parameter-symbol* will return its *parameter-value*.

An example of this is the Common Lisp reset to top level message parameter

```
(c:reset-top-message
  "(progn (format t \"Reset to top level\") (values))")
```

where for example the Allegro specifics parameter `c:top-listener` references this value

```
(c:top-listener (list ":reset" (vls-spec 'c:reset-top-message)))
```

The reader may wonder why strings are used in cases (1.) through (4.) instead of just s-expressions, since after all VLS only talks to Lisp processes. There are two reasons for this. First is that the syntax of some forms in any particular Lisp may not have a representation in Elisp forms. And second, some modern Lisp implementations add easy typing conveniences interpreted by their Lisp listeners in special cases, like break loops. And in these cases the easy typing conveniences are usually non-legitimate s-expressions.

8.4 Lisp Specifics Parameters

As a convention VLS Lisp specifics parameters are named with Elisp symbols that start with the following letter prefixes

```
c: command
m: mechanism
p: probe
v: variable
```

Although a convention every specifics parameter symbol name must have a colon as the second character. This is to insure that internal specifics parameter symbols will never accidentally clash with these specifics parameter symbols.

If the symbol if prefixed with a "c:" it means that the symbol defines a command parameter used by Lisp shell commands. The "p:" prefix means that the symbol defines a probe command sent to the current Lisp shell for collecting some information about the Lisp process. The "v:" prefix means that its symbol defines a variable used by VLS having a value specific to the type of Lisp. The "m:" prefix means its parameter value contains some internal VLS mechanism that is peculiar to the specific type of Lisp. It is helpful to classify the specifics parameters in this way and the subsections in this section are based on this classification.

8.4.1 Command Specifics Parameters

When you execute a VLS command in a VLS Lisp shell buffer, like `vls-back-trace`, it dispatches a command to the current Lisp process using the value of one of the specifics parameters in this subsection. These parameters must be a form that resolves to a string or strings that represents a legitimate expression that will be interpreted by the current Lisp process without error, *Specifics Parameter Forms*. If a prefix argument is specified it is passed in `arg`. If other than a prefix argument is passed in `arg` or `args` it is explained separately for each parameter.

Lisp Specific Parameter: `c:apropos` Meaning: Used by the `vls-apropos` command to do an apropos on the symbol that point in on. The symbol is passed as the first of `args`.

Lisp Specific Parameter: `c:back-trace` Meaning: Used by the `vls-back-trace` command to print a stack back-trace when in a break handler. Prefix arguments should do something implementation dependent and useful with the execution stack, like print out a more or less detailed stack or more or less of a stack.

Lisp Specific Parameter: `c:compile-file` Meaning: Used by the `vls-compile-file` command to compile the current buffer's file in the current Lisp process. The file string is passed in `arg`.

Lisp Specific Parameter: `c:current-frame` Meaning: Used by the `vls-current-frame` command to print out information about the current stack frame in the current Lisp shell buffer.

Lisp Specific Parameter: `c:compile` Meaning: Used by various commands to compile a definition who's symbol name is passed in `arg`.

Lisp Specific Parameter: `c:describe` Meaning: Used by the `vls-describe` command to describe the symbol that point in on. The symbol is passed as the first of `args`.

Lisp Specific Parameter: `c:down-stack` Meaning: Used by the `vls-down-stack` command to move down the run time stack one frame; meaning to make that frame the current frame. A positive integer prefix argument N should move down N frames. A `C-u` prefix argument should move to the bottom of the stack.

Lisp Specific Parameter: `c:exit-lisp` Meaning: Used by the `vls-exit-lisp` command to exit the Lisp process.

Lisp Specific Parameter: `c:focus-process` Meaning: For Lisps that have multiprocessing and access to a process list, this parameter is used by the `vls-focus-process` command to focus a Lisp listener on the first process in a process list. A numeric prefix argument N should focus the Nth process in a process list. A `C-u` prefix argument should print the process list in the current Lisp shell buffer.

Lisp Specific Parameter: `c:help` Meaning: Used by the `vls-help` command to print out a list of commands used by the current Lisp in the current Lisp context. Typically these are abbreviated commands as opposed to proper s-expressions, but they don't have to be.

Lisp Specific Parameter: `p:in-package` Meaning: For Lisps that have a symbol package system this expression should put the current Lisp process in a given package. The package string is passed in `arg`.

Lisp Specific Parameter: `c:load-file-print` Meaning: Used by the `vls-load-source` and `vls-load-source-or-binary` commands to load the file passed in `arg` and to print the results of each s-expression in the file.

Lisp Specific Parameter: `c:load-file` Meaning: Used by the `vls-load-source` and `vls-load-source-or-binary` commands to load the file passed in `arg`.

Lisp Specific Parameter: `c:message` Meaning: An expression that prints a given message using the current Lisp process in the current Lisp buffer. The message string is passed in `arg`. This expression *should not* return a value if possible.

Lisp Specific Parameter: `c:pop-listener` Meaning: Used by the `vls-reset-listener` command to "pop" the current listener into the previous listener. The prefix argument `arg`is always an integer N and it means to pop the listeners to the Nth previous listener. For example if the current Lisp was in a break listener and typed an expression that signaled an error into another break listener, this command were `arg = 1` would pop back to the first break listener.

Lisp Specific Parameter: `c:reset-top-message` Meaning: If needed, used by the `c:top-listener` specifics parameter to include a string s-expression that will print that the Lisp process is at top level after the first parameter's s-expression actually puts the process in top level.

Lisp Specific Parameter: `c:return-frame` Meaning: Used by the `vls-return-frame` command (usually during a break loop) to return `nil` to the current stack frame call and continue the computation. If a `non-nil` value is passed in the local `arg` variable then return that value and do the same.

Lisp Specific Parameter: `c:step-next` Meaning: Used by the `vls-step` command to step the Lisp stepper one step. If the `arg` is an integer N then step N times.

Lisp Specific Parameter: `c:step-over` Meaning: Used by the `vls-step` command to evaluate the current expression in non-stepping mode and continue stepping.

Lisp Specific Parameter: `c:top-listener` Meaning: Used by the `vls-reset-listener` command to pop the current Lisp into the top level listener regardless of how many listeners nested deep the current Lisp process is.

Lisp Specific Parameter: `c:up-stack` Meaning: Used by the `vls-up-stack` command to move up the run time stack one frame; meaning to make that frame the current frame.

A positive integer prefix argument N should move up N frames. A `C-u` prefix argument should move to the top of the stack.

Lisp Specific Parameter: `c:var-print` Meaning: This expression should pretty print if possible or just print if not possible a given variable in the current Lisp buffer. The variable string is passed in `arg`.

Lisp Specific Parameter: `p:var-reset` Meaning: This expression should a reset the value for a given variable and then print that the given variable name was reset and its new value. The variable name string will be passed in `arg`. The reset value will be based on the data type of the variable value. So if the type is number it should be 0, if the type is a list it should be the empty list, if the type is a string it should be the empty string, ... etc.

Lisp Specific Parameter: `c:var-value` Meaning: Used by the `vls-var-value` command to return the value of a local variable in a stack frame.

Lisp Specific Parameter: `c:what-error` Meaning: Used by the `vls-what-error` command to print the current error being handled by the Lisp process.

Lisp Specific Parameter: `c:what-package` Meaning: For Lisps that have a symbol package system this parameter is used by the `vls-what-package` command to display the current package in the current Lisp process.

8.4.2 Mechanism Specifics Parameters

Some specifics parameters are needed for VLS internal mechanisms, such as adding specific documentation strings to VLS functions. This section describes those parameters.

Lisp Specific Parameter: `m:more-doc` Meaning: Indicates more documentation string is to be added to the vanilla documentation string of VLS commands. Its value must be a list of two element lists were each two element list is of the form

(*function-name additional-doc-string*)

where *function-name* is a VLS function that needs additional specific documentation added to the vanilla documentation of *function-name* and *additional-doc-string* is that additional documentation string. A common use is to add additional documentation for command prefix arguments that have slightly different meaning for each specific Lisp type.

Lisp Specific Parameter: `m:replace-doc` Meaning: Indicates two things. First, different from `m:more-doc` where more documentation is added to the vanilla string, the whole vanilla documentation string is replaced by this string. Its value must be a list of two element lists were each two element list is of the form

(*function-name replacement-doc-string*)

where *function-name* is a VLS function that will have its documentation replaced by *replacement-doc-string*. And thus, second and equally important, since replacing the whole command documentation VLS assumes that any predefined logic of the command should not be used and instead the a VLS types parameter named *function-name* will return a value being a string that is sent to the current Lisp process. Note that in this case any Elisp logic for the command, if even needed, must be performed by the Elisp code, if any, specified as the parameter value. Also note that this implies that whenever a `m:replace-doc` parameter is specified a *function-name* parameter *must* be specified; otherwise the replaced command will refuse to execute with an error.

The `m:replace-doc` parameter and its implications seems rather complicated but is actually very simple, it simply triggers a specific VLS command whose current Lisp buffer specific semantics are totally different from any vanilla semantics. This is usually only necessary when a Lisp implementation does not provide enough functionality to implement the vanilla semantics but still wants to have such a specific VLS command. Clisp for example prefers to prompt the user for a return/continue value when in a break loop rather than have Emacs prompt the user with a `C-u` prefix argument as the vanilla VLS command `vls-return-frame` specifies. The Clisp type specifics file then contains the following parameters to make this happen

```
(m:replace-doc
 ((vls-return-frame
   "Return a value from the current stack frame and continue.")))
(vls-return-frame "Return")
```

The first parameter `m:replace-doc` changes the vanilla command documentation and the second parameter `vls-return-frame` required by virtue of the first, changes the behavior of the vanilla command `vls-return-frame` for Clisp to just send the string "Return" to the current Lisp process.

8.4.3 Probe Specifics Parameters

The probe specifics parameters are used by VLS to get some necessary information from the current Lisp process. Each of these parameters should be an Elisp string that contains an expression that when sent to the current Lisp will return a value. In cases where there should be instance values substituted in the Elisp string there should be `%s` characters at that point in the s-expression.

Lisp Specific Parameter: `p:bound-var-test` Meaning: A predicate that should be true if a given variable is bound in the current Lisp process. There must be a `%s` where the name of the variable will occur in the s-expression.

Lisp Specific Parameter: `p:current-package` Meaning: For Lisps that have a symbol package system this should be an expression that will return the current Lisp process package name.

Lisp Specific Parameter: `p:package-predicate` Meaning: For Lisps that have a symbol package system this expression is a predicate that should be true if the a given package exists. There must be a `%s` where the package name should occur.

8.4.4 Variable Specifics Parameters

Variable specifics parameters are specific values that are needed by the VLS interface to the current Lisp process and buffer and can not be classified as commands, probes or internal mechanism. Or in other words everything else.

Lisp Specific Parameter: `v:end-expression` Meaning: A string of a character or characters that the specific Lisp process needs to indicate that all the data that is to be sent in fact has been sent. For example in most Common Lisps this is the newline character. Note that this works for any amount or kind of data sent to the Lisp process since embedded end expression characters in the data are eaten by the parsing of s-expressions.

Lisp Specific Parameter: `v:eval-print-separator` Meaning: After evaluating a Lisp expression in the current Lisp buffer the characters in this string will be inserted in the buffer before the return value is printed. This is mean to be something like a newline or empty string depending on how the specific Lisp process returns the return value string.

Lisp Specific Parameter: `v:global-debug-variable-prefix` Meaning: Used by VLS helper functions like `vlsi-gvar-unique` to automatically create a global variable name that is unique for all practical purposes. This prefix string will start the symbol name and other parts of the variable name will be computed based on the usage.

Lisp Specific Parameter: `v:has-packages` Meaning: If this parameter exits for the current Lisp and has a non-nil value then it indicates the the current Lisp has a symbol package system. This means that the current Lisp has more than one name-space for symbols, but further means that it has some functionality for dealing with packages like the `in-package` of Common Lisp. If some Lisp is invented that has a symbols package system with a different functionality VLS may have to be augmented to deal with that new functionality.

Lisp Specific Parameter: `v:in-package-re` Meaning: If the current Lisp has `v:has-packages` non-nil and it has a form that puts the current Lisp into a new symbol package then `v:in-package-re` should be an Elisp regular expression that allows Elisp search functions to use and successfully find such an expression in a buffer.

Lisp Specific Parameter: `v:instrument-visual` Meaning: A string that serves as a comment for the current Lisp and the nature of the comment indicates and delineates an instrument s-expression follows that was inserted in the users code. The characters `%s` must appear in the string where the instrument will be inserted.

For example this string for Common Lisp defaults to

```
"\n;; *Instrument* \n%s\n\n"
```

Note the newlines which also delineate the instrument s-expression from user code.

Lisp Specific Parameter: `v:instruments` Meaning: This is an Elisp variable where the instruments for the current lisp are stored, See Section 7.4 [Creating Instruments], page 20. For example in Common Lisp the variable `vls-cl-instruments`. Having the value of this parameter be an Emacs variable rather than the instrument list itself gives the user a bit more flexibility in maintaining his own instruments.

8.5 Lisp Specifics Code

In composing Lisp specifics code, if the specific Lisp has a package system then strictly speaking one needs to make sure that all lisp code functions are of the fully qualified package form `package:function`. For example in Common Lisp specifying `lisp:car` instead of just `car`. The reason is that when sending code to the current Lisp process, the current package may not use the given package.

Considering that doing this is too pedantic VLS chooses not to do this in the given set of Lisp specifics parameters for the sake of making it easier to add specifics Lisp code. The risk of course is that if the symbols used are not in the current Lisp process package then then a VLS command may break with an undefined function. But this should be rare and seldom be a problem. If it does become a problem then a future version of VLS would

probe the Lisp process to make sure that the current package is copacetic and if not put the process in a package that is.

8.6 Dealing with prompts

It is important to understand issues of Lisp prompts in any Lisp shell. Lisp prompts can be problematical to a Lisp shell if not handled properly. Not all such things that are thus problematical are strictly speaking Lisp prompts. A "proem" is a kind of introductory text and this document uses the term "proem" to refer to such Lisp prompts that are not really prompts. In a Lisp read-eval-print loop, more modern referred to as a "Lisp listener", there is a continuous loop where the Lisp process reads an s-expression from the user, evaluating it and then returning the value and printing it. It most modern Lisps an option is to have a prompt printed asking for the user input to be read. Some Lisps will also have a proem to introduce the printed return value, like for example printing "Value: " followed by the actual return value.

For prompts where there is white-space before and after the prompt there will never be a problem in VLS. There is almost always a white-space before a prompt since traditionally read-eval-print loops emit a newline character after the printing of each return value. If not VLS provides a specifics parameter `v:eval-print-separator` for this purpose.

If there is no white-space after the prompt the VLS command `vls-eval-previous` for example might include the prompt in the value to be evaluated. For example if you specified a prompt of "Enter:" and you typed "a" to evaluate the variable `a` VLS would interpret `Enter:a` as a whole legitimate symbol, as it should it most Lisps. The obvious cure in this example is to use a prompt like "Enter: " instead with a white-space character at the end.

For VLS, proems however are problematical regardless of white space. When VLS probes the current Lisp process for values, if a proem appears in the output stream along with the value VLS has no way of knowing if the proem is part of the output or not without some extra help. If there is a proem in a specific Lisp then the specifics files should specify the `v:output-proem-regexp` parameter.

VLS was designed to allow a free form scratch pad paradigm for operating the Lisp shell where there is no prompt or proem, much like the Elisp *scratch* buffer. We encourage the user to try this kind of paradigm and there is usually a way to set your prompts to nil or the empty string so that there are no prompts. It is a lot cleaner and it makes the Lisp shell buffer almost completely the same as a Lisp file buffer. Lisp read-eval-print loops do not need a prompt since there is always a return value printed as feedback to the user that it is ready for another input. There in one exception to this that the author knows of which is in Common Lisp if a function ends with a no argument `values` expression which means that the function will return no value. But in this case it is frequently used for making a nice display without an interfering output value, much like Common Lisp's `pprint`. VLS makes it easy to get information that is typically provided in a prompt string. For example in Lisps that have symbol package systems, it is easy in VLS to view the current Lisp package at any time. All this being said the encouraged free form paradigm applies mostly to Lisp files and the Lisp shell buffer at the top level listener. In other listeners however, like a stepper or break listener prompts with information like the break level are useful however.

8.7 Using Specifics Files

VLS provides a set of Lisp Type Specifics Files for many Lisp implementations and specifications. You can use these files if they serve your purposes, you can augment those files or you can completely create your own Specifics files using the given ones as examples.

Lets say for example that you want to augment the given Lisp Specifics and just change two Lisp Specifics parameters, one of for a Lisp specification of Common Lisp and another for a Lisp implementation Allegro. Here is an example of how to do this.

First create your own Specifics file in some directory $SOMEDIR for allegro in a file $SOMEDIR/allegro.el as follows

```
$VLSLIBDIR/types/allegro.el"

;; My Allegro specific stuff examples
;;; Change cl.el c:compile-file option only known to Allegro
((c:compile-file
  (format "(compile-file \"%s\" :xref nil :verbose t)" arg))
;;; Change allegro.cl default of debug global variable prefix
 (v:global-debug-variable-prefix "/*"))
```

Then if you create a Lisp shell command

```
; Example Lisp shell command for illustration
(defun my-lisp-shell ()
  "My Lisp Shell"
  (interactive)
  (vls-shell '(my-allegro "lisp" "$SOMEDIR/allegro.el")))
```

and after executing `my-lisp-shell` the following will happen:

First `$VLSLIBDIR/types/allegro.el` will be interpreted by VLS. At this point everything would be as if you had just used the given `$VLSLIBDIR/types/allegro.el` in the above example Lisp shell command. Then the following parameters list would override the given Specifics parameters `c:compile-file` and `c:compile-file` with your versions of those parameters. And finally VLS would execute this `my-allegro` Lisp shell that will use these new parameters.

9 Creating VLS Shell Commands

VLS allows many different kinds of Lisp shell commands to be written given the basic VLS library. VLS does not want to assume that any one Lisp command is better than any other. VLS provides a simple VLS shell command called `vlsc` that we showed how to use in the Sample Lisp Shell Command section, See Section 3.3 [Sample Lisp Shell Command], page 5. But you can create your own VLS Lisp shell commands that operate in different ways. The author for example prefers to invoke VLS Lisp shells as Agroups entries, See Section 9.2 [VLS Shell Commands in Agroups], page 31.

9.1 VLS Shell Commands in General

More generally VLS is meant to be used by having a user written Lisp shell command call the `vls-shell` function. You use the `vls-shell` function by calling with

`(vls-shell` *specifics*`)`

where *specifics* is a list of the form

`(`*shell-id lisp-executable specifics-file*`)`

For example this function call

```
(vls-shell '(allegro "lisp" "$VLSLIBDIR/types/allegro.el"))
```

is an example of giving the `vls-shell` function's *specifics* input. The Lisp shell command would usually select a *specifics* input based on some user set variable or strategy. The supplied `vlsc`, command is a simple example of a user set variable containing a list of *specifics* inputs, See Section 3.3 [Sample Lisp Shell Command], page 5.

The *shell-id* is a symbol that makes a VLS Lisp shell instance unique from all other VLS Lisp shells. Additionally the Lisp shell buffer name and Lisp process name is derived from this symbol. These could be different Lisp shell types or two or more instances of the Lisp shell buffer of the same type. For example `allegro-1` and `allegro-2` could be two Lisp shell buffers running the same Lisp execute but in different Lisp processes in different buffers.

The *lisp-executable* is a string that when executed as an Emacs process will run the Lisp type. It will use the user's execution paths if necessary.

The *specifics-file* is a string containing the pathname of the VLS format Lisp specifics file of that type, See Chapter 8 [Type Specifics Files], page 22.

When you write such a command you usually specify that when the user uses your Lisp shell command the VLS function `vls-shell` will get auto-loaded. You specify this as putting

```
(autoload 'vls-shell "$LISPDIR/vls")
```

in the user's `~/.emacs` file.

9.2 VLS Shell Commands in Agroups

Action Groups (Agroups) is a powerful facility for making it easy to create. manipulate, execute and maintain groups and subgroups of automations called *actions*. Each specific action is generated from a generalized action template. Agroups is extensible and makes it

easy to add new such action templates. VLS provides such action templates for creating actions that evaluate things in Lisp shells and for running Lisp shells.

If you use Agroups it is a lot easier to create new Lisp shells without having to write any code. After installing VLS the VLS action templates are in the file

```
$VLSLIBDIR/extra/agroups-vls.el
```

All you need to do is load the VLS provided action templates after you load the Agroups facility in your `~/.emacs` file. So for example using the notation in this document you could put in your `~/.emacs` file

```
(load "$LISPDIR/agroups"
(load "$VLSLIBDIR/extra/agroups-vls")
(autoload 'vls-shell "$LISPDIR/vls")
```

Thereafter when executing the Agroups operation "View user defined Actions" you will see

```
User defined actions
  Vanilla Lisp Shell (keys: l s)
  Vanilla Lisp Shell: Evaluate a Lisp expression (keys: l e)
```

When you want to create a new VLS Lisp shell you simply execute the user defined action "Vanilla Lisp Shell" and it will prompt you with the data for a specific Lisp shell and create that specific action entry in the current Agroups group. Each time you execute that specific Lisp shell action it will create a new Lisp shell buffer Associated with that specific Lisp shell process or return to a previously such created Lisp shell buffer.

The user defined action "Vanilla Lisp Shell: Evaluate a Lisp expression" is also very powerful and allows you to have any arbitrary Lisp expression as an Agroups entry that when selected gets evaluated in the current VLS Lisp shell.

Agroups is one of the more preferred ways of using VLS Lisp shells since you can arrange specific Lisp shells and evaluation automations that are local to specific groups or projects.

10 Other VLS Interfaces

Other than Lisp shell capability VLS provides two other function interfaces to VSL. One is for sending Lisp data to the current Lisp process for evaluation and the other is for probing the current Lisp process for information. These can be used in Lisp type files or in ELisp code written by the user.

Function: `vls-send-lisp-string` Action: Given a string argument it sends the string followed by the `v:end-expression` parameter value to the current Lisp process. Any resulting output will appear in the current Lisp buffer.

Function: `vls-lisp-evalue` Action: This function is used to probe the current Lisp process for information. Given a string it sends it to the current Lisp process followed by the `v:end-expression` parameter value. The output of this function does not appear in the current Lisp buffer but instead is parsed by Elisp and returns a single Elisp expression. In other words it probes the current Lisp process, usually with an s-expression, and returns the equivalent Elisp value to be used by Elisp code. If the returned data from the current Lisp process can not be parsed by Elisp an error will be signaled.

An example of using `vls-send-lisp-string` is in the VLS provided Agroups template for creating actions to evaluate expressions in the current Lisp process, See Section 9.2 [VLS Shell Commands in Agroups], page 31. An example of using `vls-lisp-evalue` is used by VLS itself in determining the current package for Lisps that have package systems. There are many reasons why the Lisp types parameters code might want to probe the current Lisp process to get an Elisp value to perform the expected results. For example some Lisp implementation's specific `c:top-listener` parameter uses this function to determine the break level.

Variable and Command Index

V

Concept Index

Table of Contents