Chapter 3

Running Nuprl 5

3.1 System Requirements

Nuprl 5 is written mostly in Common Lisp, but uses some extensions that require Lucid, Allegro or LCMU Common Lisp and a Unix-based X window system. The implementation of CMU Common Lisp is freely available. Other Lisp versions require a license. CMUCL is faster with smaller memory footprint but currently Allegro is a bit more stable.

The Nuprl homepage provides an executable copies of the CMUCL version of Nuprl 5 running under Linux. Executable copies for Allegro can be provided upon request. The source code as well as instructions for installing Nuprl 5 to run under other Lisp and Unix versions will be made available as soon as the system has stabilized.

The Linux release of Nuprl 5 contains 3 binary executables (the library, the editor, and the refiner) that altogether require 120 MegaBytes of disk space for the CMUCL version and 40 MegaBytes for the Allegro Version. The standard library initially requires an additional 120 MegaBytes of disk space and quickly grows to 500 MegaBytes or more.

It is recommended to run Nuprl 5 on systems that have at least 256MB of RAM, 512MB of swap space, and 800MB of disk space available. Due to its implementation in Lisp, Nuprl runs more efficiently if more memory is available. In large applications it can utilize several GigaBytes of RAM. Nuprl 5 can also profit from multiple processors or a network of computers, because the library, editor, and refiner run as independent processes,

3.2 Preparation

Before Nuprl 5 can be started for the first time, it needs to be installed properly and certain configuration files must be set up for each user.

3.2.1 Retrieving and Installing Nuprl 5

The executable copy of Nuprl 5 running under Linux can be found by going to the Nuprl 5 web page `http://www.nuprl.org/nuprl5/index.html` and following the link to the download pages for the actual Nuprl 5 release.[1] To retrieve Nuprl 5, read the instructions in the various README files for up-to-date information.

[1] Currently, the release can be found at the URL `ftp://ftp.cs.cornell.edu/pub/nuprl/nuprl5` but this directory may be moved in the future.

Currently you have to download the following files.

```
install.tgz
nuprltop.tgz
nuprl5.tgz
standard-db.tgz
nuprl5-linux-cmucl-run.tgz
```

If you run Solaris instead of Linux, download the file `nuprl5-solaris-cmucl-run.tgz` instead of `nuprl5-linux-cmucl-run.tgz`. Experienced users who want to build their own NUPRL binaries or modify system tactics may also want to download the file `nuprl5-source.tgz`.

For a single-user installation, it is best to build the NUPRL system within a subdirectory of the user's home directory. For a default installation, this subdirectory should be called `nuprl`. For a multi-user installation, it is recommended to build NUPRL in a directory `/home/nuprl` and to create a user `nuprl` and a group `nuprl` that gives users controlled access to this directory.

In the following we describe the default single-user installation for Linux. Instructions for a custom installation can be found in the file `README.install`.

Move all the downloaded .tgz files into your home directory and then untar the file `install.tgz` using the commands

```
SHELL-PROMPT> cd ~
SHELL-PROMPT> tar -xzf install.tgz
```

This will build a directory `/nuprl` in which you will find an installation script `install-nuprl5.pl` as well as a file `README.install` with instructions for building NUPRL 5. As the installation procedure may change in the future, it is advisable to read these instructions before starting the actual installation.
Enter the `nuprl` subdirectory and execute the installation script.

```
SHELL-PROMPT> cd ~/nuprl
SHELL-PROMPT> ./install-nuprl5.pl
```

This will untar the other .tgz files in your home directory and build the NUPRL system within the current directory. The NUPRL knowledge base will be installed at `~/nuprl/nuprl5/NuPrlDB`.

The library, editor, and refiner processes as well as several utilities can be found in `~/nuprl/bin`. To run NUPRL 5, the directory for NUPRL binaries must be included in the user's Unix load path. To do so, add the following line to your .chsrc or .login file

```
set path = (  /nuprl/bin $path)
```

The syntax is slightly different for other Unix shells.

To support the use of special mathematical symbols in formal theorems, the NUPRL system requires some special font files to be installed. These files are contained in the directory `~/nuprl/fonts/bdf` and must be included in the font search path of the X server controlling your display.[2] Add the following lines to your .xinitrc file

```
xset fp+  /nuprl/fonts/bdf
xset fp rehash
```

These commands tell X the font path to the NUPRL fonts when the X server is first started. One may also run them interactively in some shell to add the font path to the *current* X environment.

[2]It is not sufficient to have the NUPRL-fonts available on the system that runs NUPRL 5. This may cause some complications when running NUPRL 5 remotely, particularly when the Exceed X server under Windows (NT/98/2000) is used as terminal.

Most X windows systems understand bdf font files. However, one may also compile the font files into machine-specific fonts to allow faster reading. To compile fonts, one has to use the command `bdftosnf` on Sparc stations and `bdftopcf` on Linux-PCs. After the fonts have been compiled,, one has to execute the command `mkfontdir` to make the font directory.

Sometimes the X-server has limited access to the file system. In this case the fonts files may need to be placed in a public directory. Your system administrator should be able to advise you in this case.

3.2.2 User-specific Configuration

Upon startup Nuprl 5 expects to find a file `~/.nuprl.config` in the user's home directory to determine how the individual Nuprl processes will communicate. If this file is missing, Nuprl 5 will use the configuration that was present at compile time and may not be able to establish the communication. The structure of a typical `.nuprl.config` file is as follows.

```
;; ---------------------------------------------------------------
;; Which server runs the library process?
;; ---------------------------------------------------------------
(libhost "HOSTNAME")

;; ---------------------------------------------------------------
;; Where is the location of the Nuprl Library?
;; ---------------------------------------------------------------
(dbpath "DATABASE-PATH")

;; ---------------------------------------------------------------
;; What environment within the Nuprl Library shall be used?
;; ---------------------------------------------------------------
(libenv "STANDARD-LIBRARY")

;; ---------------------------------------------------------------
;; Optional settings
;; ---------------------------------------------------------------
;; Which X-server shall be used to display the Nuprl windows?
;; ---------------------------------------------------------------
;; (eddhost "localhost" 0)

;; ---------------------------------------------------------------
;; What sockets shall be used for communication?
;; ---------------------------------------------------------------
;; (sockets SOCKET1 SOCKET2)

;; ---------------------------------------------------------------
;; How to identify the user?
;; ---------------------------------------------------------------
;; (iam "USER-NAME")

;; ---------------------------------------------------------------
;; Color and size of displayed windows
;; ---------------------------------------------------------------
;; (foreground "FOREGROUND-COLOR'')
;; (background "BACKGROUND-COLOR")
;; (font "FONT-NAME")
```

The HOSTNAME for the `libhost` configuration should be the name of the host running the library process. Currently even a stand-alone machine needs a host name.

The values for `dbpath` and the `libenv` describe the physical and logical location of the standard library. In the default single-user installation, it should be `~/nuprl/nuprl5/NuPrlDB`. In a multi-user installation, it is usually `/home/nuprl/nuprl5/NuPrlDB`. The value of STANDARD-LIBRARY is usually `standard`.

All other entries of in the `.nuprl.config` file are optional. Users may redirect the NUPRL windows to a specific X server or choose a specific set of sockets for communication between the NUPRL processes. If several users shall work with the same library, the socket numbers should be identical to the ones used by the joint library process. Specific users can be identified by the system, which will be necessary for granting controlled access to certain parts of the library.

The `foreground` and `background` colors set the colors for the NUPRL windows and can be chosen according to personal preferences. Users may also chose a font for the NUPRL windows. By default, `nuprl-13` is being used but users may also select any other NUPRL fonts or other fonts that are consistent with them.[3]

The NUPRL 5 editor will read the file `~/mykeys.macro` to determine any user key bindings for motion and macro commands. If this file is missing, the key bindings that were present at compile time will be used. The standard bindings of the NUPRL 5 system are listed in the file `~/nuprl/nuprl5/sys/macro/keys.macro`. Users who wish to customize their NUPRL 5 key bindings may copy this file into the file `~/mykeys.macro` and modify it according to their needs.

It is helpful for the user to become familiar with an editor like emacs (version 19 and higher) that supports 8-bit fonts and has a capability for starting sub-shells. The editor should be run with one of the nuprl fonts. This is not strictly necessary, but is a good idea for several reasons:

- Each NUPRL process runs a "top loop" in the same window as the one from which it was started up. It accepts input from that window and frequently writes output to it. If NUPRL is started up from an editor sub-shell, it becomes easy to review this output and save portions of it to files. Editing capabilities for the input are sometimes useful as well.
- Some of NUPRL's output is in NUPRL's 8-bit font.
- Listings of theory files use NUPRL's 8-bit font. These files contain definitions, theorems and proofs, and it is often useful to be able to browse them.

3.3 Starting NUPRL 5

The basic NUPRL 5 configuration consists of three separate processes: a library, an editor, and a refiner. In single-user mode you have to start all three processes. In multi-user mode you will connect to an already running library process and only have to start an editor and an optional refiner[4] after setting up your `.nuprl.config` file accordingly. It is important to initialize the library before the editor and the refiner.

Generally it is a good idea to run the NUPRL 5 processes in separate emacs frames. In addition to editor support for the corresponding top loops this also allows you to define an interactive emacs command that starts all three processes and initializes them correctly. The NUPRL distribution provides a few utilities for starting the NUPRL 5 processes from within emacs buffers. Consult the file `README.running-nuprl` for instructions how to use them.

The next three subsections describe three interactive emacs commands `nulib`, `nuedit`, and `nurefine`. If you add these and the following definition of the emacs command `nuprl5` to your `.emacs` file, you may start NUPRL 5 from emacs by typing ⌞⟨M-x⟩nuprl5 ↵⌟.

[3]The table of NUPRL's special characters (Table 5.1 on page 73) is actually based on a font `nuprl-8x13`, which extends `nuprl-13` by a few special characters with code 204 and higher.

[4]Actually, it may not even be necessary to start a new NUPRL 5 refiner, as the editor will connect to refiners that are already running. However, this would mean that you need to share that refiner with other users, which may cause unnecessary delays if the refiner is busy. Generally, it is recommended that you start your own refiner unless you only intend to browse the library.

```
(defun nuprl5 ()
  (interactive)
  (message "Starting NuPRL 5 Library, Editor, and Refiner ...")
  (nulib)
  (sleep-for 5)
  (nuedit)
  (nurefine)
)
```

It will take several minutes until all the NUPRL editor windows will begin to pop up, because initially there is a lot of communication between the editor and the library.

3.3.1 Starting the Library

The library process should be started first because both the editor and the refiner rely on information that is explicitly stored in the knowledge base (to simplify customization of these processes). In a shell, enter the command ⌞nulib ↵⌟.

```
SHELL-PROMPT> nulib
```

A Lisp session will start, followed by system messages. At the Lisp USER prompt enter ⌞(top) ↵⌟.

```
USER(1): (top)
```

This will start an ML *top loop* with some library-specific commands preloaded. At the ML prompt, enter ⌞go. ↵⌟ to initialize the NUPRL 5 library.

```
    CURRENT TIME : TIME AND DATE

ML[(ORB)]> go.
```

The library process will now use the information in the file `.nuprl.config` to load the desired library environment and to open the sockets for communication. It will write some system messages to the process window and then wait for other processes to connect.

The following emacs script defines an interactive function `nulib` that performs all the above steps in a new emacs shell. Using the function `nuprl-frame` it pops up a new frame at a specific position on the screen, opens a shell process `*NuLibrary*` in it, and then subsequently sends the above command to that process.

```
(defun nuprl-frame (bufname height top-corner cmd)
  (save-excursion
    (set-buffer (make-comint bufname "/bin/csh" nil "-v"))
    (switch-to-buffer-other-frame (concat "*" bufname "*" ))
    (let ((NuPRLframe (car (cadr (current-frame-configuration)))))
      (set-frame-size     NuPRLframe 81  height)
      (set-frame-position NuPRLframe 515 top-corner)
    )
    (set-default-font "6x10")
    (while (= (buffer-size) 0) (sleep-for 1))
    (comint-send-string bufname "limit coredumpsize 0\n")
    (comint-send-string bufname cmd)
) )

(defun nulib ()
  (interactive)
  (nuprl-frame "NuLibrary" 10 76 "nulib\n")
  (message "Starting Library ...")
  (set-foreground-color "Red")
  (set-background-color "#ddddff")
  (comint-send-string "NuLibrary" "(top)\n")
  (comint-send-string "NuLibrary" "go.\n")
)
```

The shell script is designed for an XGA (1024x786) display and uses a fairly small font. You need to adjust the frame position on larger displays and if a larger font is chosen. The colors are chosen to distinguish the library frame from the other windows.

Experienced users can make further use of the function `comint-send-string` to send additional commands to the NUPRL 5 process at their convenience.

3.3.2 Starting the Refiner

Starting the refiner is similar to starting the library. In a shell, enter the command ⌞`nuref` ↵⌟.

```
SHELL-PROMPT> nuref
```

At the Lisp `USER` prompt enter ⌞`(top)` ↵⌟.

```
USER(1): (top)
```

At the ML prompt, enter ⌞`go.` ↵⌟ to initialize the NUPRL 5 refiner.

```
ML[(ORB)]> go.
```

Although it is not necessary to wait for the library process before starting the refiner it is important to enter the refiner's `go.` command *after* the library's `go.`. The library contains informations about tactics and rules that the refiner needs in order to operate properly. Initially there will be only little exchange between the library and the refiner. The refiner process will return quickly with another prompt `ML[(ORB)]>`.

The following emacs script defines an interactive function `nurefine` that performs the above steps in a new emacs shell. It pops up a `*NuRefine*` window immediately below the editor window and starts the NUPRL 5 refiner in it.

```
(defun nurefine ()
  (interactive)
  (nuprl-frame "NuRefine" 10 280 "nuref\n")
  (message "Starting Refiner ...")
  (set-foreground-color "Green4")
  (set-background-color "#ffffbb")
  (comint-send-string "NuRefine" "(top)\n")
  (comint-send-string "NuRefine" "go.\n")
)
```

It should be noted that the emacs functions `nulib`, `nuedit`, and `nurefine` can be invoked independently from each other. This may be helpful if one of the processes breaks and has to be completely restarted or if several refiners and/or editors shall be started.

3.3.3 Starting the Editor

To start the editor, enter the command ⌞`nuedd` ↵⌟ into a shell and then proceed as before.

```
SHELL-PROMPT> nuedd

USER(1): (top)

ML[(ORB)]> go.
```

Again, the library's `go.` command must precede that of the editor. The library contains a variety of explicit set up information that the editor needs to receive in order to determine how to display data, e.g. how to present the directory structure of the knowledge base, when and how to pop up windows, the location and meaning of editor buttons, etc.

Because of the amount of communication between the editor and the library it takes several minutes until the editor process is set up correctly. When it is ready, it will return with another prompt `ML[(ORB)]>`.
Enter ␣`win.` ↵␣ to have the editor pop up NUPRL 5 windows.

```
ML[(ORB)]> win.
```

The editor will establish a connection to the X-windows system and then pop up two windows: a *navigator* and a *top loop*, which will be explained in detail in Chapter 4. Afterwards it will return with another prompt `ML[(ORB)]>`.

The NUPRL *top loop* can be used for issuing commands to the library, refiner, and editor processes and provides support for editing NUPRL 5 terms (see Section 5.3). Users may safely iconify the emacs windows of the library, refiner, and editor processes at this point. A typical NUPRL 5 session will have many NUPRL windows open at the same time, so it is advisable to create some space for this, particularly when using medium-sized or larger fonts. The library, refiner, and editor process windows will only be needed for issuing low-level system commands and dealing with error situations that cause one of the NUPRL processes to break.

The following emacs script defines an interactive function `nuedit` that performs all the above steps in a new emacs shell. It pops up a `*NuEditor*` window immediately below the library window and starts the NUPRL 5 editor in it.

```
(defun nuedit ()
  (interactive)
  (nuprl-frame "NuEditor" 10 178 "nuedd\n")
  (message "Starting Editor ... please be patient")
  (set-foreground-color "midnightblue")
  (set-background-color "#ffd8ff")
  (comint-send-string "NuEditor" "(top)\n")
  (comint-send-string "NuEditor" "go.\n")
  (comint-send-string "NuEditor" "win.\n")
)
```

If you run the NUPRL 5 editor on a remote machine, make sure that its display is directed to your local machine *before* `nuedd` is entered. This is usually done by setting the environment variable `DISPLAY`. In a cshell you can do this with the following command

```
CSHELL-PROMPT> setenv DISPLAY LOCAL-HOST-NAME:0.0
```

In the emacs script you would have to insert the line

```
(comint-send-string bufname "setenv DISPLAY LOCAL-HOST-NAME:0.0\n'")
```

into the definition of `nuprl-frame`, immediately before `(comint-send-string bufname cmd)`.

Alternatively you may change the `eddhost` setting in you `.nuprl.config` file to redirect the NUPRL windows to your local machine's X server.

3.4 Exiting NUPRL 5

When you are ready to stop, first stop the editor and refiner process, and lastly the library. To shutdown gracefully, enter ␣`stop.` ↵␣ at the ML prompts of the three processes.

```
ML[(ORB)]> stop.
```

As a result, the editor and refiner will communicate to the library that they will disconnect now and then stop the respective ML and Lisp processes. The library process will cleanly shut down the knowledge base and then stop as well. Depending on the size of the knowledge base this may take between a few seconds and several minutes.

It is important that you explicitly terminate the three NUPRL processes rather than just quitting out of the editor NUPRL 5 is running under. In the latter case, the Lisp process can be left floating around in a hung state, hogging memory resources. This could also happen if your editor crashes or if you kill the shell (or emacs buffer) in which NUPRL 5 runs. You can use the Unix command `ps` to check for a hung Lisp process and the command `kill` to kill it.

The interactive emacs command `nuxit`, described below, is a safe way to terminate NUPRL 5.

```
(defun nuxit ()
  (interactive)
   (message "Shutting Down NuPRL 5 Library, Editor, and Refiner ...")
   (comint-send-string "NuEditor"  "stop.\n")
   (comint-send-string "NuRefine"  "stop.\n")
   (sleep-for 5)
)
```

Advanced emacs users may want to add to this script commands that kill the respective buffers and emacs frames after the library process has terminated.

Instead of shutting down gracefully you may also simply kill all three processes to stop NUPRL 5. In this case the library process will clean up the knowledge base when it is started the next time. This method for exiting NUPRL, however, is is not recommended.

3.5 Hints on Using the System

NUPRL's windows are at the "top-level" in the X environment. The windows can be managed (positioned, sized, etc.) in the same way as other top-level applications such as X-terminals. Creation and destruction of NUPRL windows, and manipulation of window contents, is done solely via commands interpreted by NUPRL.

NUPRL will receive mouse clicks and keyboard strokes whenever the input focus is on any of its windows. Any input event will make this window *active*, which is identified by the presence of NUPRL's *cursor*. This cursor appears either as a thin vertical bar between characters or as a highlighted (reverse video) region. The specific location of the cursor determines the semantics of keyboard strokes and mouse clicks, and is – like in most editors – independent of the current location of the mouse cursor.

The two main windows – the navigator window and the NUPRL 5 top loop – are intended to remain throughout the session. You may kill and reopen them at any time although it is not recommended to do so. You may create multiple clones of the navigator but not of the top loops. Chapter 4 describes the use of these windows as well as the kinds of objects that can be found in the library.

There are two other kinds of windows; *term editor* windows and *proof editor* windows. Both are used for editing objects in the library. The structure of NUPRL terms and the term editor is described in Chapter 5. The proof editor is described in Chapter 6.

If the system appears to be inexplicably stuck, check the Lisp windows; it is very possible that Lisp is garbage-collecting. This sometimes takes a few minutes.

Most Lisp versions allow computations to be interrupted. This is usually done by sending ⟨C-C⟩ to the Lisp process, or ⟨C-C⟩⟨C-C⟩ if Lisp is started up from an emacs sub-shell. (Sometimes Lisp catches the first two or three interrupt requests.) This will cause Lisp to enter its *debugger*, from which the computation can be resumed or aborted. Section 3.6 below describes how to use the Lisp debugger, and in particular, what to do if a NUPRL 5 process crashes. NUPRL is a continually-evolving experimental research system, and it is inevitable that it will contain bugs.

Aborting either of the three NUPRL 5 processes is always safe, since changes to objects, e.g. the effects of editor commands or inference steps, are immediately committed to the persistent library. When a NUPRL 5 process is restarted, the state should be exactly as it was before the process was killed.

Please report any behavior you think is due to a bug, or inconsistencies between the operation of the system and the documentation. Also report any break-points that you hit; they have either been left in the code accidentally, or they are there to help track down the source of bugs. We welcome suggestions for improvement. Send e-mail to nuprlbugs@cs.cornell.edu.

3.6 Troubleshooting

In this section we discuss problem situations that need to be resolved on the system level. Recovering from errors *within* either of the editors or the navigator is discusses in the respective chapters.

All system and error messages are directed to the emacs windows containing the three NUPRL 5 processes. It is recommended to check these windows if the system seems to be stuck or if other problems occur.

In most cases where the system appears to be stuck, one of the three Lisp processes is garbage-collecting. Depending on processor speed and available memory this may take a few minutes.

If the editor hangs for an unusually long time, one of the three main processes may have been thrown into the Lisp debugger. This may happen if a breakpoint was mistakenly left in the NUPRL code or if you hit a bug. You may also have accidentally interrupted Lisp. In either there will be an error message in the corresponding (Lisp) top loop. The particular debugger appearance and commands given below are for Allegro Common Lisp. Other Lisps should be similar.

The initial message put out by the debugger should tell you what caused it to be invoked. The following message, for instance appears after a keyboard interrupt

```
Error: Received signal number 2 (Keyboard interrupt)
  [condition type: INTERRUPT-SIGNAL]

Restart actions (select using :continue):
 0: continue computation

[changing package from "COMMON-LISP-USER" to "NUPRL5"]
[1c] NUPRL5(2):
```

To resume after an interrupt or breakpoint, enter ␣:cont ↵␣.

```
[1c] NUPRL5(2): :cont

ML[(edd)]>
```

If the ML prompt appears again, the process has successfully resumed. In some cases, Lisp cannot simply resume and will print another error message, as in the following case

```
Error: Non-structure argument NIL passed to structure-ref
[1] NUPRL5(3): :cont
Error: Can't continue and no restarts.
[2] NUPRL5(4):
```

In most of these cases, entering the expression ␣(fooe) ↵␣ will reset and restart the process.

```
[2] NUPRL5(4): (fooe)
ML[(edd)]>
```

In the worst case, kill the process by entering ␣:exit↵ and then restart it from scratch. The other processes will detect the link going dead and clean it up automatically.

If you kill a NUPRL window using window manager commands instead of the appropriate NUPRL editor commands, you will break the X connection and crash the editor. Future releases of NUPRL 5 will automatically repair the editor process but until then you have to recover at the lisp debug prompt using the following command sequence.

```
[2] NUPRL5(4): (ml-text "nuprl_oed_reset()")
[2] NUPRL5(5): (fooe)
ML[(edd)]> win.
```

If you kill the library process while it is shutting down the knowledge base, the knowledge base may end up in an unstable state and you may not be able to restart the system anymore. In this case you have to enter the directory containing the knowledge base (e.g. `~/nuprl/nuprl5/NuPrlDB`) and move the most recent subdirectories out of it, preserving them in some temporary directory.

The system will usually come up properly afterwards and return the knowledge base into a well-defined state but you will lose all the modifications recorded in the subdirectories that you moved out of the main directory, so it may be useful to move them out one at a time, until the NUPRL system starts again.

If you run into the same type of unrecoverable error twice, you may want to send a bug report to `nuprlbugs@cs.cornell.edu`. In this case type ␣:zoom↵ into the Lisp process before killing it and copy the output, together with the initial error messages into you bug report. Also, mention briefly what you were doing at the time of the crash. This will help the NUPRL programmers to identify the cause for the problem and fix it.

3.7 Customization

Experienced users will probably want to create their own initialization procedures for NUPRL. These could allow customizations such as:

- Changing key-bindings for the term and proof editors.
- Loading more / different tactics.

Currently, these initialization can only be run after starting up the pre-prepared disksaves for the NUPRL 5 library, editor, or refiner. You probably will want to put all your initialization commands into a Lisp file that is automatically loaded whenever a disksave is started up.

Note that NUPRL runs in the *nuprl* package. All symbols entered in Lisp will be interpreted relative to this package. The package inherits all the symbols of Common Lisp, but does *not* contain the various implementation-specific utilities found in the package *user* (or *common-lisp-user*). To refer to these other symbols, either change packages using `(in-package "USER")`, or explicitly qualify the symbols with a package prefix. If you change packages, you can change back to the NUPRL package using `(in-package "NUPRL")`.

The key bindings for the navigator and the term and proof editors can be altered by creating your own key macro files. The NUPRL 5 editor will look for a file `~/mykeys.macro` to determine any user-defined key bindings.