

Chapter 3

Running Nuprl 5

3.1 System Requirements

Nuprl 5 is written mostly in Common Lisp, but uses some extensions that require [Lucid](#) or [Allegro Lisp](#) and a Unix-based X window system. The Linux implementation of Allegro Common Lisp is freely available. All other versions of Allegro and Lucid require a license.

The Nuprl homepage provides an executable copy of the Allegro version of Nuprl 5 running under Linux. 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 combined require 36 MegaBytes of disk space. The contents of the standard library require an additional 120 MegaBytes of disk space and quickly grows to about 200–300 MegaBytes.

It is recommended to run Nuprl 5 on systems that have at least 128MB of RAM, 256MB of swap space, and 400MB 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 at the URL

<http://www.cs.cornell.edu/Info/Projects/NuPr1/nuprl5/index.html>

In the Nuprl 5 disksaves (i.e. the binary executable Lisp files) there are some dependencies on some shared `.so`-files. For the time being this requires these files to be at the same path as when the disksaves were made. This necessitates a directory called `/home/nuprl`.

To retrieve Nuprl 5, download the following files into the directory `/home/nuprl`.

[README](#)
[nuprl5-run.tgz](#)
[nuprl-fonts.tar.gz](#)

The file `nuprl5-run.tgz` contains both the executables and the library. It should be untarred on top of `/home/nuprl` with the following Unix commands.

```
cd /home/nuprl
tar -C /home -xzf nuprl5-run.tgz
```

The Nuprl 5 system can now be found within the subdirectory `nuprl5` with the executables residing in `nuprl5/bin`. This subdirectory also contains a compressed tar file with the contents of the standard Nuprl 5 library. It will not be installed automatically to prevent an existing library from being accidentally overwritten. To install the standard library, execute the following commands.

```
cd nuprl5
tar -xzf ./nuprl5.tgz
rm nuprl5.tgz
```

Note that this expands a 10MB `.tgz`-file into a 120MB directory called `NuprlDB`.

For compatibility reasons, it is advisable to have the Nuprl 5 binaries accessible from the directory `/home/nuprl/bin`. This is not strictly necessary but prevents that the executable path has to be overspecified. Create links to the Nuprl 5 library (`nulib`), editor (`nuedd`), and refiner (`nuref`) as follows.

```
cd /home/nuprl
mkdir bin
cd bin
ln -s ../nuprl5/bin/nulib
ln -s ../nuprl5/bin/nuedd
ln -s ../nuprl5/bin/nuref
```

To support the use of special mathematical symbols in formal theorems, the Nuprl system requires some special font files to be installed. The file `nuprl-fonts.tar.gz` contains the `bdf` format files for these fonts. It is recommended to install these fonts in the directory `/home/nuprl/fonts/bdf` as follows¹

```
cd /home/nuprl
gunzip nuprl-fonts.tar.gz
mkdir -p fonts/bdf
cd fonts/bdf
tar -xvf ../../nuprl-fonts.tar
rm ../../nuprl-fonts.tar
```

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 Sparcstations 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. Thus the fonts files may need to be placed in a public directory. Your system administrator should be able to advise you in this case.

¹This step is not necessary if Nuprl 4 is already installed. The special fonts for Nuprl 4 and Nuprl 5 are the same.

3.2.2 User-specific Configuration

To run Nuprl 5, a user must ensure that the directory for Nuprl binaries is included in the Unix load path and that the X server has the Nuprl fonts loaded. Furthermore, a Nuprl 5 configuration file must be set up. It is also recommended to set up a file with Nuprl 5 keybindings for customization purposes.

Add the directory `/home/nuprl/bin` to the user's load path. This can, for instance be done by adding the following line to the user's `.cshrc` file

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

The syntax is slightly different for other Unix shells.

To display Nuprl-specific symbols on your terminal, the X server that controls your display must have the Nuprl fonts loaded. 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. The following instructions assume that the user runs Unix and that the Nuprl fonts reside in some directory `FONT-DIR` (e.g. `/home/nuprl/fonts/bdf`) where the user's workstation can access them. Add the following lines to the start of the user's `.xinitrc` after any initial comments.

```
xset fp+ FONT-DIR
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.

The current Nuprl fonts in order of size from smallest to largest are named `nuprl-6x10`, `nuprl-8x13`, `nuprl-13` and `nuprl-20`, with `nuprl-13` being the standard font for Nuprl 5. To look at the special characters provided by these fonts, one may use the `xfd` command.

Upon startup Nuprl 5 expects to find a file `~/.nuprl.config` in the 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.

Create a file `.nuprl.config` in the user's home directory with the following entries:

```
(sockets SOCKET1 SOCKET2)
(libhost "HOSTNAME")
(dbpath "DATABASE-PATH")
(libenv "STANDARD-LIBRARY")
(foreground "FOREGROUND-COLOR")
(background "BACKGROUND-COLOR")
```

As `sockets` one may select any unused socket numbers if Nuprl 5 shall be run in single-user mode. If several users shall work with the same library, the socket numbers should be identical to the ones used by the library process.

The `libhost` should be the name of the host running the library. Currently this means that even a standalone machine needs a host name.

The `dbpath` and the `libenv` describe the location of the standard library. Usually, `DATABASE-PATH` should be chosen as `/home/nuprl/nuprl5/NuprlDB` and `STANDARD-LIBRARY`, the name of the basic library environment one will start with, should be `standard`.

The `foreground` and `background` colors set the colors for the Nuprl windows and can be chosen according to personal preferences.

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 bindings described in this manual correspond to the entries of the file `/home/nuprl/nuprl5/mykeys.macro`, which may have been modified after the creation of the editor disksave. To ensure that a user has the latest key bindings, copy this file into the user's home directory. This will also allow the user to customize the Nuprl 5 key bindings later.

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 refiner² 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 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`.

```
(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.

²**Is that true?** 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.

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 some 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 "nuprl-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 "#dddf")
  (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 Editor

Starting the editor is similar to starting the library. In a shell, enter the command `nuedd`.

```
SHELL-PROMPT> nuedd
```

At the Lisp USER prompt enter `(top)`.

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

At the ML prompt, enter `go.` to initialize the Nuprl 5 editor.

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

Although it is not necessary to wait for the library process before starting the editor it is important to enter the editor's `go.` command *after* the library's `go.` 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 pop up a few windows and return with another prompt `ML[(ORB)]>`.

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")
)
```

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)`.

3.3.3 Starting the Refiner

To start the refiner, enter the command `nuref` into a shell and then proceed as before.

```
SHELL-PROMPT> nuref
```

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

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

Again, the library's `go.` command must precede that of the refiner. 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.

3.3.4 The Initial Screen



Figure 3.1: Initial Nuprl 5 screen

In the standard configuration, the editor process will pop up three windows: a *navigator*, an *editor top loop*, a *refiner top loop*. A typical initial screen is shown in Figure 3.1. The snapshot was taken after starting Nuprl 5 from emacs and rearranging the Nuprl 5 windows on the screen.

The emacs windows on the upper right are the frames for the library, editor and refiner top loops. The Nuprl 5 editor top loop and the Nuprl 5 refiner top loop are shown in the two windows below. All top loops can be used for issuing commands to the respective processes. In contrast to the *text-oriented* emacs top loops, the Nuprl 5 top loops are *term-oriented* and incorporate the

Nuprl 5 term editor (see Section 5.4). They are better suited for editing object-level terms while editing text is more flexibly handled in the emacs top loops.

The emacs top loops will also receive all system output and error messages. It is recommended to keep them visible if there is sufficient space on the screen. Most users will seldomly use the Nuprl 5 top loops and may iconify them. 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.

```
(defun kill-frame-at (top left current-frames)
  (let ((actual-frame (car current-frames)))
    (if (and (equal top (cdr (cadr (caddr (caddr (cadr actual-frame))))))
            (equal left (cdr (car (caddr (caddr (caddr (cadr actual-frame))))))))
        (delete-frame (car actual-frame))
      ) )
  (if (cdr current-frames) (kill-frame-at top left (cdr current-frames)))
)

(defun kill-nuprl-frames ()
  (interactive)
  (let ((current-frames (cdr (current-frame-configuration))))
    (kill-frame-at 74 513 current-frames)
    (kill-frame-at 176 513 current-frames)
    (kill-frame-at 278 513 current-frames)
  ) )

(defun nuxit ()
  (interactive)
  (message "Shutting Down NuPRL 5 Library, Editor, and Refiner ...")
  (comint-send-string "NuEditor" "stop.\n")
  (comint-send-string "NuRefine" "stop.\n")
  (comint-send-string "NuLibrary" "stop.\n")
  (sleep-for 30)
  (comint-send-string "NuEditor" ":\exit\n")
  (comint-send-string "NuRefine" ":\exit\n")
  (comint-send-string "NuLibrary" ":\exit\n")
  (sleep-for 1)
  (comint-send-string "NuEditor" "exit\n")
  (comint-send-string "NuRefine" "exit\n")
  (comint-send-string "NuLibrary" "exit\n")
  (sleep-for 1)
  (kill-buffer "*NuLibrary*" )
  (kill-buffer "*NuEditor*" )
  (kill-buffer "*NuRefine*" )
  (kill-nuprl-frames)
)
```

Figure 3.2: emacs script for shutting down Nuprl 5

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 processes. The library process will cleanly shut down the knowledge base and then stop as well. This may take a while.

When you are now back to the Lisp mode in either of the processes, enter `(excl:exit)` to quit.

```
NUPRL5(5): (excl:exit)
```

It is important that you explicitly type `:exit` to terminate the Lisp process, 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 in Figure 3.2, is a safe way to terminate Nuprl 5. Note that the time for shutting down the library process cleanly depends on the size of the knowledge base and the machine on which the process runs. The 30 seconds are safe for a single-user Pentium II 366Mhz machine that runs Nuprl 5 under Linux. It is recommended that you go through the exit procedure by hand at least once, measure the time for the library to shut down, and modify the sleep time in above emacs script accordingly before you run it. You may also have to adjust the parameters for the frames to be killed (they are always 2 points less than the parameters for opening the frames).

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 three main windows – the navigator window and the Nuprl 5 top loops – are intended to remain throughout the session. You may kill and reopen them at any time. 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.

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.

Section 3.6 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.

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 discussed in the respective chapters.

The most common problem user may encounter is accidentally closing the navigator window or either of the two Nuprl 5 top loops. In this case one has to enter `win.` into the editor window. This will reopen all the windows that were present on the initial screen.

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 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 your 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.

Must be a bit more specific here

3.8 Utilities

Only a few fragments. Write more later after interviewing Lori and Rich

Patches to the current disksaves must be placed in the directory `/home/nuprl/nuprl5/bin/patches`.

If you edit the file `bin/patches/n5.0-LIB-restart-32-0` you will see `:(patches patch1 .. patchn)` then the patch will be loaded automatically when `nulib` is started.

Garbage Collection: Since Nuprl 5 preserves previous versions of modified library objects and even of removed ones, the size of the library grows rapidly. Occasionally it becomes necessary to remove objects which are not linked to (by a name or whatever). Nuprl 5 provides a garbage collector for this purpose.

In Nuprl 5, we will need to have a section describing how to garbage collect the database. There is a lot of flexibility here since the user can set various parameters that determine what gets collected when. I have some text on this already, but we will need to enhance it.

GC before shutdown ML> `library_close_gc 'standard';;`

`pui_refresh();;` (editor, ML)

what was that again??

loading macros