# XREPL: eXtended REPL

Version 7.1

Eli Barzilay <eli@barzilay.org>

October 26, 2018

```
(require xrepl)
```
package: `xrepl-lib`

XREPL extends the `racket` REPL significantly, turning it into a more useful tool for interactive exploration and development. Additions include "meta commands," using readline, keeping past evaluation results, and more.

XREPL is enabled by default in the Racket REPL when the `xrepl-lib` package is installed.

# 1 Meta REPL Commands

Most of the XREPL extensions are implemented as meta commands. These commands are entered at the REPL, prefixed by a `,` and followed by the command name. Note that several commands correspond directly to Racket functions (e.g., `,exit`) — but since they work outside of your REPL, they can be used even if the matching bindings are not available.

## 1.1 Generic Commands

`,help [<command-name>]`
*display available commands*
[Synonyms: `,h` `,?`]

Without an argument, displays a list of all known commands. Specify a command to get help specific to that command.

`,exit [<exit-code>]`
*exit racket*
[Synonyms: `,quit` `,ex`]

Exits Racket, optionally with an error code (see `exit`).

`,cd [<path>]`
*change the current directory*

Sets the `current-directory` to the given path. If no path is specified, use your home directory. Path arguments are passed through `expand-user-path` so you can use `~`. An argument of `-` means "the previous path".

`,pwd`
*display the current directory*

Reports the value of `current-directory`.

`,shell <shell-command>`
*run a shell command*
[Synonyms: `,sh` `,ls` `,cp` `,mv` `,rm` `,md` `,rd` `,git` `,svn`]

Use `,shell` (or `,sh`) to run a generic shell command (via `system`). For convenience, a few synonyms are provided — they run the specified executables (still using `system`).

When the REPL is in the context of a module with a known source file, the shell command can use the `F` environment variable as the path to the file. Otherwise, `F` is set to an empty string.

`,edit <file> ...`

*edit files in your $EDITOR*
[Synonyms: `,e`]

Runs an editor, as specified by your `EDITOR` environment variable, with the given file/s arguments. If no files are specified and the REPL is currently inside a module's namespace, then the file for that module is used. If the `EDITOR` environment variable is not set, use the `,drracket` command instead.

`,drracket [-flag] <file> ...`
*edit files in DrRacket*
[Synonyms: `,dr` `,drr`]

Runs DrRacket with the specified file/s. If no files are given, and the REPL is currently inside a module, the file for that module is used.

DrRacket is launched directly, without starting a new subprocess, and it is then kept running in a hidden window so further invocations are immediate. (When this command is used for the first time, you will see DrRacket start as usual, and then its window will disappear — that window is keeping DrRacket ready for quick editing.)

In addition to file arguments, arguments can specify one of a few flags for additional operations:

- `-new`: opens a new editing window. This is the default when no files are given and the REPL is not inside a module,
- `-open`: opens the specified file/s (or the current module's file). This is the default when files are given or when inside a module.
- `-quit`: exits the running DrRacket instance. Quitting DrRacket is usually not necessary. Therefore, if you try to quit it from the DrRacket window, it will instead just close the window but DrRacket will still be running in the background. Use this command in case there is some exceptional problem that requires actually quitting the IDE. (Once you do so, future uses of this command will start a fresh instance.)

## 1.2 Binding Information

`,apropos <search-for> ...`
*look for a binding*
[Synonyms: `,ap`]

Searches for known bindings in the current namespace. The arguments specify which binding to look for: use a symbol (without a `'`) to look for bindings that contain that name, and use a regexp (e.g., `#rx"..."`) to use a regexp for the search. Multiple arguments are and-ed together.

If no arguments are given, *all* bindings are listed.

`,describe [<phase-number>] <identifier-or-module> ...`
*describe a (bound) identifier*
[Synonyms: `,desc` `,id`]

For each of the specified names, describe where where it is coming from and how it was defined if it names a known binding. In addition, describe the module (list its imports and exports) that is named by arguments that are known module names.

By default, bindings are searched for at the runtime level (phase 0). You can add a different phase level for identifier lookups as a first argument. In this case, only a binding can be described, even if the same name is a known module.

`,doc <any> ...`
*browse the racket documentation*

Uses Racket's `help` to browse the documentation, look for a binding, etc. Note that this can be used even in languages that don't have the `help` binding.

## 1.3 Requiring and Loading Files

`,require <require-spec> ...+`
*require a module*
[Synonyms: `,req` `,r`]

Most arguments are passed to `require` as is. As a convenience, if a symbolic argument specifies an existing file name, then use its string form to specify the require, or use a `file` in case of an absolute path. In addition, an argument that names a known symbolic module name (e.g., one that was defined on the REPL, or a builtin module like `#%network`), then its quoted form is used. (Note that these shorthands do not work inside require subforms like `only-in`.)

`,require-reloadable <module> ...`
*require a module, make it reloadable*
[Synonyms: `,reqr` `,rr`]

Same as `,require`, but arranges to load the code in a way that makes it possible to reload it later, or if a module was already loaded (using this command) then reload it. Note that the arguments should be simple module names, without any require macros. If no arguments are given, use arguments from the last use of this command (if any).

Module reloading is enabled by turning off the `compile-enforce-module-constants` parameter — note that this prohibits some optimizations, since the compiler assumes that all bindings may change.

`,enter [<module>] [noisy?]`
*require a module and go into its namespace*
[Synonyms: `,en`]

Uses `enter!` to have the REPL go "inside" a given module's namespace. A module name can specify an existing file as with the `,require-reloadable` command. If no module is given, and the REPL is already in some module's namespace, then 'enter!' is used with that module, causing it to reload if needed. Using `#f` makes it go back to the toplevel namespace.

Note that this can be used even in languages that don't have the `enter!` binding. In addition, `enter!` is used in a way that does not make it require itself into the target namespace.

`,toplevel`
*go back to the toplevel*
[Synonyms: `,top`]

Makes the REPL go back to the toplevel namespace. Same as using the `,enter` command with a `#f` argument.

`,load <filename> ...`
*load a file*
[Synonyms: `,ld`]

Uses `load` to load the specified file(s).

## 1.4 Debugging

`,backtrace`
*see a backtrace of the last exception*
[Synonyms: `,bt`]

Whenever an error is displayed, XREPL will not show its context printout. Instead, use the `,backtrace` command to display the backtrace for the last error.

`,time [<count>] <expr> ...`
*time an expression*

Times execution of an expression (or expressions). This is similar to `"time"` but the information that is displayed is a bit easier to read.

In addition, you can provide an initial number to specify repeating the evaluation a number of times. In this case, each iteration is preceded by two garbage collections, and when the iteration is done its timing information and evaluation result(s) are displayed. When the requested number of repetitions is done, some extreme results are removed (top and bottom 2/7ths), and the remaining results are be averaged. Finally, the resulting value(s) are from the last run are returned (and can be accessed via the bindings for the last few results, see §2

"Past Evaluation Results").

`,trace <function> ...`
*trace a function*
[Synonyms: `,tr`]

Traces the named function (or functions), using `trace`.

`,untrace <function> ...`
*untrace a function*
[Synonyms: `,untr`]

Untraces the named function (or functions), using `untrace`.

`,errortrace [<flag>]`
*errortrace instrumentation control*
[Synonyms: `,errt` `,inst`]

`errortrace` is a useful Racket library which can provide a number of useful services like precise profiling, test coverage, and accurate error information. However, using it can be a little tricky. `,errortrace` and a few related commands fill this gap, making `errortrace` easier to use.

`,errortrace` controls global use of `errortrace`. With a flag argument of `+` errortrace instrumentation is turned on, with `-` it is turned off, and with no arguments it is toggled. In addition, a `?` flag displays instrumentation state.

Remember that `errortrace` instrumentation hooks into the Racket compiler, and applies only to source code that gets loaded from source and therefore compiled. Therefore, you should use it *before* loading the code that you want to instrument.

`,profile [<expr> | <flag> ...]`
*profiler control*
[Synonyms: `,prof`]

This command can perform profiling of code in one of two very different ways: either statistical profiling via the `profile` library, or using the exact profiler feature of `errortrace`.

When given a parenthesized expression, `,profile` will run it via the statistical profiler, as with the `profile` form, reporting results as usual. This profiler adds almost no overhead, and it requires no special setup. In particular, it does not require pre-compiling code in a special way. However, there are some imprecise elements to this profiling: the profiler samples stack snapshots periodically which can miss certain calls, and it is also sensitive to some compiler optimizations like inlining procedures and thereby not showing them in the displayed analysis. See *Profile: Statistical Profiler* for more information.

In the second mode of operation, `,profile` uses the precise `errortrace` profiler. This profiler produces precise results, but like other uses of the `errortrace`, it must be enabled

before loading the code that is to be profiled. It can add noticeable overhead (potentially affecting the reported runtimes), but the results are accurate in the sense that no procedure is skipped. (For additional details, see *Errortrace: Debugging and Profiling*.)

In this mode, the arguments are flags that control the profiler. A `+` flag turns the profiler on — and as usual with `errortrace` functionality, this applies to code that is compiled from now on. A `-` flag turns this instrumentation off, and without any flags it is toggled. Once the profiler is enabled, you can run some code and then use this command to report profiling results: use `*` to show profiling results by time, and `#` for the results by counts. Once you've seen the results, you can evaluate additional code to collect more profiling information, or you can reset the results with a `!` flag. You can also combine several flags to perform the associated operations, for example, `,prof *!-` will show the accumulated results, clear them, and turn profiler instrumentation off.

Note that using *any* of these flags turns errortrace instrumentation on, even `,prof -` (or no flags). Use the `,errortrace` command to turn off instrumentation completely.

> `,execution-counts <file> ...`
> *execution counts*

This command makes it easy to use the execution counts functionality of `errortrace`. Given a file name (or names), `,execution-counts` will enable errortrace instrumentation for coverage, require the file(s), display the results, disables coverage, and disables instrumentation (if it wasn't previously turned on). This is useful as an indication of how well the test coverage is for some file.

> `,coverage <file>`
> *coverage information via a sandbox*
> [Synonyms: `,cover`]

Runs a given file and displays coverage information for the run. This is somewhat similar to the `,execution-counts` command, but instead of using `errortrace` directly, it runs the file in a (trusted) sandbox, using the `racket/sandbox` library and its ability to provide coverage information.

## 1.5 Miscellaneous Commands

> `,switch-namespace [<name>] [? | - | ! [<init>]]`
> *switch to a different repl namespace*
> [Synonyms: `,switch`]

This powerful command controls the REPL's namespace. While `,enter` can be used to make the REPL go into the namespace of a specific module, the `,switch-namespace` command can switch between *toplevel namespaces*, allowing you to get multiple separate "workspaces".

Namespaces are given names that are symbols or integers, where `*` is the name for the first initial namespace, serving as the default one. These names are not bindings — they are only used to label the known namespaces.

The most basic usage for this command is to simply specify a new name. A namespace that corresponds to that name will be created and the REPL will switch to that namespace. The prompt will now indicate this namespace's name. The name is usually insignificant, except when it is a `require`-able module: in this case, the new namespace is initialized to use that module's bindings. For example, `,switch racket/base` creates a new namespace that is called `racket/base` and initializes it with `racket/base`. For all other names, the new namespace is initialized the same as the current one.

Additional `,switch` uses:

- `,switch !` — reset the current namespace, recreating it using the same initial library. Note that it is forbidden to reset the default initial namespace, the one named `*` — this namespace corresponds to the one that Racket was started with, and where XREPL was initialized. There is no technical reason for forbidding this, but doing so is not useful as no resources will actually be freed.
- `,switch ! <module>` — resets the current namespace with the explicitly given simple module spec.
- `,switch <name> !` — switch to a newly made namespace. If a namespace by that name already existed, it is rest.
- `,switch <name> ! <module>` — same, but reset to the given module instead of what it previously used.
- `,switch - <name>` — drop the specified namespace, making it possible to garbage-collect away any associated resources. You cannot drop the current namespace or the default one (`*`).
- `,switch ?` — list all known namespaces.

Do not confuse namespaces with sandboxes or custodians. The `,switch` command changes *only* the `current-namespace` — it does not install a new custodian or restricts evaluation in any way. Note that it is possible to pass around values from one namespace to another via past result reference; see §2 "Past Evaluation Results".

> `,syntax [<expr>] [<flag> ...]`
> *set syntax object to inspect, and control it*
> [Synonyms: `,stx` `,st`]

Manipulate syntaxes and inspect their expansion.

Useful operations revolve around a "currently set syntax". With no arguments, the currently set syntax is displayed; an argument of `^` sets the current syntax from the last input to the REPL; and an argument that holds any other s-expression will set it as the current syntax.

Syntax operations are specified via flags:

- `+` uses `expand-once` on the current syntax and prints the resulting syntax. In addition, the result becomes the new "current" syntax, so you can use this as a poor-man's syntax stepper. (Note that in some rare cases expansion via a sequence of `expand-once` might differ from the actual expansion.)
- `!` uses `expand` to completely expand the current syntax.
- `*` uses the macro debugger's textual output to show expansion steps for the current syntax, leaving macros from `racket/base` intact. Does not change the current syntax. Uses `expand/step-text`, see *Macro Debugger: Inspecting Macro Expansion* for details.
- `**` uses the macro debugger similarly to `*`, but expands `racket/base` macros too, showing the resulting full expansion process.

Several input flags and/or syntaxes can be specified in succession as arguments to `,syntax`. For example, `,stx (when 1 2) ** !`.

`,check-requires [<module>]`
*check the `require's of a module*
[Synonyms: `,ckreq`]

Uses `show-requires` to analyze the `requires` of the specified module, defaulting to the currently entered module if we're in one. See *Macro Debugger: Inspecting Macro Expansion* for details.

`,log <opt> ...`
*control log output*

Starts (or stops) logging events at a specific level. The level can be:

- a known level name (currently one of `fatal`, `error`, `warning`, `info`, `debug`),
- `#f` for no logging,
- `#t` for maximum logging,
- an integer level specification, with `0` for no logging and bigger ones for additional verbosity.

`,install!`
*install xrepl in your Racket init file*

Convenient utility command to install XREPL in your Racket initialization file. This is done carefully, you will be notified of potential issues, and asked to authorize changes.

Changed in version 6.7 of package `xrepl-lib`: XREPL is enabled by default in the Racket REPL, which makes installation unnecessary.

# 2 Past Evaluation Results

XREPL makes the last few interaction results available for evaluation via special toplevel variables: `^`, `^^`, ..., `^^^^^`. The first, `^`, refers to the last result, `^^` to the previous one and so on.

As with the usual REPL printouts, `#<void>` results are not kept. In case of multiple results, they are spliced in reverse, so `^` refers to the last result of the last evaluation. For example:

```
-> 1
1
-> (values 2 3)
2
3
-> (values 4)
4
-> (list ^ ^^ ^^^ ^^^^)
'(4 3 2 1)
```

The rationale for this is that `^` always refers to the last *printed* result, `^^` to the one before that, etc.

In addition to these names, XREPL also binds `$1`, `$2`, ..., `$5` to the same references, so you can choose the style that you like. All of these bindings are made available only if they are not already defined. This means that if you have code that uses these names, it will continue to work as usual (and it will shadow the saved value binding).

The bindings are identifier macros that expand to the literal saved values; so referring to a saved value that is missing (because not enough values were shown) raises a syntax error. In addition, the values are held in a weak reference, so they can disappear after a garbage-collection.

Note that this facility can be used to "transfer" values from one namespace to another—but beware of struct values that might come from a different instantiation of a module.

# 3 Hacking XREPL

```
(require xrepl/xrepl)
```
package: `xrepl-lib`

XREPL is mainly a convenience tool, and as such you might want to hack it to better suit your needs. Currently, there is little convenient way to customize and extend it, but this will be added in the future.

```
(toplevel-prefix) → string?
(toplevel-prefix prefix) → void?
  prefix : string?
 = "-"
```

Sets the prefix for when not in a module. When in a module (using `,enter`), this prefix is not displayed.

## 3.1 Unstable and potentially unsafe modifications

If you're interested in tweaking XREPL beyond the public `xrepl/xrepl` interface, the `,enter` command can be used as usual to go into its implementation. The commands in there are unstable and likely to change, but can still be modified for convenience. For example — change an XREPL parameter:

```
-> ,en xrepl/xrepl
xrepl/xrepl> ,e
xrepl/xrepl> (saved-values-patterns '(#\~))
xrepl/xrepl> ,top
-> 123
123
-> ~
123
```

or add a command:

```
-> ,en xrepl/xrepl
xrepl/xrepl> (defcommand eli "stuff" "eli says" ["Make eli say
stuff"]
                 (printf "Eli says: ~a\n" (getarg 'line)))
xrepl/xrepl> ,top
-> ,eli moo
Eli says: moo
```

While this is not intended as *the* way to extend and customize XREPL, it is a useful debugging tool should you want to do so.

If you have any useful tweaks and extensions, please mail the author or the Racket developer's mailing list.