

Quicksript, a scripting plugin for DrRacket

Version 7.7.0.901

July 30, 2020

Laurent Orseau <laurent.orseau@gmail.com>

1 Introduction

Quickscript's makes it easy to extend DrRacket with small Racket scripts to automate some actions in the editor, while avoiding the need to restart DrRacket.

Creating a new script is as easy as a click on *Scripts | New script...* Each script is automatically added as an item to the *Scripts* menu, without needing to restart DrRacket. A keyboard shortcut can be assigned to a script (via the menu item). By default, a script takes as input the currently selected text, and outputs the replacement text. There is also direct access to some elements of DrRacket GUI for advanced scripting, like DrRacket's frame and the definition or interaction editor.

2 Installation

Quickscript is installed automatically with DrRacket, so you don't need to do anything.

2.1 Quickscript Extra

You can use Quickscript on its own, but the Quickscript Extra package has a wide range of useful scripts as well as some example scripts intended for customisation by the user.

To install it, either look for `quickscript-extra` in the DrRacket menu *File|Package Manager*, or run the `raco` command:

```
raco pkg install quickscript-extra
```

To activate you will need to register the new list of scripts in quickscript either by evaluating in DrRacket:

```
(require quickscript-extra/register)
```

or on the command line with

```
$ racket -l quickscript-extra/register
```

Then click on *Scripts|Manage scripts|Compile scripts and reload*. (There is no need to restart DrRacket.)

3 Make your own script: First simple example

Click on the *Scripts|Manage scripts|New script...* menu item, and enter *Reverse* for the script name. This creates and opens the file *reverse.rkt* in the user's scripts directory. Also, a new item automatically appears in the *Scripts* menu.

In the *.rkt* file that just opened in DrRacket, modify the `define-script` definition to the following:

```
(define-script reverse-selection
  #:label "Reverse"
  (λ (selection)
    (list->string (reverse (string->list selection)))))
```

and save the file.

Then go to a new tab, type some text, select it, and click on *Scripts|Reverse*, and voilà!

Don't interchange
script: function
property: you will
shadow it in the
menu by making
script: Manage
scripts|Reload
scripts menu after
saving the file).

4 Into more details

Quickscript adds a *Scripts* menu to the main DrRacket window. This menu has several items, followed by the list of scripts.

The *New script* item asks for a script name and creates a corresponding .rkt file in the user's script directory, and opens it in DrRacket.

Each script is defined with `define-script`, which among other things adds an entry in DrRacket's Scripts menu. A single script file can contain several calls to `define-script`.

By default, the new script is reduced to its simplest form. However, scripts can be extended with several optional *properties* and *arguments*. When all of them are used, a script can look like this:

```
(define-script a-complete-script
  ; Properties:
  #:label "Full script"
  #:help-string "A complete script showing all properties and ar-
  guments"
  #:menu-path ("Submenu" "Subsubmenu")
  #:shortcut #\a
  #:shortcut-prefix (ctl shift)
  #:output-to selection
  #:persistent
  #:os-types (unix macosx windows)
  ; Procedure with its arguments:
  (λ (selection #:frame fr
              #:editor ed
              #:definitions defs
              #:interactions ints
              #:file f)
    "Hello world!"))
```

Note that the arguments of the properties are literals, not expressions, so they must *not* be quoted. Below we detail first the procedure and its arguments and then the script's properties.

```
(require quickscript)      package: quickscript
```

```
(define-script name
  property ...
  proc)
```

```

property = #:label label-string
          | #:help-string string
          | #:menu-path (label-string ...)
          | #:shortcut char | symbol | #f
          | #:shortcut-prefix (shortcut-prefix ...)
          | #:persistent? #t | #f
          | #:output-to output-to
          | #:os-types (os-type ...)

shortcut-prefix = alt | cmd | meta | ctl | shift | option

output-to = selection | new-tab | message-box | clipboard | #f

os-type = macosx | unix | windows

proc = (λ (selection-id
          [#:editor editor-id]
          [#:definitions definitions-id]
          [#:interactions interactions-id]
          [#:frame frame-id]
          [#:file file-id])
        body-expr ...
        return-expr)

```

See the following subsections for a complete description.

Observe again that the arguments of the properties are literals and not expressions. This is because the script file is read twice for different purposes. The first time, Quicksript reads the script file to extract the minimum information necessary to build the menu items in DrRacket. No Racket operation is performed at this stage so as to be as light and quick as possible. Then, when the corresponding menu item is clicked, Quicksript reads the script file a second time, this time to actually read and visit the Racket module and call the corresponding procedure. That is, the script modules are instantiated only on demand to reduce the loading time and memory footprint.

4.1 The script's procedure

When clicking on a script label in the Scripts menu in DrRacket, its corresponding procedure is called. The procedure takes at least the `selection` argument, which is the string that is currently selected in the current editor. The procedure must return either `#f` or a `string?`. If it returns `#f`, no change is applied to the current editor, but if it returns a string, then the current selection is replaced with the return value.

If some of the above keywords are specified in the procedure, the Script Plugin detects them

and passes the corresponding values, so the procedure can take various forms:

```
(λ (selection) ....)
(λ (selection #:frame fr) ....)
(λ (selection #:file f) ....)
(λ (selection #:editor ed #:file f) ....)
....
```

Here is the meaning of the keyword arguments:

- `#:file` : (or/c path? #f)

The path to the current file of the definition window, or #f if there is no such file (i.e., unsaved editor).

Example:

```
(define-script current-file-example
  #:label "Current file example"
  #:output-to message-box
  (λ (selection #:file f)
    (string-append "File: " (if f (path->string f) "no-file")
      "\nSelection: " selection)))
```

See also: [file-name-from-path](#), [filename-extension](#), [path->string](#), [split-path](#).

- `#:definitions` : text%

The text% editor of the current definition window. See text% for more details.

- `#:interactions` : text%

The text% editor of the current interaction window. Similar to `#:definitions`.

- `#:editor` : text%

The text% current editor, either the definition or the interaction editor. Similar to `#:definitions`.

- `#:frame` : drracket:unit:frame<%>

DrRacket's frame. For advanced scripting.

Example:

```
(require racket/class)
(define-script number-tabs
  #:label "Number of tabs"
  #:output-to message-box
  (λ (selection #:frame fr)
    (format "Number of tabs in DrRacket: ~a"
      (send fr get-tab-count))))
```

4.2 The script's properties

The properties are mere data and cannot contain expressions.

Most properties (`#:label`, `#:shortcut`, `#:shortcut-prefix`, `#:help-string`) are the same as for the `menu-item%` constructor. In particular, a keyboard shortcut can be assigned to an item.

If a property does not appear in the dictionary, it takes its default value.

There are some additional properties:

- `#:menu-path` : `(listof string?) = ()` This is the list of submenus in which the script's label will be placed, under the Script menu.

Note that different scripts in different files can share the same submenus.

- `#:output-to` : `(or/c selection new-tab message-box clipboard #f) = selection`

If `selection`, the output of the procedure replaces the selection in the current editor (definitions or interactions), or insert the output at the cursor if there is no selection. If `new-tab`, the return value is written in a new tab. If `message-box`, the return value (if a string) is displayed in a `message-box`. If `clipboard`, the return value (if a string) is copied to the clipboard. If `#f`, the return value is not used.

If this value is changed, make sure to reload the menu with *Scripts | Manage scripts | Reload menu*.

- `#:persistent`

If the keyword `#:persistent` is *not* provided, each invocation of the script is done in a fresh namespace.

But if `#:persistent` is provided, a fresh namespace is created only the first time it is invoked, and the same namespace is re-used for the subsequent invocations. Note that a single namespace is kept per file, so if different scripts in the same file are marked as persistent, they will all share the same namespace (and, thus, variables). Also note that a script marked as non-persistent will not share the same namespace as the other scripts of the same file marked as persistent.

Consider the following script:

```
(define count 0)

(define-script persistent-counter
  #:label "Persistent counter"
  #:persistent
  #:output-to message-box
  (λ (selection)
```



```
(set! count (+ count 1))  
(number->string count)))
```

If the script is persistent, the counter increases at each invocation of the script via the menu, whereas it always displays 1 if the script is not persistent.

Note: Persistent scripts can be "unloaded" by clicking on the *Scripts|Manage scripts|Unload persistent scripts* menu item. In the previous example, this will reset the counter. Make sure to unload a persistent script after editing it.

Technical point: The script's procedure is called *outside* of the namespace that was used to `dynamic-require` it, and inside DrRacket frame's namespace so as to have access to objects in this frame.

- `#:os-types (listof (one-of/c unix macosx windows))`

This keyword must be followed by a list of supported os-types. Defaults to all types, i.e. `(unix macosx windows)`.

If changes are made to these properties, the Scripts menu will probably need to be reloaded by clicking on *Scripts|Manage scripts|Reload scripts menu*.

5 Script library

When the user creates a new script, the latter is placed into a sub-directory of (`find-system-path 'pref-dir`). A direct access to this folder is provided via the *Scripts|Manage scripts|Open script...* menu entry.

Additional directories to look for scripts can be added via the *Scripts|Manage scripts|Library* menu entry. When a directory is added to the library, all its `.rkt` files (non-recursively) are considered as scripts. Specific files can be excluded from the library.

6 Shadow scripts

When a script is installed from a third party package (like quickscript-extra), it comes with its set of own values for its properties. These values may not suit the user who may want to redefine some of them, like the menu path or the keyboard shortcuts. An obvious choice for the user is to copy/paste the entire script, but this would prevent from benefiting from further bug fixes and enhancements made by the writer of the original script.

To solve this problem, the user can instead make a *shadow script*, which creates a new script in the user's directory, with its own set of properties that can be changed by the user, but the procedure of this script is bound to that of the original script.

To make a shadow script, open the script library in *Scripts|Manage scripts|Library*, navigate to the third-party script and click on *Shadow*.

7 Updating the quickscript package

To update Quickscript once already installed, either do so through the *File|Package Manager* menu in DrRacket, or run `raco pkg update quickscript`.

The user's scripts will not be modified in the process.

8 Distributing your own scripts

The *simplest* way to distribute a small script `s` is to publish it as a gist or on PasteRack, and share the link. A user can then copy/paste the contents into a new script. Don't forget to include a permissive license such as MIT/Apache 2.

The *best* way to distribute scripts is by creating a package—the user only has to install the package. Assuming your scripts are stored in the "`scripts`" subdirectory, include a file (say "`register.rkt`") at the root directory of the package containing the following code:

```
#lang racket/base
(require (for-syntax racket/base
                   racket/runtime-path
                   (only-in quickscript/library
                           add-third-party-script-directory!)))
```

```
;; This file is going to be called during setup and will automatically
;; register the scripts subdirectory in quickscript's library.
```

```
(begin-for-syntax
 (define-runtime-path script-dir "scripts")
 (add-third-party-script-directory! script-dir))
```

If the file "`register.rkt`" is not at the root, the `runtime-path` needs to be modified accordingly.

You can see an example with `quickscript-extra`.

Don't forget to register your package on the Racket server.

9 License

MIT License

Copyright (c) 2012-2018 by Laurent Orseau <laurent.orseau@gmail.com>.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.