Geiser User Manual

Emacs and Scheme talk to each other

Jose Antonio Ortega Ruiz

This manual documents Geiser, an Emacs environment to hack in Scheme.

The document was typeset with GNU Texinfo.

Table of Contents

1 Introduction

Geiser is an Emacs environment to hack and have fun in Scheme. If that's enough for you, see Chapter 2 [Installation], page 3 to get it running and Chapter 3 [The REPL], page 5 for the fun part.

1.1 Modus operandi

As already mentioned, Geiser relies on a running Scheme process to obtain the information it makes accessible to the programmer. There's little effort, on the Elisp side, to understand, say, the module system used by the Scheme implementation at hand; instead, a generic interface between the two worlds is defined, and each supported Scheme includes a library implementing that API, together with some wee shims in Elisp allowing the reuse of the Emacs-side framework, which constitutes the bulk of the code.

While being as generic as possible, the Scheme-Elisp interface makes some assumptions about the capabilities and interaction mode of the corresponding REPL. In particular, Geiser expects the latter to support namespaces in the form of a module system, and to provide a well defined way to establish the REPL's current namespace (or module), as well as the current's file module (or namespace). Thus, all evaluations performed by Geiser either in the REPL or in a source code buffer happen in the context of the current namespace. Every time you switch to a different file, you're switching namespaces automatically; at the REPL, you must request the switch explicitly (usually just using means provided by the Scheme implementation itself).

If your favourite scheme supports the above modus operandi, it has all that's needed for a bare-bones Geiser mode. But Geiser can, and will, use any metadata available: procedure arities and argument lists to display interactive help, documentation strings, location information to jump to definitions, export lists to provide completion, and so on and so forth. Although this is not an all-or-none proposition (Geiser can operate with just part of that functionality available), i've concentrated initially in supporting those Schemes with the richest (to my knowledge) introspection capabilities, namely, Guile and Racket.

1.2 Showing off

When working with a fully conniving Scheme, Geiser can offer the following functionality:

- Form evaluation in the context of the current file's module.
- Macro expansion.
- File/module loading and/or compilation.
- Namespace-aware identifier completion (including local bindings, names visible in the current module, and module names).
- Autodoc: the echo area shows information about the signature of the procedure/macro around point automatically.
- Jump to definition of identifier at point.
- Access to documentation (including docstrings when the implementation provides it).
- Listings of identifiers exported by a given module.
- Listings of callers/callees of procedures.

- Rudimentary support for debugging (when the REPL provides a debugging) and error navigation.
- Support for multiple, simultaneous REPLs.

In the following pages, i'll try to explain what these features actually are (i'm just swanking here), and how to use them for your profit. But, before that, let's see how to install Geiser.

2 Installation

2.1 Must needs

If Geiser came with any guarantees, you'd break all of them by not using GNU Emacs 23.2 (or better: i regularly use it with a recent Emacs snapshot) and at least one of the supported schemes, namely:

- Racket 5.0.1 or better
- Guile 1.9.14, directly compiled from a recent checkout of its Git *master* branch.

Since Geiser supports multiple REPLs, having both of them will just add to the fun.

Downloading Geiser

You'll also need Geiser itself. The latest release tarball can be found here, while older versions are here. Just download geiser-0.1.tar.gz and untar it in a directory of your choice.

If you feel like living on the bleeding edge, just grab Geiser from its Git repository over at Savannah, either with the following incantation:

```
git clone git://git.sv.gnu.org/geiser.git
```

or, if you happen to live under a firewall, with the alternative:

```
git clone http://git.sv.gnu.org/r/geiser.git
```

You can also follow Geiser's development in one or three mirrors that are kept synchronized with the one at Savannah.

Either way, you'll now be in possession of a copy of Geiser's libre code. I'll follow you into its directory and the next section.

2.2 Setting it up

Geiser is ready to be used out of the box without much more ado. For the sake of concreteness, let's assume you put its source in the directory '`~/lisp/geiser`'. All you need to do is to add the following line to your Emacs initialisation file (be it '`~/.emacs`' or any of its moral equivalents):

```
(load-file "~/lisp/geiser/elisp/geiser.el")
```

or simply evaluate that form inside Emacs (you wouldn't kill a friend just to start using Geiser, would you?). That's it: you're ready to [go], page 5. If you obtained the Geiser source tree from a release tarball, you can even continue to read this fine manual inside Emacs by opening '`doc/geiser.info`' using `C-u C-h i`.

What? You still here? I promise the above is all that's needed to start using Geiser. But, in case you are missing your `configure/make all install` routine, by all means, you can go through those motions to byte compile and install Geiser too. That is, you enter the source directory and (since we grabbed the development tree) run the customary *autogen* script:

```
$ cd ~/lisp/geiser
$ ./autogen.sh
```

I recommend that you compile Geiser in a separate directory:

```
$ mkdir build && cd build
$ ../configure
<some drivel here>
$ make all
<more of the above>
```

Now you have two options: loading the byte-compiled Geiser from the '`elisp`' subdirectory, or installing it system-wide. To load the bytecode from here, add this line to your initialisation file:

```
(load "~/lisp/geiser/build/elisp/geiser-load")
```

and eval that form or (gasp!) restart Emacs and you're done. Yes, that's `load` and '`geiser-load`' instead of `load-file` and '`geiser.el`'.

If you prefer a system-wide installation, just type:

```
$ sudo make install
```

With the above spell, Geiser will be compiled and installed in a safe place inside Emacs load path. To load it into Emacs you'll need, *instead* of the `load-file` form above, the following line in your initialisation file:

```
(require 'geiser-install)
```

Please note that we're requiring `geiser-install`, and *not* `geiser`, and that there's no `load-file` to be seen this time. There are some ways of fine-tuning this process, mainly by providing additional arguments in the call to `configure`: you'll find those gory details in the file called '`INSTALL`', right at the root of the source tree. The installation will also take care of placing this manual, in Info format, where Emacs can find it, so you can continue to learn about Geiser inside its natural habitat. See you there and into the next chapter!

2.3 Friends

Although Geiser does not need them, it plays well with (and is enhanced by) the following Emacs packages:

- Paredit. Regardless of whether you use Geiser or not, you shouldn't be coding in any Lisp dialect without the aid of Taylor Campbell's structured editing mode.
- Company. Nikolaj Schumacher's `company-mode` provides a generic front-end for completion engines (such as Geiser's). Very nice if you like that kind of thing: judge by yourself with the help of this screencast.
- Quack. You can still use the many goodies provided by Neil van Dyke's `quack-mode`, since most of them are not (yet) available in Geiser. The only caveat might be a conflict between Quack's and Geiser's default key bindings, which i'm sure you'll manage to tackle just fine. It's also probably a good idea to require `quack` *after* loading '`geiser.el`' (or requiring a compiled version).

You just need to install and setup them as usual, for every package's definition of usual. Geiser will notice their presence and react accordingly.

3 The REPL

If you've followed the indications in Section 2.2 [Setting it up], page 3, your Emacs is now ready to start playing. Otherwise, i'll wait for you: when you're ready, just come back here and proceed to the following sections.

3.1 Starting the REPL

To start a Scheme REPL (meaning, a scheme process offering you a Read-Eval-Print Loop), Geiser provides the generic interactive command `run-geiser`. If you run it (via, as is customary in Emacs, *M-x run-geiser*, you'll be saluted by a prompt asking which one of the supported implementations you want to launch (yes, you can stop the asking: see [below], page 8). Tabbing for completion will offer you, as of this writing, `guile` and `racket`. Just choose your poison, and a new REPL buffer will pop-up.

If all went according to plan, you'll be facing an implementation-dependent banner, followed by an interactive prompt. Going according to plan includes having the executable of the Scheme you chose in your path. If that's not the case, you can tell Emacs where it is, as described [below], page 8. Returning to our REPL, the first thing to notice is that the funny prompt is telling you your current module: its name is the part just after the `@` sign (in Guile, that means `guile-user`, while Racket's top namespace doesn't have a name; cf. Section 3.3 [Switching context], page 6 below). Other than that, this is pretty much equivalent to having a command-line interpreter in a terminal, with a bunch of add-ons that we'll be reviewing below. You can start typing sexps right there: Geiser will only dispatch them for evaluation when they're complete, and will indent new lines properly until then. It will also keep track of your input, maintaining a history file that will be reloaded whenever you restart the REPL.

Connecting to an external Scheme

There's an alternative way of starting a Geiser REPL: you can connect to an external Scheme process, provided it's running a REPL server at some known port. How to make that happen depends on the Scheme implementation.

If you use Guile, you just need to start your Guile process (possibly outside Emacs) passing to it the flag `--listen`. This flag accepts an optional port as argument (as in `--listen=1969`), if you don't want to use the default.

In Racket, you have to use the REPL server that comes with Geiser. To that end, put Geiser's Racket scheme directory in the Racket's collection search path and invoke `start-geiser` (a procedure in the module `geiser/server`) somewhere in your program, passing it the desired port. This procedure will start the REPL server in a separate thread. For an example of how to do that, see the script '`bin/geiser-racket.sh`' in the source distribution, or, if you've compiled Geiser, '`bin/geiser-racket-noinst`' in the build directory, or, if you've installed Geiser, '`geiser-racket`' in '`<installation-prefix>/bin`'. These scripts start a new interactive Racket that is also running a REPL server (they also load the errortrace library to provide better diagnostics, but that's not strictly needed).

With your external Scheme process running and serving, come back to Emacs and execute *M-x geiser-connect*, *M-x connect-to-guile* or *M-x connect-to-racket*. You'll be asked for a host and a port, and, voila, you'll have a Geiser REPL that is served by the

remote Scheme process in a dedicated thread, meaning that your external program can go on doing whatever it was doing while you tinker with it from Emacs. Note, however, that all Scheme threads share the heap, so that you'll be able to interact with those other threads in the running scheme from Emacs in a variety of ways. For starters, all your (re)defintions will be visible everywhere. That's dangerous, but will come in handy when you need to debug your running webserver.

The connection between Emacs and the Scheme process goes over TCP, so it can be as remote as you need, perhaps with the intervention of an SSH tunnel.

3.2 First aids

A quick way of seeing what else Geiser's REPL can do for you, is to display the corresponding entry up there in your menu bar. No, i don't normally use menus either; but they can come in handy until you've memorized Geiser's commands, as a learning device. And yes, i usually run Emacs inside a terminal, but one can always use La Carte to access the menus in a convenient enough fashion.

Or just press `C-h m` and be done with that.

Among the commands at your disposal, we find the familiar input navigation keys, with a couple twists. By default, `M-p` and `M-n` are bound to *matching* items in your input history. That is, they'll find the previous or next sexp that starts with the current input prefix (defined as the text between the end of the prompt and your current position, a.k.a. *point*, in the buffer). For going up and down the list unconditionally, just use `C-c M-p` and `C-c M-n`. In addition, navigation is sexp- rather than line-based.

There are also a few commands to twiddle with the Scheme process. `C-c C-q` will gently ask it to quit, while `C-u C-c C-q` will mercilessly kill the process (but not before stowing your history in the file system). Unless you're using a remote REPL, that is, in which case both commands will just sever the connection and leave the remote process alone. If worse comes to worst and the process is dead, `C-c C-z` will restart it (but the same shortcut, issued when the REPL is alive, will bring you back to the buffer you came from, as explained [here], page 12).

The remaining commands are meatier, and deserve sections of their own.

3.3 Switching context

In tune with Geiser's [modus operandi], page 1, evaluations in the REPL take place in the namespace of the current module. As noted above, the REPL's prompt tells you the name of the current module. To switch to a different one, you can use the command `switch-to-geiser-module`, bound to `C-c C-m`. You'll notice that Geiser simply uses a couple of meta-commands provided by the Scheme REPL (the stock `,m` in Guile and `,enter` in Racket), and that it doesn't even try to hide that fact. That means that you can freely use said native ways directly at the REPL, and Geiser will be happy to oblige.

Once you enter a new module, only those bindings visible in its namespace will be available to your evaluations. All schemes supported by Geiser provide a way to import new modules in the current namespace. Again, there's a Geiser command, `geiser-repl-import-module`, to invoke such functionality, bound this time to `C-c C-i`. And, again, you'll see Geiser just introducing the native incantation for you, and you're free to use such incantations by hand whenever you want.

One convenience provided by these two Geiser commands is that completion is available when introducing the new module name, using the `TAB` key. Pressing it at the command's prompt will offer you a prefix-aware list of available module names.

Which brings me to the next group of REPL commands.

3.4 Completion and error handling

We've already seen Geiser completion of module names in action at the mini-buffer. You won't be surprised to know that it's also available at the REPL buffer itself. There, you can use either *C-.* or *M-‘* to complete module names, and `TAB` or *M-TAB* to complete identifiers. Geiser will know what identifiers are bound in the current module and show you a list of those starting with the prefix at point. Needless to say, this is not a static list, and it will grow as you define or import new bindings in the namespace at hand. If no completion is found, `TAB` will try to complete the prefix after point as a module name.

REPL buffers use Emacs' compilation mode to highlight errors reported by the Scheme interpreter, and you can use the `next-error` command (*M-g n*) to jump to their location. By default, every time you enter a new expression for evaluation old error messages are forgotten, so that *M-g n* will always jump to errors related to the last evaluation request, if any. If you prefer a not so forgetful REPL, set the customization variable `geiser-repl-forget-old-errors-p` to `nil`. Note, however, that even when that variable is left as *t*, you can always jump to an old error by moving to its line at the REPL and pressing *RET*. When your cursor is away from the last prompt, *TAB* will move to the next error in the buffer, and you can use *BACKTAB* everywhere to go to the previous one.

3.5 Autodoc and friends

Oftentimes, there's more you'll want to know about an identifier besides its name: what module does it belong to? is it a procedure and, if so, what arguments does it take? Geiser tries to help you answering those questions too.

Actually, if you've been playing with the REPL as you read, you might have notice some frantic activity taking place in the minibuffer every now and then. That was Geiser trying to be helpful (while, hopefully, not being clippy), or, more concretely, what i call, for want of a better name, its *autodoc* mode. Whenever it's active (did you notice that *A* in the mode-line?), Geiser's gerbils will be scanning what you type and showing (unless you silent them with *C-c C-a*) information about the identifier nearest to point.

If that identifier corresponds to a variable visible in the current namespace, you'll see the module it belongs to and its value. For procedures and macros, autodoc will display, instead of their value, the argument names (or an underscore if Geiser cannot determine the name used in the definition). Optional arguments are surrounded by parenthesis. When the optional argument has a default value, it's represented by a list made up of its name and that value. When the argument is a keyword argument, its name has "#:" as a prefix.

If that's not enough documentation for you, *C-c C-d d* will open a separate documentation buffer with help on the symbol at point. This buffer will contain implementation-specific information about the identifier (e.g., its docstring for Guile, or its contract, if any, for Racket), and a handy button to open the corresponding manual entry for the symbol, which will open an HTML page (for Racket) or the texinfo manual (for Guile).

Geiser can also produce for you a list, classified by kind, of the identifiers exported by a given module: all you need to do is press *C-c C-d m*, and type or complete the desired module's name.

The list of exported bindings is shown, again, in a buffer belonging to Geiser's documentation browser, where you have at your disposal a bunch of navigation commands listed in See Section 5.3 [our cheat-sheet], page 18. We'll have a bit more to say about the documentation browser in See [a later section], page 13.

If that's still not enough, Geiser can jump, via *M-.*, to the symbol's definition. A buffer with the corresponding file will pop up, with its point resting upon the identifier's defining form. When you're done inspecting, *M-,* will bring you back to where you were. As we will see, these commands are also available in scheme buffers. *M-.* also works for modules: if your point is on an unambiguous module name, the file where it's defined will be opened for you.

3.6 Customization and tips

The looks and ways of the REPL can be fine-tuned via a bunch of customization variables. You can see and modify them all in the corresponding customization group (by using the menu entry or the good old *M-x customize-group geiser-repl*), or by setting them in your Emacs initialization files (as a rule, all knobs in Geiser are turnable this way: you don't need to use customization buffers if you don't like them).

I'm documenting below a proper subset of those settings, together with some related tips.

Choosing a Scheme implementation

Instead of using the generic `run-geiser` command, you can start directly your Scheme of choice via `run-racket` or `run-guile`. In addition, the variable `geiser-active-implementations` contains a list of those Schemes Geiser should be aware of. Thus, if you happen to be, say, a racketeer not to be beguiled by other schemes, you can tell Geiser to forget about the richness of the Scheme ecosystem with something like

```
(setq geiser-active-implementations '(racket))
```

in your initialisation files.

When starting a new REPL, Geiser assumes, by default, that the corresponding Scheme binary is in your path. If that's not the case, the variables to tweak are `geiser-guile-binary` and `geiser-racket-binary`, which should be set to a string with the full path to the requisite binary.

You can also specify a couple more initialisation parameters. For Guile, `geiser-guile-load-path` is a list of paths to add to its load path when it's started, while `geiser-guile-init-file` is the path to an initialisation file to be loaded on startup. The equivalent variables for Racket are `geiser-racket-collects` and `geiser-racket-init-file`.

History

By default, Geiser won't record duplicates in your input history. If you prefer it did, just set `geiser-repl-history-no-dups-p` to `nil`. History entries are persistent across REPL sessions: they're saved in implementation-specific files whose location is controlled by the

variable `geiser-repl-history-filename`. For example, my Geiser configuration includes the following line:

```
(setq geiser-repl-history-filename "~/.emacs.d/geiser-history")
```

which makes the files '`geiser-history.guile`' and '`geiser-history.racket`' to live inside my home's '`.emacs.d`' directory.

Autodoc

If you happen to love peace and quiet and prefer to keep your REPL's echo area free from autodoc's noise, `geiser-repl-autodoc-p` is the customization variable for you: set it to `nil` and autodoc will be disabled by default in new REPLs. You can always bring the fairies back, on a per REPL basis, using *C*-`c` *C*-`a`.

Remote connections

When using `connect-to-guile` or `geiser-connect`, you'll be prompted for a host and a port, defaulting to localhost and 37146. You can change those defaults customizing `geiser-repl-default-host` and `geiser-repl-default-port`, respectfully.

4 Between the parens

A good REPL is a must, but just about half the story of a good Scheme hacking environment. Well, perhaps a bit more than a half; but, at any rate, one surely needs also a pleasant way of editing source code. Don't pay attention to naysayers: Emacs comes with an excellent editor included for about any language on Earth, and just the best one when that language is sexpy (specially if you use [Paredit], page 4). Geiser's support for writing Scheme code adds to Emacs' `scheme-mode`, rather than supplanting it; and it does so by means of a minor mode (unimaginatively dubbed `geiser-mode`) that defines a bunch of new commands to try and, with the help of the same Scheme process giving you the REPL, make those Scheme buffers come to life.

4.1 Activating Geiser

With Geiser installed following any of the procedures described in Section 2.2 [Setting it up], page 3, Emacs will automatically activate *geiser-mode* when opening a Scheme buffer. Geiser also instructs Emacs to consider files with the extension '`rkt`' part of the family, so that, in principle, there's nothing you need to do to ensure that Geiser's extensions will be available, out of the box, when you start editing Scheme code.

Indications that everything is working according to plan include the 'Geiser' minor mode indicator in your mode-line and the appearance of a new entry for Geiser in the menu bar. If, moreover, the mode-line indicator is the name of a Scheme implementation, you're indeed in a perfect world; otherwise, don't despair and keep on reading: i'll tell you how to fix that in a moment.

The menu provides a good synopsis of everthing Geiser brings to the party, including those keyboard shortcuts we Emacsers love. If you're seeing the name of your favourite Scheme implementation in the mode-line, have a running REPL and are comfortable with Emacs, you can stop reading now and, instead, discover Geiser's joys by yourself. I've tried to make Geiser as self-documenting as any self-respecting Emacs package should be. If you follow this route, make sure to take a look at Geiser's customization buffers (*`M-x customize-group RET geiser`*): there's lot of fine tunning available there. You might also want to take a glance at the Chapter 5 [Cheat sheet], page 17.

Since *geiser-mode* is a minor mode, you can toggle it with *`M-x geiser-mode`*, and control its activation in hooks with the functions `turn-on-geiser-mode` and `turn-off-geiser-mode`. If, for some reason i cannot fathom, you prefer *geiser-mode* not to be active by default, customizing `geiser-mode-auto-p` to `nil` will do the trick.

And if you happen to use a funky extension for your Scheme files that is not recognised as such by Emacs, just tell her about it with:

```
(add-to-list 'auto-mode-alist '("\\.funky-extension\\'" . scheme-mode))
```

Now, *geiser-mode* is just a useless wretch unless there's a running Scheme process backing it up. Meaning that virtually all the commands it provides require a REPL up and running, preferably corresponding to the correct Scheme implementation. In the following section, we'll see how to make sure that that's actually the case.

4.2 The source and the REPL

As i've already mentioned a couple of times, *geiser-mode* needs a running REPL to be operative. Thus, a common usage pattern will be for you to first call `run-geiser` (or one of its variants, see them described [here], page 8), and then open Scheme files; but there's nothing wrong in first opening a couple Scheme buffers and then starting the REPL (you can even find it more convenient, since pressing `C-c C-z` in a Scheme buffer will start the REPL for you). Since Geiser supports more than one Scheme implementation, though, there's the problem of knowing which of them is to be associated with each Scheme source file. Serviceable as it is, *geiser-mode* will try to guess the correct implementation for you, according to the algorithm described below. If you find that Geiser is already guessing right the Scheme implementation, feel free to skip to the [next subsection], page 12.

How Geiser associates a REPL to your Scheme buffer

To determine what Scheme implementation corresponds to a given source file, Geiser uses the following algorithm:

1. If the file-local variable `geiser-scheme-implementation` is defined, its value is used. A common way of setting buffer-local variables is to put them in a comment near the beginning of the file, surrounded by `-*-` marks, as in:

   ```
   ;; -*- geiser-scheme-implementation: guile -*-
   ```

2. If you've customized `geiser-active-implementations` so that it's a single-element list (as explained [here], page 8), that element is used as the chosen implementation.
3. The contents of the file is scanned for hints on its associated implementation. For instance, files that contain a `#lang` directive will be considered Racket source code, while those with a `define-module` form in them will be assigned to a Guile REPL.
4. The current buffer's file name is checked against the rules given in `geiser-implementations-alist`, and the first match is applied. You can provide your own rules by customizing this variable, as explained below.
5. If we haven't been lucky this far and you have customized `geiser-default-implementation` to the name of a supported implementation, we'll follow your lead.
6. See? That's the problem of being a smart alec: one's always outsmarted by people around. At this point, *geiser-mode* will humbly give up and ask you to explicitly choose the Scheme implementation.

As you can see in the list above, there are several ways to influence Geiser's guessing by mean customizable variables. The most direct (and most impoverishing) is probably limiting the active implementations to a single one, while customizing `geiser-implementations-alist` is the most flexible (and, unsurprisingly, also the most complex). Here's the default value for the latter variable:

```
(((regexp "\\.scm$") guile)
 ((regexp "\\.ss$") racket)
 ((regexp "\\.rkt$") racket))
```

which describes the simple heuristic that files with '`.scm`' as extension are by default associated to a Guile REPL while those ending in '`.ss`' or '`.rkt`' correspond to Racket's implementation (with the caveat that these rules are applied only if the previous heuristics

have failed to detect the correct implementation, and that they'll match only if the corresponding implementation is active). You can add rules to `geiser-implementations-alist` (or replace all of them) by customizing it. Besides regular expressions, you can also use a directory name; for instance, the following snippet:

```
(eval-after-load "geiser-impl"
  '(add-to-list 'geiser-implementations-alist
                '((dir "/home/jao/prj/frob") guile)))
```

will add a new rule that says that any file inside my '`/home/jao/prj/frob`' directory (or, recursively, any of its children) is to be assigned to Guile. Since rules are first matched, first served, this new rule will take precedence over the default ones.

Switching between source files and the REPL

Once you have a working *geiser-mode*, you can switch from Scheme source buffers to the REPL or `C-c C-z`. Those shortcuts map to the interactive command `switch-to-geiser`.

If you use a numeric prefix, as in `C-u C-c C-z`, besides being teleported to the REPL, the latter will switch to the namespace of the Scheme source file (as if you had used `C-c C-m` in the REPL, with the source file's module as argument; cf. Section 3.3 [Switching context], page 6). This command is also bound to `C-c C-Z`, with a capital zed.

Once you're in the REPL, the same `C-c C-z` shortcut will bring you back to the buffer you jumped from, provided you don't kill the Scheme process in between. This is why the command is called *switch-to-geiser* instead of *switch-to-repl*, and what makes it really handy, if you ask me.

If for some reason you're not happy with the Scheme implementation that Geiser has assigned to your file, you can change it with `C-c C-s`, and probably take a look at [the previous subsection], page 12 to make sure that Geiser doesn't get confused again.

A note about context

As explained before (see Section 1.1 [Modus operandi], page 1), all Geiser activities take place in the context of the *current namespace*, which, for Scheme buffers, corresponds to the module that the Scheme implementation associates to the source file at hand (for instance, in Racket, there's a one to one correspondence between paths and modules, while Guile relies on explicit `define-module` forms in the source file).

Now that we have `geiser-mode` happily alive in our Scheme buffers and communicating with the right REPL instance, let us see what it can do for us, besides jumping to and fro.

4.3 Documentation helpers

Autodoc redux

The first thing you will notice by moving around Scheme source is that, every now and then, the echo area lightens up with the same autodoc messages we know and love from our REPL forays. This happens every time the Scheme process is able to recognise an identifier in the buffer, and provide information either on its value (for variables) or on its arity and the name of its formal arguments (for procedures and macros). That information will only be available if the module the identifier belongs to has been loaded in the running Scheme image. So it can be the case that, at first, no autodoc is shown for identifiers defined in

the file you're editing. But as soon as you evaluate them (either individually or collectively using any of the devices described in Section 4.4 [To eval or not to eval], page 14) their signatures will start appearing in the echo area.

Autodoc activation is controlled by a minor mode, `geiser-autodoc`, which you can toggle with *M-x* `geiser-autodoc`, or its associated keyboard shortcut, *C-c C-d* `a`. That `/A` indicator in the mode-line is telling you that autodoc is active. If you prefer, for some obscure reason, that it be inactive by default, just set `geiser-mode-autodoc-p` to `nil` in your customization files.

The way autodoc displays information deserves some explanation. It will first show the name of the module where the identifier at hand is defined, followed by a colon and the identifier itself. If the latter corresponds to a procedure or macro, it will be followed by a list of argument names, starting with the ones that are required. Then there comes a list of optional arguments, if any, enclosed in parenthesis. When an optional argument has a default value (or a form defining its default value), autodoc will display it after the argument name. When the optional arguments are keywords, their names are prefixed with "#:" (i.e., their names *are* keywords). An ellipsis (...) serves as a marker of an indeterminated number of parameters, as is the case with *rest* arguments or when autodoc cannot fathom the exact number of arguments (this is often the case with macros defined using `syntax-case`). Another way in which autodoc displays its ignorance is by using and underscore to display parameters whose name is beyond its powers.

It can also be the case that a function or macro has more than one signature (e.g., functions defined using `case-lambda`, or some `syntax-rules` macros, for which Geiser has often the black magic necessary to retrieve their actual arities). In those cases, autodoc shows all known signatures (using the above rules for each one) separated by a vertical bar (`|`).

As you have already noticed, the whole autodoc message is enclosed in parenthesis. After all, we're talking about Scheme here.

Finally, life is much easier when your cursor is on a symbol corresponding to a plain variable: you'll see in the echo area its name, preceded by the module where it's defined, and followed by its value, with an intervening arrow for greater effect. This time, there are no enclosing parenthesis (i hope you see the logic in my madness).

You can change the way Geiser displays the module/identifier combo by customizing `geiser-autodoc-identifier-format`. For example, if you wanted a tilde surrounded by spaces instead of a colon as a separator, you would write something like

```
(setq geiser-autodoc-identifier-format "%s ~ %s")
```

in your Emacs initialization files. There's also a face (`geiser-font-lock-autodoc-identifier`) that you can customize (for instance, with *M-x* `customize-face`) to change the appearance of the text. And another one (`geiser-font-lock-autodoc-current-arg`) that controls how the current argument position is highlighted.

Other documentation commands

Sometimes, autodoc won't provide enough information for you to understand what a function does. In those cases, you can ask Geiser to ask the running Scheme for further information on a given identifier or module.

For symbols, the incantation is *M-x geiser-doc-symbol-at-point*, or *C-c C-d C-d* for short. If the associated scheme supports docstrings (as, for instance, Guile does), you'll be teleported to a new Emacs buffer displaying Geiser's documentation browser, filled with information about the identifier, including its docstring (if any; unfortunately, that an implementation supports docstrings doesn't mean that they're used everywhere).

Pressing `q` in the documentation buffer will bring you back, enlightened, to where you were. There's also a handful of other navigation commands available in that buffer, which you can discover by means of its menu or via the good old *C-h m* command. And feel free to use the navigation buttons and hyperlinks that justify my calling this buffer a documentation browser.

For Racket, which does not support docstrings out of the box, this command will provide less information, but the documentation browser will display the corresponding contract when it's available, as well as some other tidbits for re-exported identifiers.

You can also ask Geiser to display information about a module, in the form of a list of its exported identifiers, using *C-c C-d C-m*, exactly as you would do [in the REPL], page 7.

In both cases, the documentation browser will show a couple of buttons giving you access to further documentation. First, you'll see a button named *source*: pressing it you'll jump to the symbol's definition. The second button, dubbed *manual*, will open the scheme implementation's manual page for the symbol at hand. For Racket, that will open your web browser displaying the corresponding reference's page (using Emacs' `browser-url` command), while in Guile a lookup will be performed in the texinfo manual.

You can also jump directly to the manual page for the symbol at point with the command `geiser-doc-look-up-manual`, bound to *C-c C-d i*.

See also our Section 5.3 [cheat-sheet], page 18 for a list of navigation commands available in the documentation browser.

4.4 To eval or not to eval

One of Geiser's main goals is to facilitate incremental development. You might have noticed that i've made a big fuss of Geiser's ability to recognize context, by being aware of the namespace where its operations happen.

That awareness is specially important when evaluating code in your scheme buffers, using the commands described below. They allow you to send code to the running Scheme with a granularity ranging from whole files to single s-expressions. That code will be evaluated in the module associated with the file you're editing, allowing you to redefine values and procedures to your heart's (and other modules') content.

Macros are, of course, another kettle of fish: one needs to re-evaluate uses of a macro after redefining it. That's not a limitation imposed by Geiser, but a consequence of how macros work in Scheme (and other Lisps). There's also the risk that you lose track of what's actually defined and what's not during a given session. But, in my opinion, those are limitations we lispers are aware of, and they don't force us to throw the baby with the bathwater and ditch incremental evaluation. Some people disagree; if you happen to find their arguments convincing, you don't have to throw away Geiser together with the baby: *M-x geiser-restart-repl* will let you restart the REPL as many times as you see fit.

For all of you bearded old lispers still with me, here are some of the commands performing incremental evaluation in Geiser.

`geiser-eval-last-sexp`, bound to *C-x C-e*, will eval the s-expression just before point.

`geiser-eval-definition`, bound to *C-M-x*, finds the topmost definition containing point and sends it for evaluation. The variant `geiser-eval-definition-and-go` (*C-c M-e*) works in the same way, but it also teleports you to REPL after the evaluation.

`geiser-eval-region`, bound to *C-c C-r*, evals the current region. Again, there's an *and go* version available, `geiser-eval-region-and-go`, bound to *C-c M-r*.

For all the commands above, the result of the evaluation is displayed in the minibuffer, unless it causes a (scheme-side) error (see Section 4.5 [To err perchance to debug], page 15).

At the risk of repeating myself, i'll remember you that all these evaluations will take place in the namespace of the module corresponding to the Scheme file from which you're sending your code, which, in general, will be different from the REPL's current module. And, if all goes according to plan, (re)defined variables and procedures should be immediately visible inside and, if exported, outside their module.

Besides evaluating expressions, definitions and regions, you can also macro-expand them. The corresponding keybindings start with the prefix *C-c C-m* and end, respectively, with *C-e*, *C-x* and *C-r*. The result of the macro expansion always appears in a pop up buffer.

4.5 To err: perchance to debug

When an error occurs during evaluation, it will be reported according to the capabilities of the underlying Scheme REPL.

In Racket, you'll be presented with a backtrace, in a new buffer where file paths locating the origin of the error are clickable (you can navigate them using the `TAB` key, and use `RET` or the mouse to jump to the offending spot; or invoke Emacs' stock commands `next-error` and `previous-error`, bound to *M-g n* and *M-g p* by default).

The Racket backtrace also highlights the exception type, making it clickable. Following the link will open the documentation corresponding to said exception type. Both the error and exception link faces are customizable (`geiser-font-lock-error-link` and `geiser-font-lock-doc-link`).

On the other hand, Guile's reaction to evaluation errors is different: it enters the debugger in its REPL. Accordingly, the REPL buffer will pop up if your evaluation fails in a Guile file, and the error message and backtrace will be displayed in there, again clickable and all. But there you have the debugger at your disposal, with the REPL's current module set to that of the offender, and a host of special debugging commands that are described in Guile's fine documentation.

In addition, Guile will sometimes report warnings for otherwise successful evaluations. In those cases, it won't enter the debugger, and Geiser will report the warnings in a debug buffer, as it does for Racket. You can control how picky Guile is reporting warnings by customizing the variable `geiser-guile-warning-level`, whose detailed docstring (which see, using, e.g. *C-h v*) allows me to offer no further explanation here. The customization group *geiser-guile* is also worth a glance, for a couple of options to fine tune how Geiser interacts with Guile's debugger (and more). Same thing for racketeers and *geiser-racket*.

4.6 Jumping around

This one feature is as sweet as easy to explain: *M-.* (`geiser-edit-symbol-at-point`) will open the file where the identifier around point is defined and land your point on its definition. To return to where you were, press *M-,* (`geiser-pop-symbol-stack`). This command works also for module names: Geiser first tries to locate a definition for the identifier at point and, if that fails, a module with that name; if the latter succeeds, the file where the module is defined will pop up.

Sometimes, the underlying Scheme will tell Geiser only the file where the symbol is defined, but Geiser will use some heuristics (read, regular expressions) to locate the exact line and bring you there. Thus, if you find Geiser systematically missing your definitions, send a message to the mailing list and we'll try to make the algorithm smarter.

You can control how the destination buffer pops up by setting `geiser-edit-symbol-method` to either `nil` (to open the file in the current window), `'window` (other window in the same frame) or `'frame` (in a new frame).

4.7 Geiser writes for you

No self-respecting programming mode would be complete without completion. In geiser-mode, identifier completion is bound to *M-TAB*, and will offer all visible identifiers starting with the prefix before point. Visible here means all symbols imported or defined in the current namespace plus locally bound ones. E.g., if you're at the end of the following partial expression:

```
(let ((default 42))
  (frob def
```

and press *M-TAB*, one of the possible completions will be `default`.

After obtaining the list of completions from the running Scheme, Geiser uses the standard Emacs completion machinery to display them. That means, among other things, that partial completion is available: just try to complete `d-s` or `w-o-t-s` to see why this is a good thing. Partial completion won't work if you have disabled it globally in your Emacs configuration: if you don't know what i'm talking about, never mind: Geiser's partial completion will work for you out of the box.

If you find the *M* modifier annoying, you always have the option to activate `geiser-smart-tab-mode`, which will make the `TAB` key double duty as the regular Emacs indentation command (when the cursor is not near a symbol) and Geiser's completion function. If you want this smarty pants mode always on in Scheme buffers, customize `geiser-mode-smart-tab-p` to `t`.

Geiser also knows how to complete module names: if no completion for the prefix at point is found among the currently visible bindings, it will try to find a module name that matches it. You can also request explicitly completion only over module names using *M-‘* (that's a backtick).

Besides completion, there's also this little command, `geiser-squarify`, which will toggle the delimiters of the innermost list around point between round and square brackets. It is bound to *C-c C-e [*. With a numeric prefix (as in, say, *M-2 C-c C-e [*), it will perform that many toggles, forward for positive values and backward for negative ones.

5 Cheat sheet

5.1 Scheme buffers

Key	Command	Description
C-c C-z	geiser-mode-switch-to-repl	Switch to REPL
C-u C-c C-z	geiser-mode-switch-to-repl	Switch to REPL and current module
C-c C-s	geiser-set-scheme	Specify Scheme implementation for buffer
M-.	geiser-edit-symbol-at-point	Go to definition of identifier at point
M-,	geiser-pop-symbol-stack	Go back to where M-. was last invoked
C-c C-e C-m	geiser-edit-module	Ask for a module and open its file
C-c C-e C-[	geiser-squarify	Toggle between () and [] for current form
C-M-x	geiser-eval-definition	Eval definition around point
C-c M-e	geiser-eval-definition-and-go	Eval definition around point and switch to REPL
C-x C-e	geiser-eval-last-sexp	Eval sexp before point
C-c C-r	geiser-eval-region	Eval region
C-c M-r	geiser-eval-region-and-go	Eval region and switch to REPL
C-c C-m C-x	geiser-expand-definition	Macro-expand definition around point
C-c C-m C-e	geiser-expand-last-sexp	Macro-expand sexp before point
C-c C-m C-r	geiser-expand-region	Macro-expand region
C-c C-k	geiser-compile-current-buffer	Compile and load current file
M-g n, C-x ‘	next-error	Jump to the location of next error
M-g p	previous-error	Jump to the location of previous error
C-c C-d C-d	geiser-doc-symbol-at-point	See documentation for identifier at point
C-c C-d C-m	geiser-doc-module	See a list of a module’s exported identifiers
C-c C-d C-i	geiser-doc-look-up-manual	Look up manual for symbol at point
C-c C-d C-a	geiser-autodoc-mode	Toggle autodoc mode
C-c<	geiser-xref-callers	Show callers of procedure at point

C-c>	geiser-xref-callees	Show callees of procedure at point
M-TAB	geiser-completion–complete-symbol	Complete identifier at point
M-‘, C-.	geiser-completion–complete-module	Complete module name at point

5.2 REPL

Key	Command	Description
C-c C-z	switch-to-geiser	Start Scheme REPL, or jump to previous buffer
C-c C-q	geiser-repl-exit	Kill Scheme process
M-.	geiser-edit-symbol-at-point	Edit identifier at point
TAB	geiser-completion–tab	Complete, indent or go to next error
S-TAB (backtab)	geiser-completion–previous-error	Go to previous error in the REPL buffer
M-TAB	geiser-completion–complete-symbol	Complete identifier at point
M-‘, C-.	geiser-completion–complete-module	Complete module name at point
M-p, M-n	(comint commands)	Prompt history, matching current prefix
C-c M-p, C-c M-n	(comint commands)	Previous/next prompt inputs
C-c C-m	switch-to-geiser-module	Set current module
C-c C-i	geiser-repl-import-module	Import module into current namespace
C-c C-d C-d	geiser-doc-symbol-at-point	See documentation for symbol at point
C-c C-d C-i	geiser-doc-look-up-manual	Look up manual for symbol at point
C-c C-d C-m	geiser-repl–doc-module	See documentation for module
C-c C-d C-a	geiser-autodoc-mode	Toggle autodoc mode

5.3 Documentation browser

Key	Command	Description
TAB, n	forward-button	Next link
S-TAB, p	backwards-button	Previous link
N	geiser-doc-next-section	Next section
P	geiser-doc-previous-section	Previous section
f	geiser-doc-next	Next page
b	geiser-doc-previous	Previous page

k	geiser-doc-kill-page	Kill current page and go to previous or next
g, r	geiser-doc-refresh	Refresh page
c	geiser-doc-clean-history	Clear browsing history
., M-.	geiser-doc-edit-symbol-at-point	Edit identifier at point
z	geiser-doc-switch-to-repl	Switch to REPL
q	View-quit	Bury buffer

6 No hacker is an island

Andy Wingo, Geiser's first user, has been a continuous source of encouragement and suggestions, and keeps improving Guile and heeding my feature requests.

The nice thing about collaborating with Andreas Rottmann over all these years is that he will not only make your project better with insightful comments and prodding: he'll send you patches galore too.

Ludovic Courts, #geiser's citizen no. 1, joined the fun after a while, and has since then been a continuous source of encouragement, ideas and bug reports.

Eduardo Cavazos' contagious enthusiasm has helped in many ways to keep Geiser alive, and he's become its best evangelist in R6RS circles.

Eli Barzilay took the time to play with an early alpha and made many valuable suggestions, besides answering all my 'how do you in PLT' questions.

Matthew Flatt, Robby Findler and the rest of the PLT team did not only answer my inquiries, but provided almost instant fixes to the few issues i found.

Thanks also to the PLT and Guile communities, for showing me that Geiser was not only possible, but a pleasure to hack on. And to the Slime hackers, who led the way.

Index

T

U

V