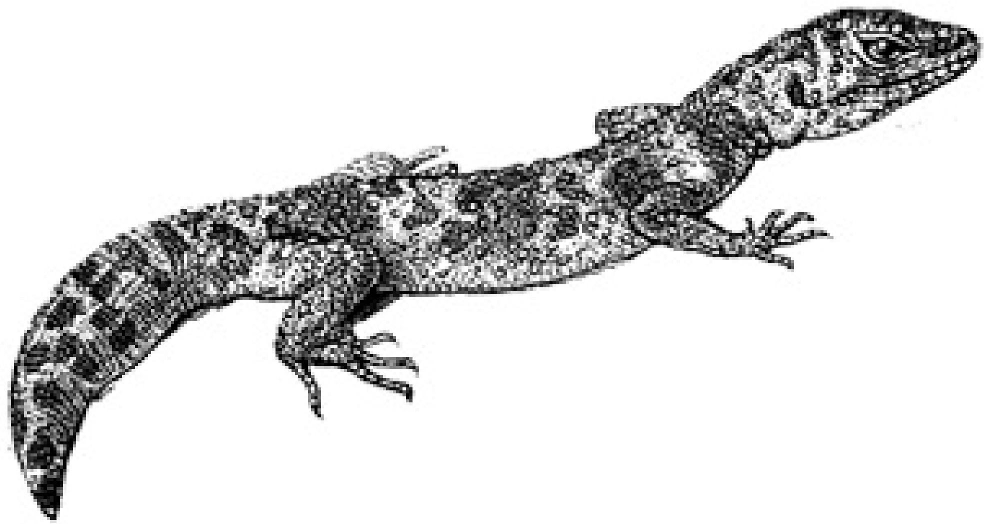
Lisp series



the Common Lisp Cookbook

Diving in

O RLY?

Collective

The Common Lisp Cookbook

Diving into the programmable programming language

The Common Lisp Cookbook contributors

The Common Lisp Cookbook

Home

> Cookbook, n. a book containing recipes and other information about the preparation and cooking of food.

A Cookbook is an invaluable resource, as it shows how to do various things in a clear fashion without all the theoretical context. Sometimes you just need to look things up. While cookbooks can never replace proper documentation such as the HyperSpec or books such as Practical Common Lisp, every language deserves a good cookbook, Common Lisp included.

The CL Cookbook aims to tackle all sort of topics, for the beginner as for the more advanced developer.

Content

Getting started

Language basics

Advanced topics

Outside world

Download in EPUB

The Cookbook is also available in EPUB (and PDF) format.

You can download it directly in EPUB and PDF, and you can **pay what you want** to further support its development:

Donate and download the EPUB version

Thank you!

Translations

The Cookbook has been translated to:

- Chinese simplified (Github)

- Portuguese (Brazilian) (Github)

Other CL Resources

- lisp-lang.org: success stories, tutorials and style guide
- Awesome-cl, a curated list of libraries
- List of Lisp Communities
- Lisp Koans - a language learning exercise, which guides the learner progressively through many language features.
- Learn X in Y minutes - Where X = Common Lisp - Small Common Lisp tutorial covering the essentials.
- Common Lisp Libraries Read the Docs - the documentation of popular libraries ported to the modern and good looking Read The Docs style.
- lisp-tips
- Articulate Common Lisp, an initiation manual for the uninitiated
- Lisp and Elements of Style by Nick Levine
- Pascal Costanza's Highly Opinionated Guide to Lisp
- Cliki, Common Lisp's wiki
- Common Lisp programming: from novice to effective developer, a video course on the Udemy platform (paywall), by one of the main Cookbook contributor. *"Thanks for supporting my work on Udemy. You can ask me for a free coupon if you are a student."* vindarel

and also: Common Lisp Pitfalls by Jeff Dalton.

Books

- Practical Common Lisp by Peter Seibel
- Common Lisp Recipes by Edmund Weitz, published in 2016,
- Common Lisp: A Gentle Introduction to Symbolic Computation by David S. Touretzky
- Successful Lisp: How to Understand and Use Common Lisp by David B. Lamkins
- On Lisp by Paul Graham
- Common Lisp the Language, 2nd Edition by Guy L. Steele
- A Tutorial on Good Lisp Style by Peter Norvig and Kent Pitman

Advanced books

- Loving Lisp - the Savy Programmer's Secret Weapon by Mark Watson
- Programming Algorithms - A comprehensive guide to writing efficient programs with examples in Lisp.

Specifications

- The Common Lisp HyperSpec by Kent M. Pitman (also available in Dash, Zeal and Velocity)
- The Common Lisp Community Spec - a new rendering produced from the ANSI specification draft, that everyone has the right to edit.

Further remarks

This is a collaborative project that aims to provide for Common Lisp something similar to the Perl Cookbook published by O'Reilly. More details about what it is and what it isn't can be found in this thread from comp.lang.lisp.

If you want to contribute to the CL Cookbook, please send a pull request in or file a ticket!

Yes, we're talking to you! We need contributors - write a chapter that's missing and add it, find an open question and provide an answer, find bugs and report them, (If you have no idea what might be missing but would like to help, take a look at the table of contents of the Perl Cookbook.) Don't worry about the formatting, just send plain text if you like - we'll take care about that later.

Thanks in advance for your help!

The pages here on Github are kept up to date. You can also download a up to date zip file for offline browsing. More info can be found at the Github project page.

License

> Cookbook, n. a book containing recipes and other information about the preparation and cooking of food.

The Common Lisp Cookbook is a collaborative resource to help you learn Common Lisp the language, its ecosystem and to get you started in a wide range of programming areas. It can be used by Lisp newcomers as a tutorial (getting started, functions, etc) and by everybody as a reference (loop!).

We hope that these EPUB and PDF versions make the learning experience even more practical and enjoyable.

Vincent "vindarel" Dardel, for the Cookbook contributors

Getting started with Common Lisp

We'll begin with presenting easy steps to install a development environment and to start a new Common Lisp project.

Want a 2-clicks install? Then get Portacle, *a portable and multi-platform* Common Lisp environment. It ships Emacs, SBCL (the implementation), Quicklisp (package manager), SLIME (IDE) and Git. It's the most straightforward way to get going!

Install an implementation

With your package manager

If you don't know which implementation of Common Lisp to use, try SBCL:

```
apt-get install sbcl
```

Common Lisp has been standardized via an ANSI document, so it can be implemented in different ways. See Wikipedia's list of implementations.

The following implementations are packaged for Debian and most other popular Linux distributions:

- Steel Bank Common Lisp (SBCL)
- Embeddable Common Lisp (ECL), which compiles to C,
- CLISP

Other well-known implementations include:

- ABCL, to interface with the JVM,
- ClozureCL, a good implementation with very fast build times (see this Debian package for Clozure CL),

- CLASP, that interoperates with C++ libraries using LLVM for compilation to native code,
- AllegroCL (proprietary)
- LispWorks (proprietary)

and older implementations:

- CMUCL, originally developed at Carnegie Mellon University, from which SBCL is derived, and
- GNU Common Lisp
- and there is more!

With the asdf-vm package manager

The asdf-vm tool can be used to manage a large ecosystem of runtimes and tools.

- Steel Bank Common Lisp (SBCL) is available via this plugin for asdf-vm

With Roswell

Roswell is:

- an implementation manager: it makes it easy to install a Common Lisp implementation (`ros install ecl`), an exact version of an implementation (`ros install sbcl/1.2.0`), to change the default one being used (`ros use ecl`),
- a scripting environment (helps to run Lisp from the shell, to get the command line arguments,…),
- a script installer,
- a testing environment (to run tests, including on popular Continuous Integration platforms),
- a building utility (to build images and executables in a portable way).

You'll find several ways of installation on its wiki (Debian package, Windows installer, Brew/Linux Brew,…).

With Docker

If you already know Docker, you can get started with Common Lisp pretty quickly. The clfoundation/cl-devel image comes with recent versions of SBCL, CCL, ECL and ABCL, plus Quicklisp installed in the home (`/home/cl`), so than we can `ql:quickload` libraries straight away.

Docker works on GNU/Linux, Mac and Windows.

The following command will download the required image (around 1.0GB compressed), put your local sources inside the Docker image where indicated, and drop you into an SBCL REPL:

```
docker run --rm -it -v /path/to/local/code:/home/cl/common-lisp/source clfoundation/cl-devel:latest sbcl
```

We still want to develop using Emacs and SLIME, so we need to connect SLIME to the Lisp inside Docker. See slime-docker, which is a library that helps on setting that up.

On Windows

All implementations above can be installed on Windows.

Portacle is multiplatform and works on Windows.

You can also try:

- ρEmacs, a preconfigured distribution of GNU Emacs specifically for Microsoft Windows. It ships with many CL implementations: CCL, SBCL, CLISP, ABCL and ECL, and also has components for other programming languages (Python, Racket, Java, C++…).
- Corman Lisp, for Windows XP, Windows 2000, Windows ME or Windows NT. It is fully integrated with the Win32 API, and all the Windows API functions are readily available from Lisp.

Start a REPL

Just launch the implementation executable on the command line to enter the REPL (Read Eval Print Loop), i.e. the interactive interpreter.

Quit with `(quit)` or `ctr-d` (on some implementations).

Here is a sample session:

```
user@debian:~$ sbcl
This is SBCL 1.3.14.debian, an implementation of ANSI Common
Lisp.
More information about SBCL is available at
<http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no
warranty.
It is mostly in the public domain; some portions are provided
under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
* (+ 1 2)

3
* (quit)
user@debian:~$
```

You can slightly enhance the REPL (the arrow keys do not work, it has no history,…) with `rlwrap`:

```
apt-get install rlwrap
```

and:

```
rlwrap sbcl
```

But we'll setup our editor to offer a better experience instead of working in this REPL. See editor-support.

Lisp is interactive by nature, so in case of an error we enter the debugger. This can be annoying in certain cases, so you might want to use SBCL's `--disable-debugger` option.

> **TIP:** The CLISP implementation has a better default REPL for the terminal (readline capabilities, completion of symbols). You can even use `clisp -on-error abort` to have error messages without the debugger. It's handy to try things out, but we recommend to set-up your editor and to use SBCL or CCL.

> **TIP:** By adding the `-c` switch to rlwrap, you can autocomplete file names.

Libraries

Common Lisp has thousands of libraries available under a free software license. See:

- Quickdocs - the library documentation hosting for CL.
- the Awesome-cl list, a curated list of libraries.
- Cliki, the Common Lisp wiki.

Some terminology

- In the Common Lisp world, a **package** is a way of grouping symbols together and of providing encapsulation. It is similar to a C++ namespace, a Python module or a Java package.

- A **system** is a collection of CL source files bundled with an .asd file which tells how to compile and load them. There is often a one-to-one relationship between systems and packages, but this is in no way mandatory. A system may declare a dependency on other systems. Systems are managed by ASDF (Another System Definition Facility), which offers functionalities similar to those of `make` and `ld.so`, and has become a de facto standard.

- A Common Lisp library or project typically consists of one or several ASDF systems (and is distributed as one Quicklisp project).

Install Quicklisp

Quicklisp is more than a package manager, it is also a central repository (a *dist*) that ensures that all libraries build together.

It provides its own *dist* but it is also possible to build our own.

To install it, we can either:

1- run this command, anywhere:

```
curl -O https://beta.quicklisp.org/quicklisp.lisp
```

and enter a Lisp REPL and load this file:

```
sbcl --load quicklisp.lisp
```

or

2- install the Debian package:

```
apt-get install cl-quicklisp
```

and load it, from a REPL:

```
(load "/usr/share/common-lisp/source/quicklisp/quicklisp.lisp")
```

Then, in both cases, still from the REPL:

```
(quicklisp-quickstart:install)
```

This will create the `~/quicklisp/` directory, where Quicklisp will maintain its state and downloaded projects.

If you wish, you can install Quicklisp to a different location. For instance, to install it to a hidden folder on Unix systems:

```
(quicklisp-quickstart:install :path "~/.quicklisp")
```

If you want Quicklisp to always be loaded in your Lisp sessions, run `(ql:add-to-init-file)`: this adds the right stuff to the init file of your CL implementation. Otherwise, you have to run `(load "~/quicklisp/setup.lisp")` in every session if you want to use Quicklisp or any of the libraries installed through it.

It adds the following in your (for example) `~/.sbclrc`:

```
#-quicklisp
  (let ((quicklisp-init (merge-pathnames
                         "quicklisp/setup.lisp"
                         (user-homedir-pathname))))
    (when (probe-file quicklisp-init)
      (load quicklisp-init)))
```

Install libraries

In the REPL:

```
(ql:quickload "system-name")
```

For example, this installs the "str" string manipulation library:

```
(ql:quickload "str")
```

and voilà. You can use it right away:

```
(str:title-case "HELLO LISP!")
```

> **SEE MORE:** To understand the `package:a-symbol` notation, read the packages page, section "Accessing symbols from a package".

We can install more than one library at once. Here we install cl-ppcre for regular-expressions, and Alexandria, a utility library:

```
(ql:quickload '("str" "cl-ppcre" "alexandria"))
```

Anytime you want to use a third-party library in your Lisp REPL, you can run this `ql:quickload` command. It will not hit the network a second time if it finds that the library is already installed on your file system. Libraries are by default installed in `~/quicklisp/dist/quicklisp/`.

Note also that dozens of Common Lisp libraries are packaged in Debian. The package names usually begin with the cl- prefix (use `apt-cache search --names-only "^cl-.*"` to list them all).

For example, in order to use the `cl-ppcre` library, one should first install the `cl-ppcre` package.

Then, in SBCL, it can be used like this:

```
(require "asdf")
(require "cl-ppcre")
(cl-ppcre:regex-replace "fo+" "foo bar" "frob")
```

Here we pretend we don't have Quicklisp installed and we use `require` to load a module that is available on the file system. In doubt, you can use `ql:quickload`.

See Quicklisp's documentation for more commands. For instance, see how to upgrade or rollback your Quicklisp's distribution.

Advanced dependencies management

You can drop Common Lisp projects into any of those folders:

- `~/quicklisp/local-projects`
- `~/common-lisp`,
- `~/.local/share/common-lisp/source`,

A library installed here is automatically available for every project.

For a complete list, see the values of:

```
(asdf/source-registry:default-user-source-registry)
```

and

```
asdf:*central-registry*
```

Providing our own version of a library. Cloning projects.

Given the property above, we can clone any library into the `~/quicklisp/local-projects/` directory and it will be found by ASDF (and Quicklisp) and available right-away:

```
(asdf:load-system "system")
```

or

```
(ql:quickload "system")
```

The practical different between the two is that `ql:quickload` first tries to fetch the system from the Internet if it is not already installed.

Note that symlinks in local-projects to another location of your liking works too.

How to work with local versions of libraries

If we need libraries to be installed locally, for only one project, or in order to easily ship a list of dependencies with an application, we can use Qlot or CLPM.

Quicklisp also provides Quicklisp bundles. They are self-contained sets of systems that are exported from Quicklisp and loadable without involving Quicklisp.

At last, there's [Quicklisp controller] to help us build *dists*.

Working with projects

Now that we have Quicklisp and our editor ready, we can start writing Lisp code in a file and interacting with the REPL.

But what if we want to work with an existing project or create a new one, how do we proceed, what's the right sequence of `defpackage`, what to put in the `.asd` file, how to load the project into the REPL ?

Creating a new project

Some project builders help to scaffold the project structure. We like cl-project that also sets up a tests skeleton.

In short:

```
(ql:quickload "cl-project")
(cl-project:make-project #P"./path-to-project/root/")
```

it will create a directory structure like this:

```
|-- my-project.asd
|-- my-project-test.asd
|-- README.markdown
|-- README.org
|-- src
|   `-- my-project.lisp
`-- tests
    `-- my-project.lisp
```

Where `my-project.asd` resembles this:

```
(asdf:defsystem "my-project"
  :version "0.1.0"
  :author ""
  :license ""
  :depends-on ()  ;; <== list of Quicklisp dependencies
  :components ((:module "src"
```

```
              :components
              ((:file "my-project"))))
  :description ""
  :long-description
  #.(read-file-string
     (subpathname *load-pathname* "README.markdown"))
  :in-order-to ((test-op (test-op "my-project-test"))))
```

and `src/my-project.lisp` this:

```
(defpackage footest
  (:use :cl))
(in-package :footest)
```

- ASDF documentation: [defining a system with defsystem]()

How to load an existing project

You have created a new project, or you have an existing one, and you want to work with it on the REPL, but Quicklisp doesn't know it. How can you do ?

Well first, if you create it or clone it into one of `~/common-lisp`, `~/.local/share/common-lisp/source/` or `~/quicklisp/local-projects`, you'll be able to `(ql:quickload …)` it with no further ado.

Otherwise you'll need to compile and load its system definition (`.asd`) first. In SLIME with the `slime-asdf` contrib loaded, type `C-c C-k` (*slime-compile-and-load-file*) in the `.asd`, then you can `(ql:quickload …)` it.

You can use `(asdf:load-asd "my-project.asd")` programmatically instead of `C-c C-k`.

Usually you want to "enter" the system in the REPL at this stage:

```
(in-package :my-project)
```

Lastly, you can compile or eval the sources (`my-project.lisp`) with `C-c C-k` or `C-c C-c` (*slime-compile-defun*) in a form, and see its result in the REPL.

Another solution is to use ASDF's list of known projects:

```
;; startup file like ~/.sbclrc
(pushnew "~/path-to/project/" asdf:*central-registry* :test #'ec
```

and since ASDF is integrated into Quicklisp, we can `quickload` our project right away.

Happy hacking !

More settings

You might want to set SBCL's default encoding format to utf-8:

```
(setf sb-impl::*default-external-format* :utf-8)
```

You can add this to your `~/.sbclrc`.

If you dislike the REPL to print all symbols upcase, add this:

```
(setf *print-case* :downcase)
```

Warning: This might break the behaviour of some packages like it happened with Mito. Avoid doing this in production.

See also

- cl-cookieproject - a project skeleton for a ready-to-use project with an entry point and unit tests. With a `src/` subdirectory, some more metadata, a 5AM test suite, a way to build a binary, an example CLI args parsing, Roswell integration.
- Source code organization, libraries and packages: https://lispmethods.com/libraries.html

Credits

- https://wiki.debian.org/CommonLisp

- http://articulate-lisp.com/project/new-project.html

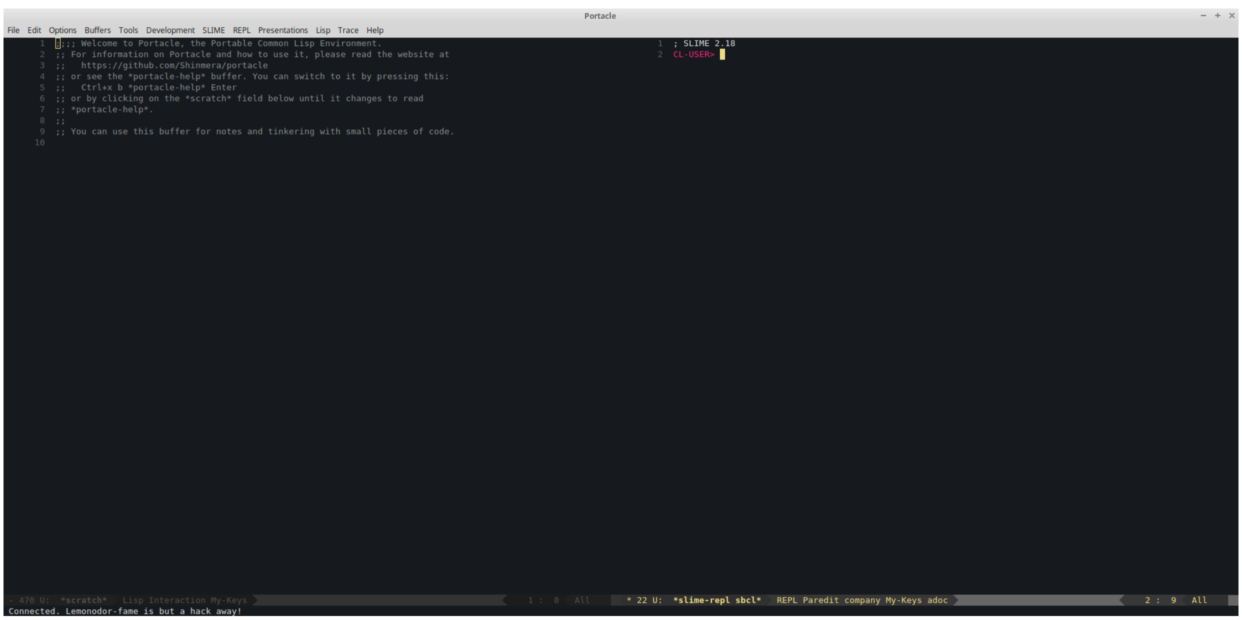
Editor support

The editor of choice is still Emacs, but it is not the only one.

Emacs

SLIME is the Superior Lisp Interaction Mode for Emacs. It has support for interacting with a running Common Lisp process for compilation, debugging, documentation lookup, cross-references, and so on. It works with many implementations.

Portacle is a portable and multi-platform Common Lisp environment. It ships Emacs, SBCL, Quicklisp, SLIME and Git.



Installing SLIME

SLIME is in the official GNU ELPA repository of Emacs Lisp packages (in Emacs24 and forward). Install with:

```
M-x package-install RET slime RET
```

Since SLIME is heavily modular and the defaults only do the bare minimum (not even the SLIME REPL), you might want to enable more features with

```
(slime-setup '(slime-fancy slime-quicklisp slime-asdf))
```

For more details, consult the documentation (also available as an Info page).

Now you can run SLIME with `M-x slime` and/or `M-x slime-connect`.

See also:

- https://wikemacs.org/wiki/SLIME - configuration examples and extensions.

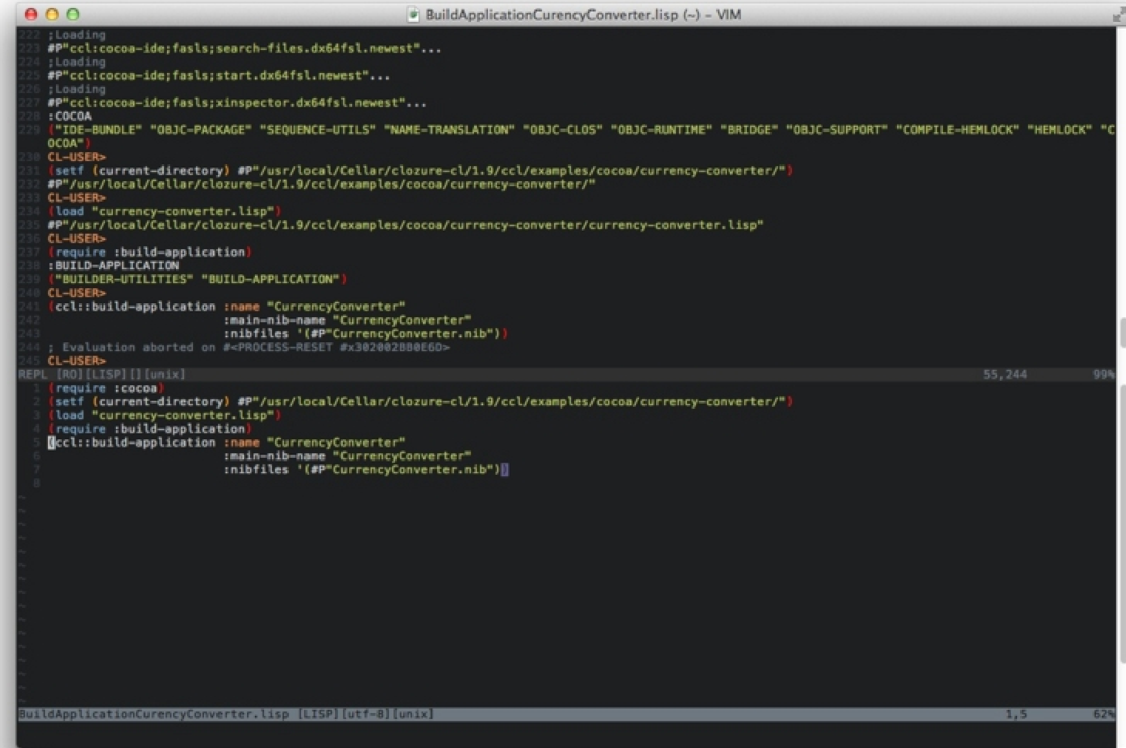
Using Emacs as an IDE

See “Using Emacs as an IDE”.

Vim & Neovim

Slimv is a full-blown environment for Common Lisp inside of Vim.

Vlime is a Common Lisp dev environment for Vim (and Neovim), similar to SLIME for Emacs and SLIMV for Vim.



cl-neovim makes it possible to write Neovim plugins in Common Lisp.

quicklisp.nvim is a Neovim frontend for Quicklisp.

Slimv_box brings Vim, SBCL, ABCL, and tmux in a Docker container for a quick installation.

See also:

- Lisp in Vim demonstrates usage and compares both Slimv and Vlime

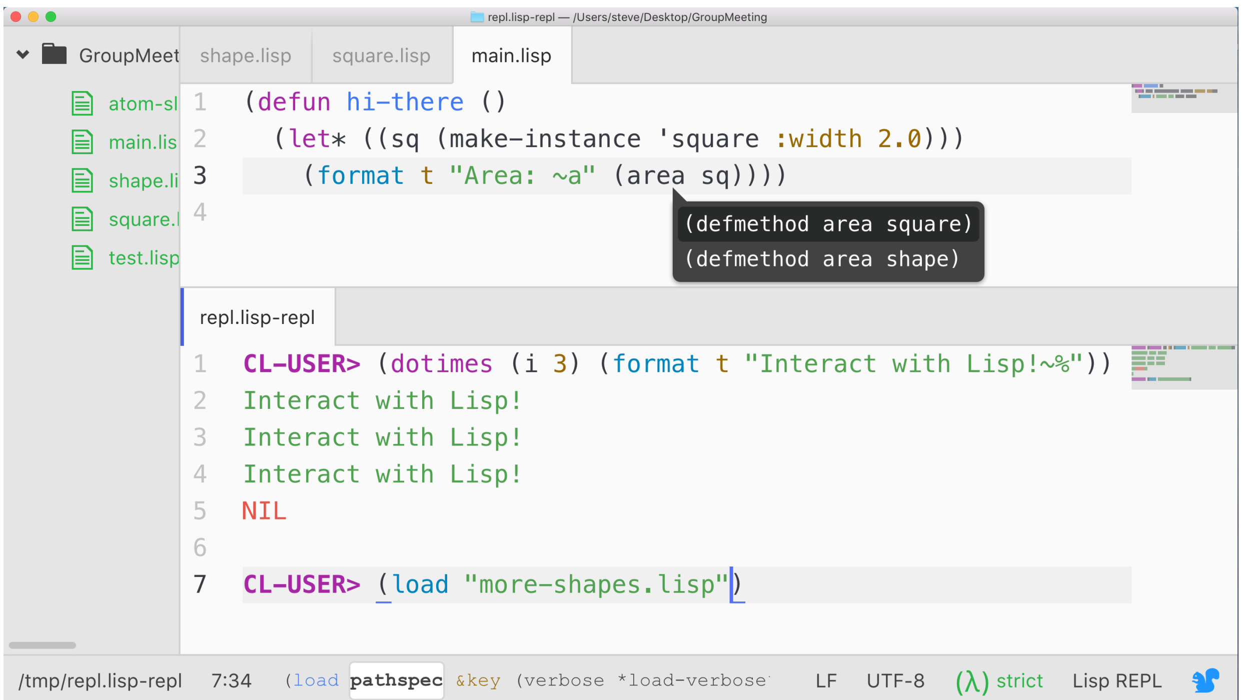
Pulsar (ex Atom)

See SLIMA. This package allows you to interactively develop Common Lisp code, turning Atom, or now Pulsar, into a pretty good Lisp IDE. It features:

- REPL

- integrated debugger
    - (not a stepping debugger yet)
- jump to definition
- autocomplete suggestions based on your code
- compile this function, compile this file
- function arguments order
- integrated profiler
- interactive object inspection.

It is based on the Swank backend, like Slime for Emacs.

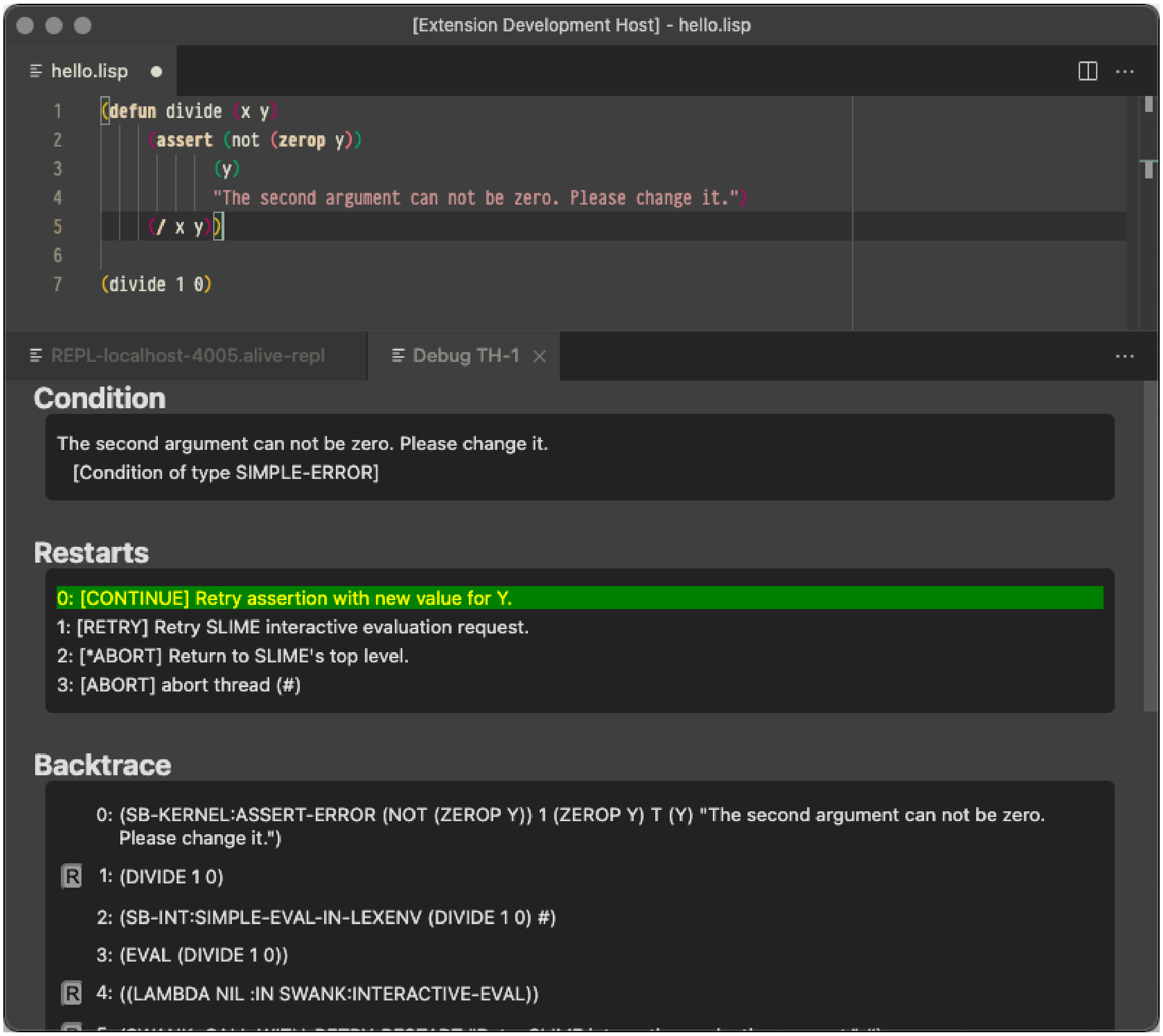


VSCode

[Alive](#) makes VSCode a powerful Common Lisp development. It hooks directly into the Swank server that Emacs Slime uses and is fully compatible with VSCode's ability to develop remotely in containers, WSL, Remote machines, etc. It has no dependencies beyond a version of Common Lisp on which to run the Swank server. It can be configured to run with Quicklisp, CLPM, and Roswell. It currently supports:

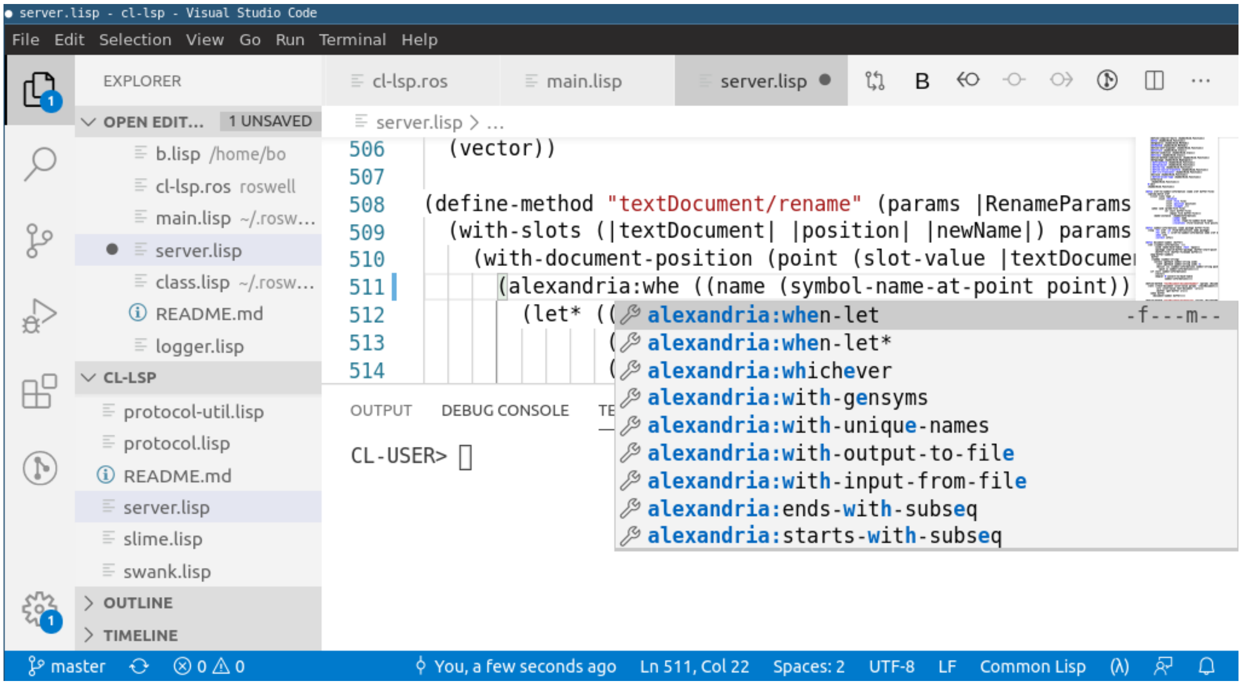
- Syntax highlighting

- Code completion
- Code formatter
- Jump to definition
- Snippets
- REPL integration
- Interactive Debugger
- REPL history
- Inline evaluation
- Macro expand
- Disassemble
- Inspector
- Hover Text
- Rename function args and let bindings
- Code folding



commonlisp-vscode extension works via the cl-lsp language server and it's possible to write LSP client that works in other editors. It depends heavily on Roswell. It currently supports:

- running a REPL
- evaluate code
- auto indent,
- code completion
- go to definition
- documentation on hover



Using VSCode with Alive

See Using VSCode with Alive.

JetBrains - NEW in Jan, 2023!

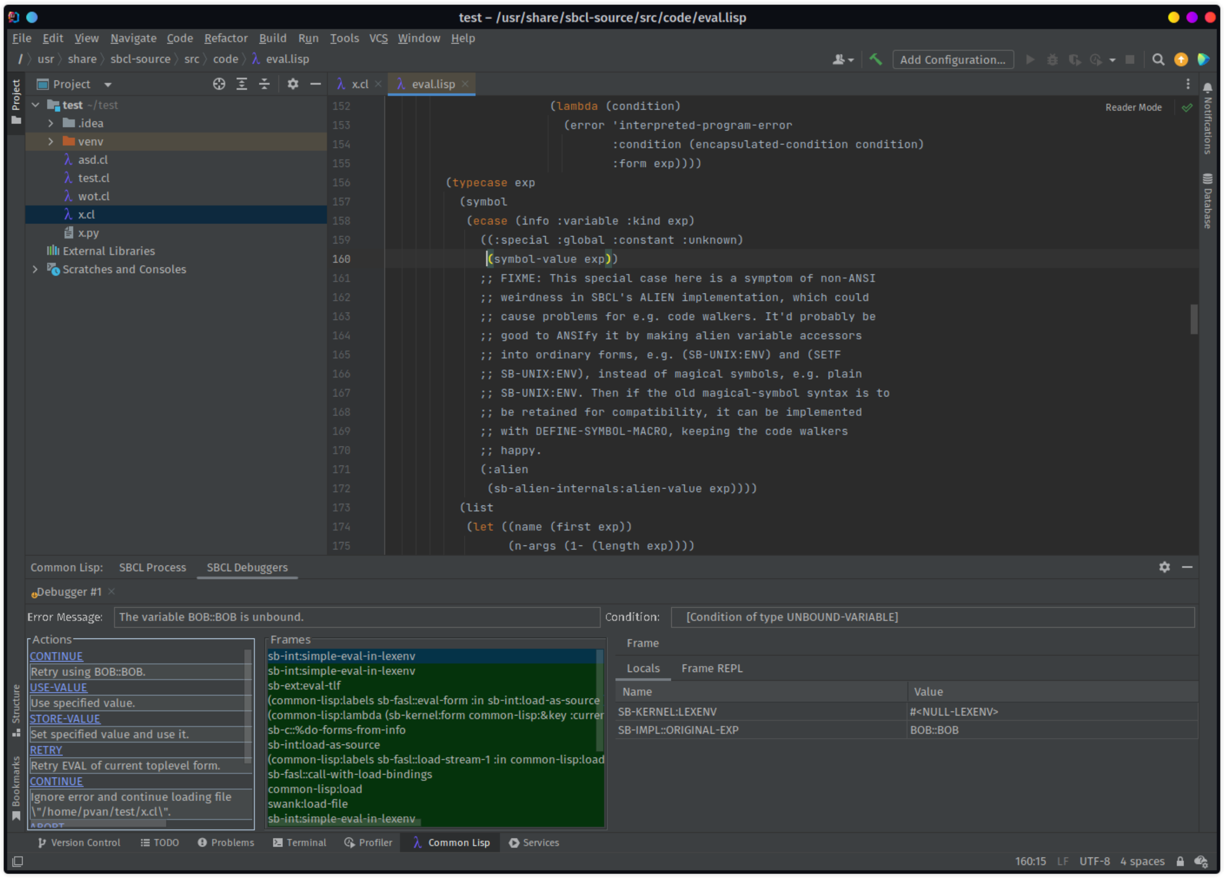
SLT is a new (published on January, 2023) plugin for the suite of JetBrains' IDEs. It uses a modified SLIME/Swank protocol to commmunicate with SBCL, providing IDE capabilities for Common Lisp.

It has a very good user guide.

At the time of writing, for its version 0.4, it supports:

- REPL
- symbol completion
- send expressions to the REPL
- interactive debugging, breakpoints
- documentation display
- cross-references
- find symbol by name, global class/symbol search

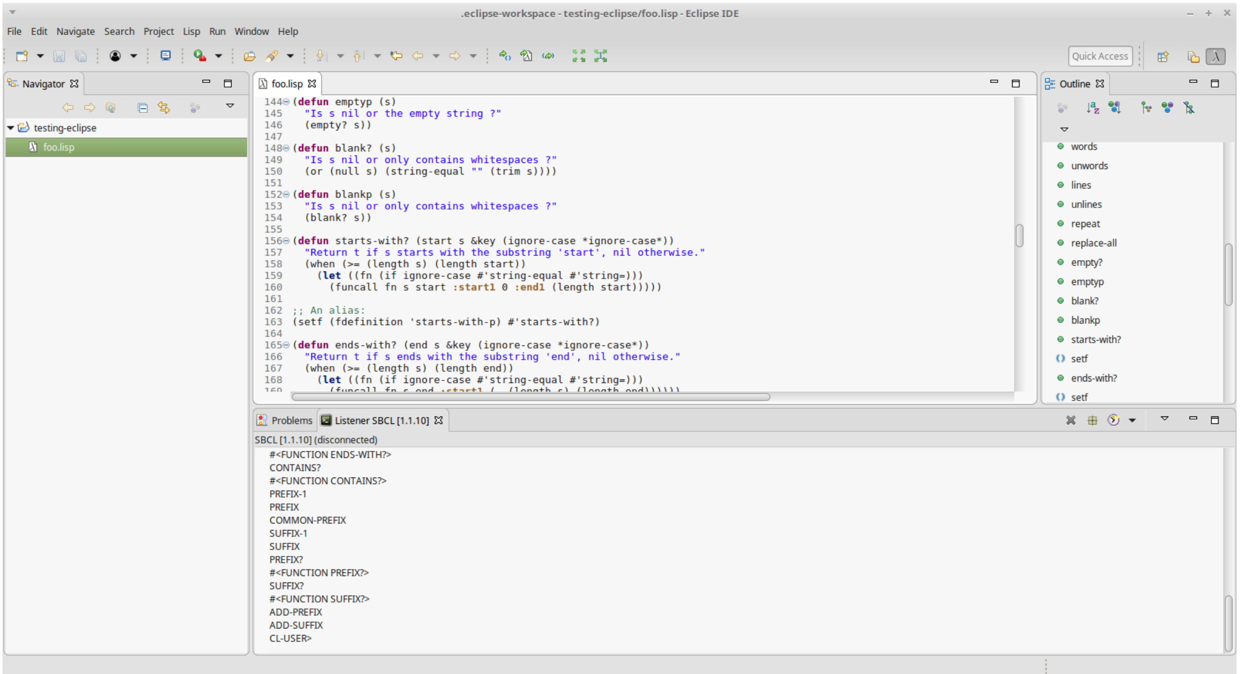
- inspector (read-only)
- graphical threads list
- SDK support, automatic download for Windows users
- multiple implementations support: SBCL, CCL, ABCL and AllegroCL.



Eclipse

Dandelion is a plugin for the Eclipse IDE.

Available for Windows, Mac and Linux, built-in SBCL and CLISP support and possibility to connect other environments, interactive debugger with restarts, macro-expansion, parenthesis matching,…



Lem

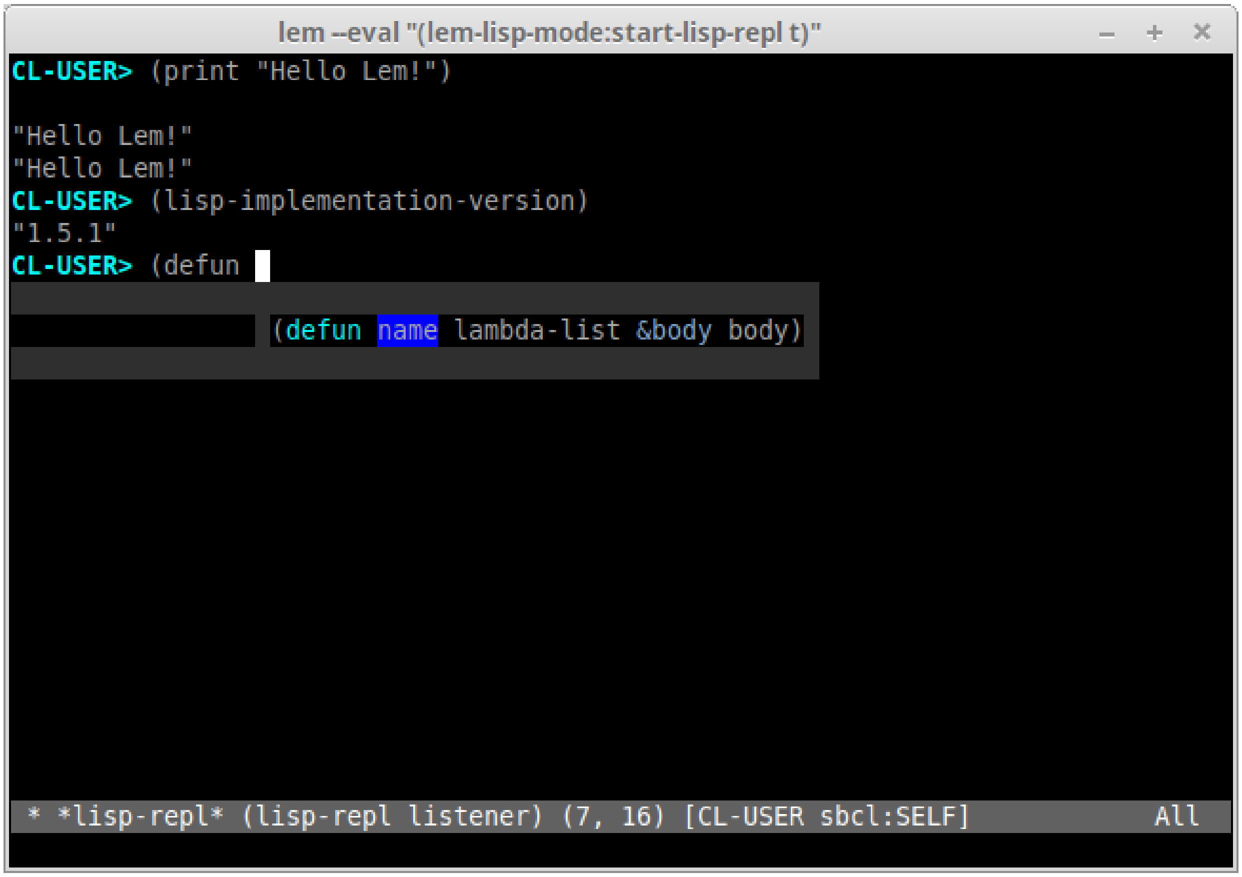
[Lem](#) is an editor tailored for Common Lisp development. Once you install it, you can start developing. Its interface resembles Emacs and SLIME (same shortcuts). It comes with an ncurses and an SDL2 frontend, and other programming modes thanks to its built-in LSP client: Python, Go, Rust, JS, Nim, Scheme, HTML, CSS, plus a directory mode, a **vim layer**, and more.

It can be started as a REPL right away in the terminal. Run it with:

```
lem --eval "(lem-lisp-mode:start-lisp-repl t)"
```

So you probably want a shell alias:

```
alias ilem='lem --eval "(lem-lisp-mode:start-lisp-repl t)"'
```

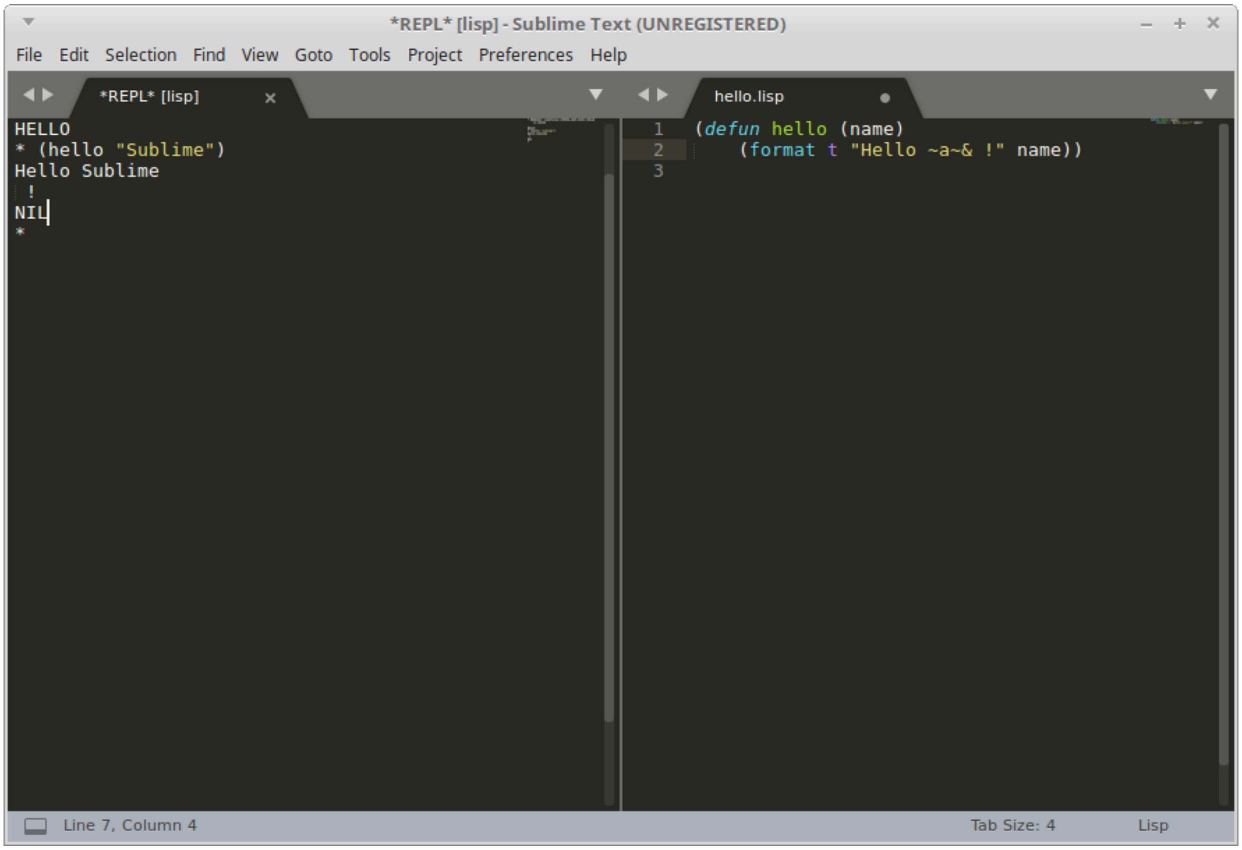


Sublime Text

Sublime Text has now good support for Common Lisp.

First install the "SublimeREPL" package and then see the options in Tools/SublimeREPL to choose your CL implementation.

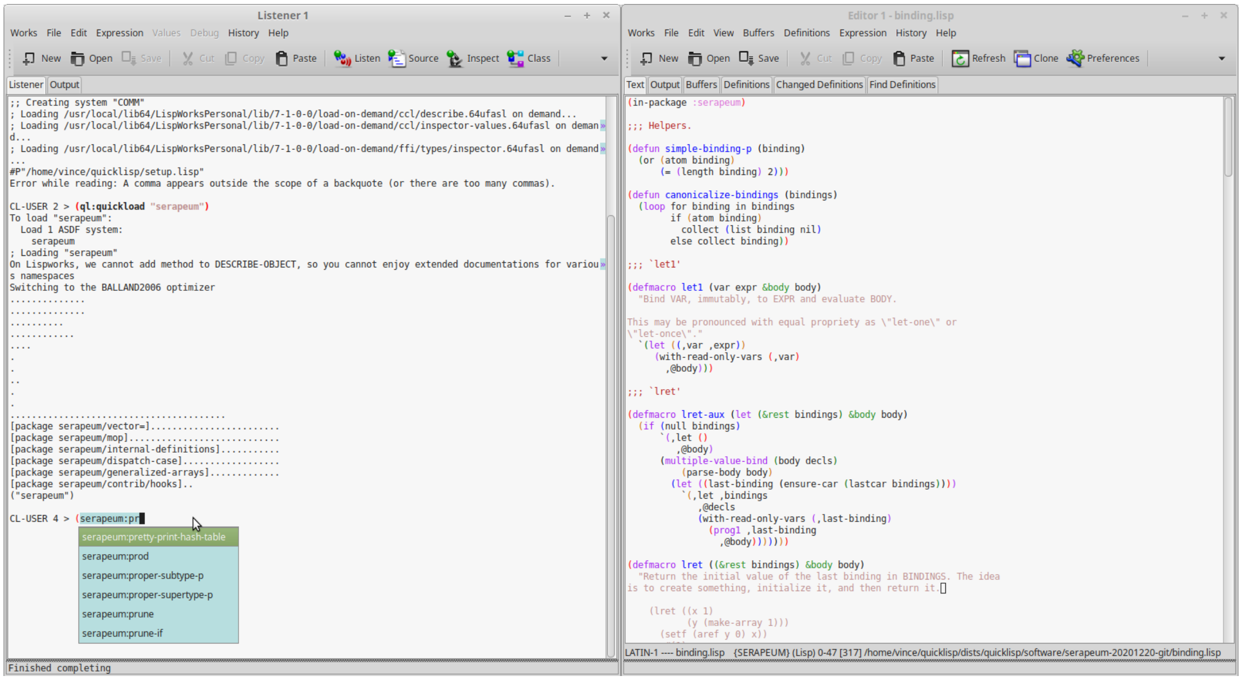
Then Slyblime ships IDE-like features to interact with the running Lisp image. It is an implementation of SLY and it uses the same backend (SLYNK). It provides advanced features including a debugger with stack frame inspection.



LispWorks (proprietary)

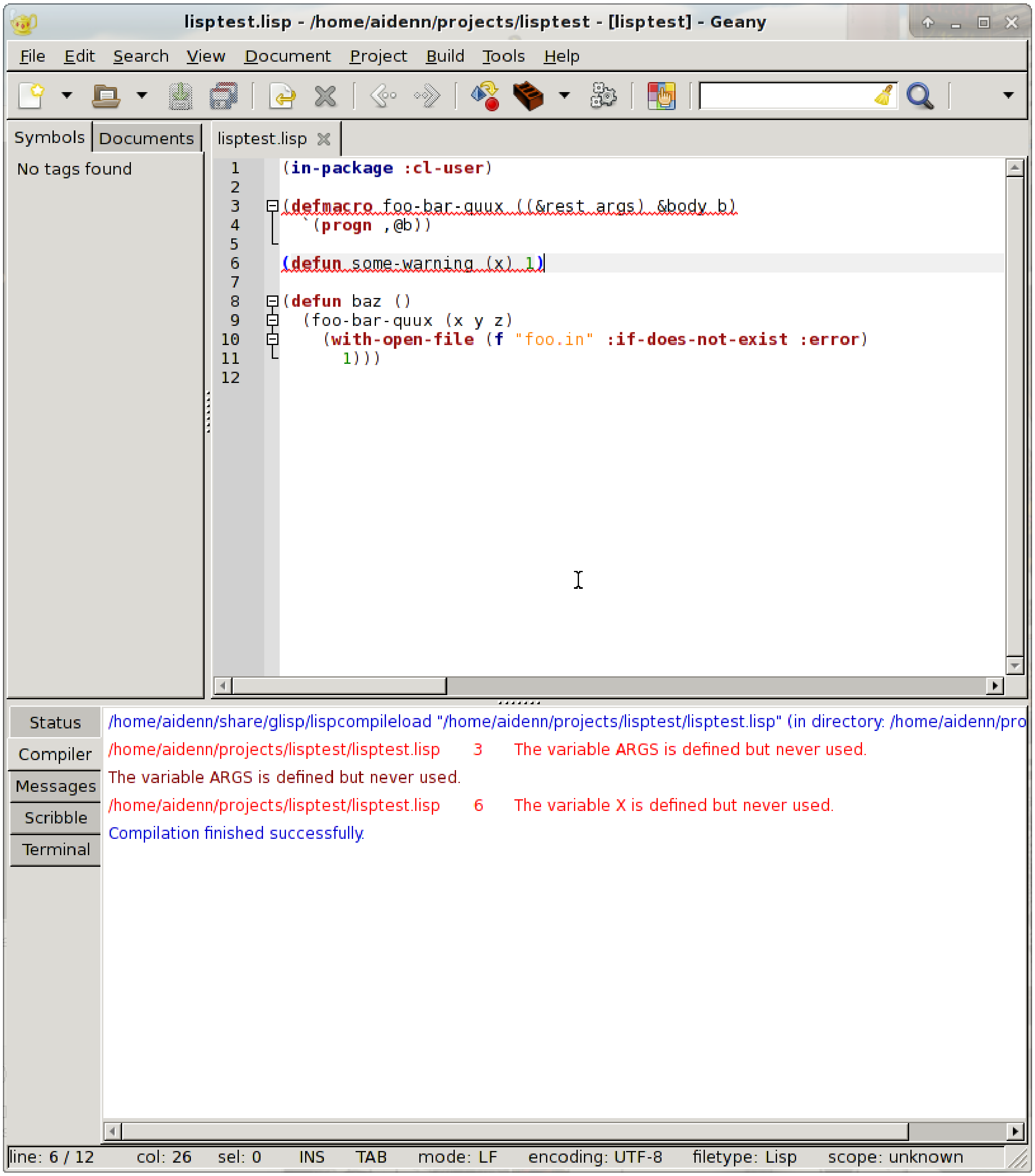
LispWorks is a Common Lisp implementation that comes with its own Integrated Development Environment (IDE) and its share of unique features, such as the CAPI GUI toolkit. It is **proprietary** and provides a **free limited version**.

You can read our LispWorks review here.



Geany (experimental)

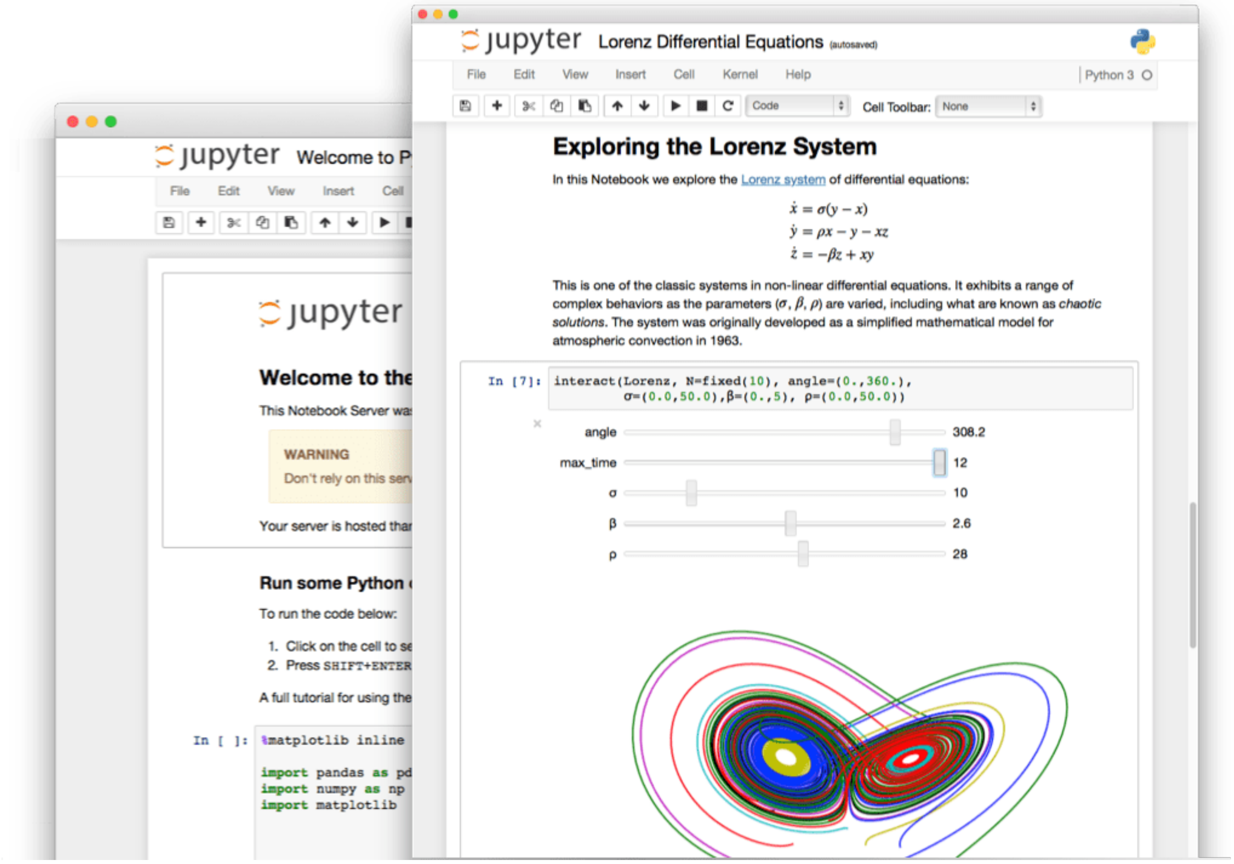
Geany-lisp is an experimental lisp mode for the Geany editor. It features completion of symbols, smart indenting, jump to definition, compilation of the current file and highlighting of errors and warnings, a REPL, and a project skeleton creator.



Notebooks

[common-lisp-jupyter](#) is a Common Lisp kernel for Jupyter notebooks.

You can [see a live Jupyter notebook written in Lisp here](). It is easy to install (Roswell, repo2docker and Docker recipes).

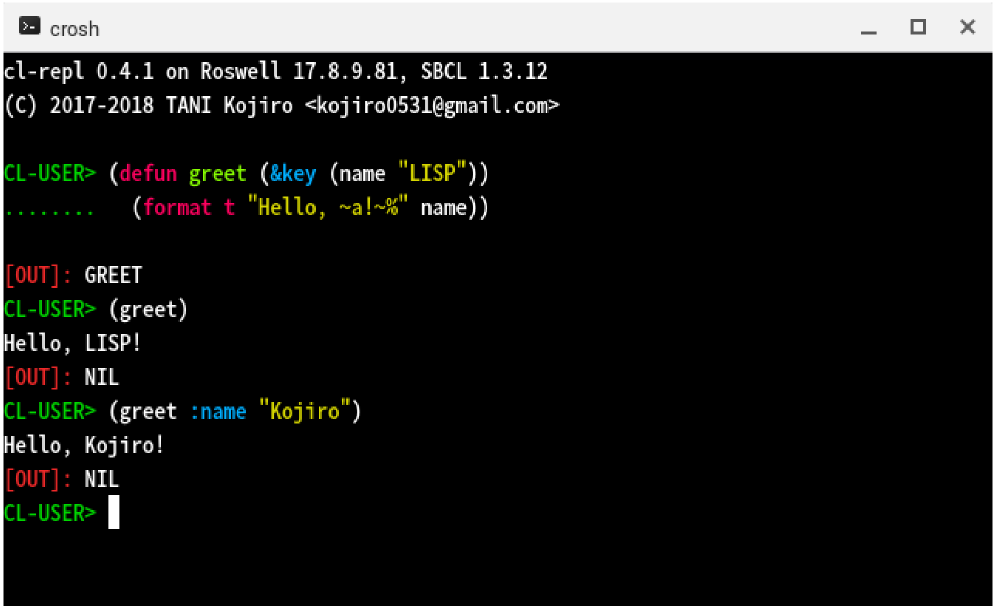


There is also [Darkmatter](), a notebook-style Common Lisp environment, built in Common Lisp.

REPLs

[cl-repl]() is an ipython-like REPL. It supports symbol completion, magic and shell commands, editing command in a file and a simple debugger.

You might also like [sbcli](), an even simpler REPL with readline capabilities. It handles errors gracefully instead of showing a debugger.



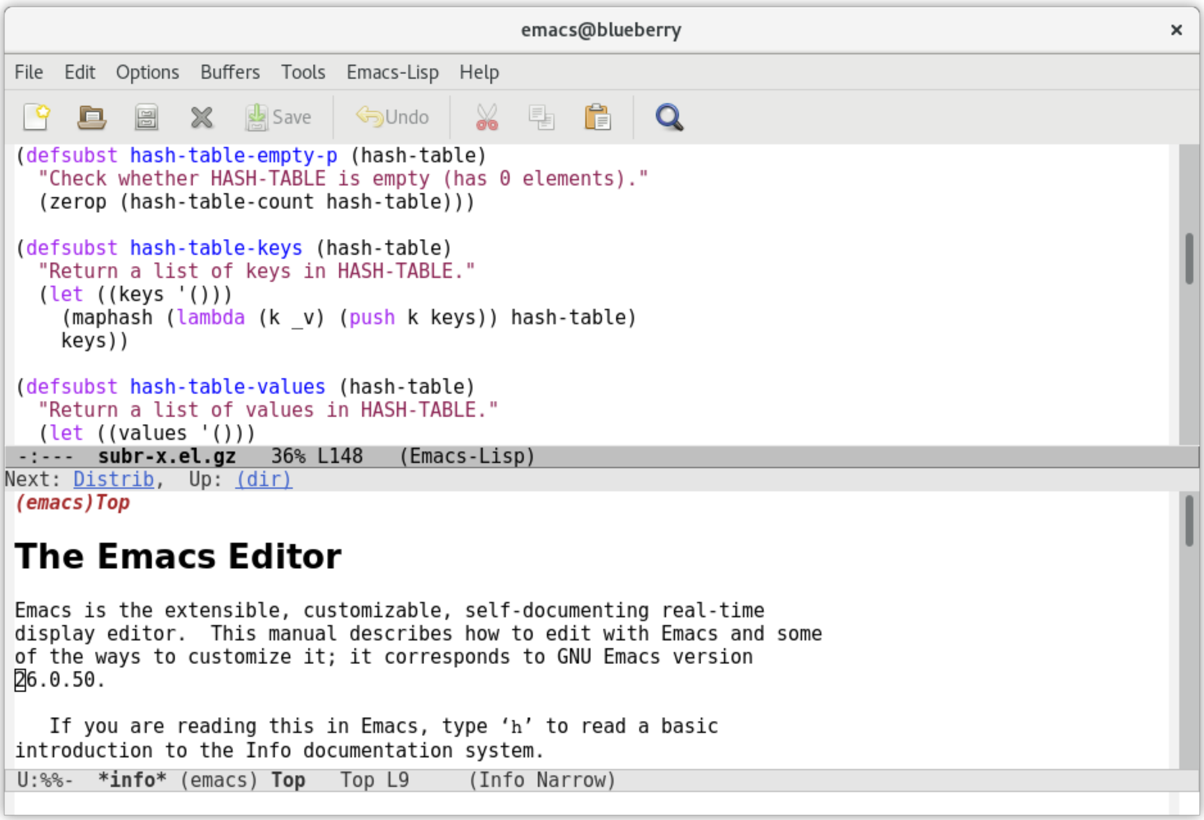
Others

There are some more editors out there, more or less discontinued, and free versions of other Lisp vendors, such as Allegro CL.

Emacs

Using Emacs as an IDE

This page is meant to provide an introduction to using Emacs as a Lisp IDE.



Note: Portacle is a portable and multi-platform CL development environment, a straightforward way to get going.

Why Use Emacs?

- Emacs has fantastic support for working with Lisp code
- Not tying yourself into a single CL vendor's editor
- Runs on virtually every OS and CL implementation
- Extensible: awesome-emacs.

- Can be customized to do many common tasks
- Built-in support for different source code version control systems
- Vast number of add-on packages
- Emacs will probably always be around
- Emacs works well either with a mouse or without a mouse
- Emacs works well either in GUI mode or in the terminal
- Emacs has a large user base with multiple newsgroups
- Benefits of using Emacs far outweigh the effort spent in learning it
- Because Org-mode
- Because Magit
- Because Emacs Rocks !

Emacs Lisp vs Common Lisp

- Learning Emacs Lisp is useful and similar (but different from CL):
  - Dynamic scope is everywhere
  - There are no reader (or reader-related) functions
  - Does not support all the types that are supported in CL
  - Incomplete implementation of CLOS (with the add-on EIEIO package)
  - Not all of CL is supported
  - No numerical tower support
- Some good Emacs Lisp learning resources:
  - An Introduction to Programming in Emacs Lisp
  - Writing GNU Emacs Extensions
  - Wikemacs

SLIME: Superior Lisp Interaction Mode for Emacs

SLIME is the goto major mode for CL programming.

- Pros:
  - Provides REPL which is hooked to implementation directly in Emacs
  - Has integrated Common Lisp debugger with Emacs interface
  - Interactive object-inspector in Emacs buffer
  - Has its own minor mode which enhances lisp-mode in many ways

    - Supports every common Common Lisp implementation
    - Readily available from MELPA
    - Actively maintained
    - Symbol completion
    - Cross-referencing
    - Can perform macroexpansions
- Cons:
    - Installing SLIME without MELPA can be tricky
- Setup:
    - Installing it from MELPA is straightforward. Search package-list-packages for 'slime' and click to install. If MELPA is configured correctly, it will install itself and all dependencies.
    - Enable the desired contribs (SLIME does very little by defaults), e.g. `(slime-setup '(slime-fancy slime-quicklisp slime-asdf))`.
    - Run SLIME with `M-x slime`.

Check out this **video tutorial** ! (and the author's channel, full of great stuff)

SLIME fancy, contrib packages and other extensions

SLIME's functionalities live in packages and so-called contrib modules must be loaded to add further functionalities. The default `slime-fancy` includes:

- slime-autodoc
- slime-c-p-c
- slime-editing-commands
- slime-fancy-inspector
- slime-fancy-trace
- slime-fontifying-fu
- slime-fuzzy
- slime-mdot-fu
- slime-macrostep
- slime-presentations
- slime-references
- slime-repl
- slime-scratch
- slime-package-fu

- slime-trace-dialog

SLIME also has some nice extensions like Helm-SLIME which features, among others:

- Fuzzy completion,
- REPL and connection listing,
- Fuzzy-search of the REPL history,
- Fuzzy-search of the *apropos* documentation.

REPL interactions

From the SLIME REPL, press `,` to prompt for commands. There is completion over the available systems and packages. Examples:

- `,load-system`
- `,reload-system`
- `,in-package`
- `,restart-inferior-lisp`

and many more.

With the `slime-quicklisp` contrib, you can also `,ql` to list all systems available for installation.

SLY: Sylvester the Cat's Common Lisp IDE

SLY is a SLIME fork that contains the following improvements:

- Completely redesigned REPL based on Emacs's own full-featured comint.el
- Live code annotations via a new sly-stickers contrib
- Consistent interactive button interface. Everything can be copied to the REPL.
- Multiple inspectors with independent history
- Regexp-capable M-x sly-apropos
- Contribs are first class SLY citizens, enabled by default, loaded with ASDF on demand.

- Support for NAMED-READTABLES, macrostep.el and quicklisp.

Finding one's way into Emacs' built-in documentation

Emacs comes with built-in tutorials and documentation. Moreover, it is a self-documented and self-discoverable editor, capable of introspection to let you know about the current keybindings, to let you search about function documentation, available variables,source code, tutorials, etc. Whenever you ask yourself questions like "what are the available shortcuts to do x" or "what does this keybinding really do", the answer is most probably a keystroke away, right inside Emacs. You should learn a few keybindings to be able to discover Emacs with Emacs flawlessly.

The help on the topic is here:

- Help page: commands for asking Emacs about its commands

The help keybindings start with either `C-h` or `F1`. Important ones are:

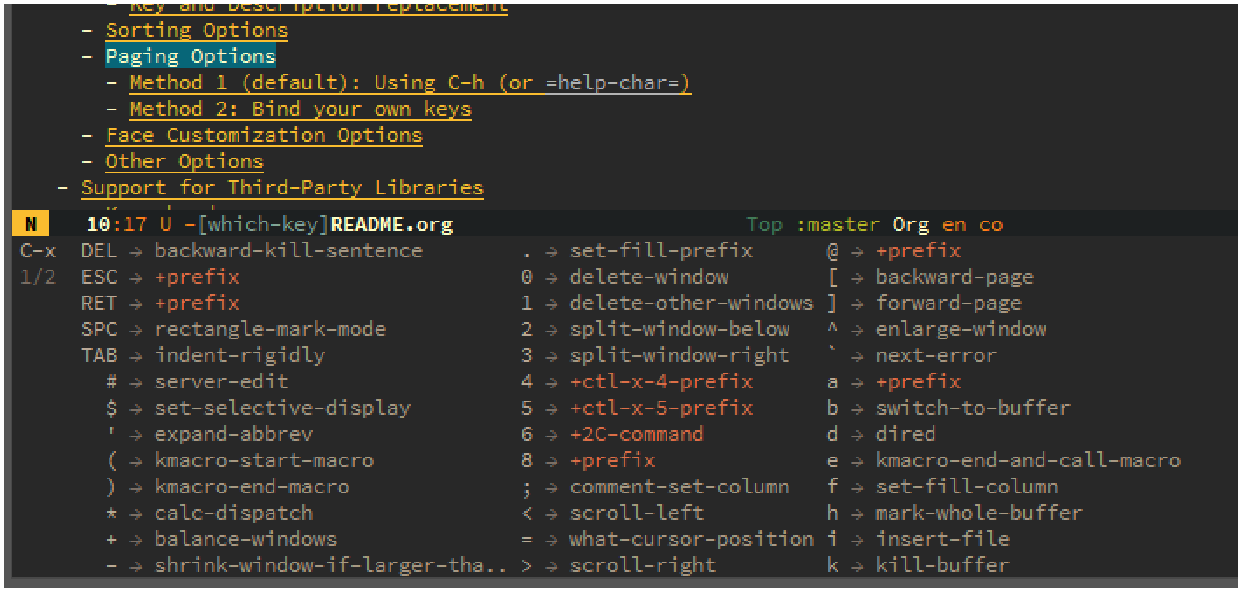
- `C-h k <keybinding>`: what function does this keybinding call?
- `C-h f <function name>`: what keybinding is linked to this function?
- `C-h a <topic>`: show a list of commands whose name match the given *topic*. It accepts a keyword, a list of keywords or a regular expression.
- `C-h i`: show the Info page, a menu of major topics.

Some Emacs packages give even more help.

More help and discoverability packages

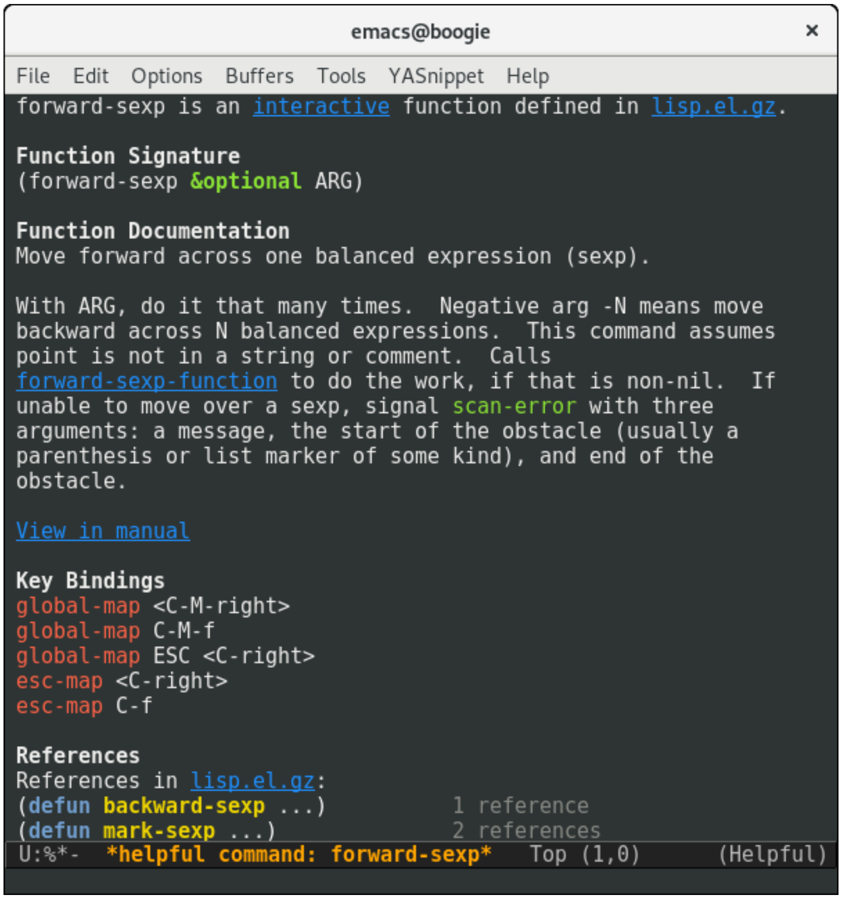
Sometimes, you start typing a key sequence but you can't remember it completely. Or, you wonder what other keybindings are related. Comes which-key-mode. This packages will display all possible keybindings starting with the key(s) you just typed.

For example, I know there are useful keybindings under `C-x` but I don't remember which ones… I just type `C-x`, I wait for half a second, and which-key shows all the ones available.



Just try it with `C-h` too!

See also [Helpful](), an alternative to the built-in Emacs help that provides much more contextual information.



Learn Emacs with the built-in tutorial

Emacs ships its own tutorial. You should give it a look to learn the most important keybindings and concepts.

Call it with `M-x help-with-tutorial` (where `M-x` is `alt-x`).

Working with Lisp Code

In this short tutorial we'll see how to:

- edit Lisp code
- evaluate and compile Lisp code
- search Lisp code

Packages for structured editing

In addition to the built-in Emacs commands, you have several packages at your disposal that will help to keep the parens and/or the indentation balanced. The list below is somewhat sorted by age of the extension, according to the history of Lisp editing:

- Paredit - Paredit is a classic. It defines the must-have commands (move, kill, split, join a sexp,…). (visual tutorial)
- Smartparens - Smartparens not only deals with parens but with everything that comes in pairs (html tags,…) and thus has features for non-lispy languages.
- Lispy - Lispy reimagines Paredit with the goal to have the shortest bindings (mostly one key) that only act depending on the point position.
- Paxedit - Paxedit adds commands based on the context (in a symbol, a sexp,…) and puts efforts on whitespace cleanup and context refactoring.
- Parinfer - Parinfer automatically fixes the parens depending on the indentation, or the other way round (or both !).

We personally advice to try Parinfer and the famous Paredit, then to go up the list. See explanations and even more on Wikemacs.

Editing

Emacs has, of course, built-in commands to deal with s-expressions.

Forward/Backward/Up/Down movement and selection by s-expressions

Use `C-M-f` and `C-M-b` (`forward-sexp` and `backward-sexp`) to move in units of s-expressions.

Use `C-M-t` to swap the first addition sexp and the second one. Put the cursor on the open parens of "(+ x" in defun c and press

Use `C-M-@` to highlight an entire sexp. Then press `C-M-u` to expand the selection "upwards" and `C-M-d` to move forward down one level of parentheses.

Deleting s-expressions

Use `C-M-k` (`kill-sexp`) and `C-M-backspace` (`backward-kill-sexp`) (but caution: this keybinding may restart the system on GNU/Linux).

For example, if point is before `(progn` (I'll use [] as an indication where the cursor is):

```
(defun d ()
  (if t
      (+ 3 3)
     [](progn
        (+ 1 1)
        (if t
            (+ 2 2)
            (+ 3 3)))
      (+ 4 4)))
```

and you press `C-M-k`, you get:

```
(defun d ()
  (if t
      (+ 3 3)
      []
      (+ 4 4)))
```

Indenting s-expressions

Indentation is automatic for Lisp forms.

Pressing TAB will indent incorrectly indented code. For example, put the point at the beginning of the `(+ 3 3)` form and press TAB:

```
(progn
(+ 3 3))
```

you correctly get

```
(progn
  (+ 3 3))
```

Use `C-M-q` (`slime-reindent-defun`) to indent the current function definition:

```lisp
;; Put the cursor on the open parens of "(defun ..."
;; and press "C-M-q" to indent the code:
(defun e ()
"A badly indented function."
(let ((x 20))
(loop for i from 0 to x
do (loop for j from 0 below 10
do (print j))
(if (< i 10)
(let ((z nil) )
(setq z (format t "x=~d" i))
(print z))))))

;; This is the result:

(defun e ()
  "A badly indented function (now correctly indented)."
  (let ((x 20))
    (loop for i from 0 to x
       do (loop for j from 0 below 10
             do (print j))
         (if (< i 10)
             (let ((z nil) )
               (setq z (format t "x=~d" i))
               (print z))))))
```

You can also select a region and call `M-x indent-region`.

Support for parenthesis

Use `M-(` to insert a pair of parenthesis (`()`) and the same keybinding with a prefix argument, `C-u M-(`, to enclose the expression in front of the cursor with a pair of parens.

For example, we start with the cursor before the first paren:

```
CL-USER> |(- 2 2)
```

Press `C-u M-(` to enclose it with parens:

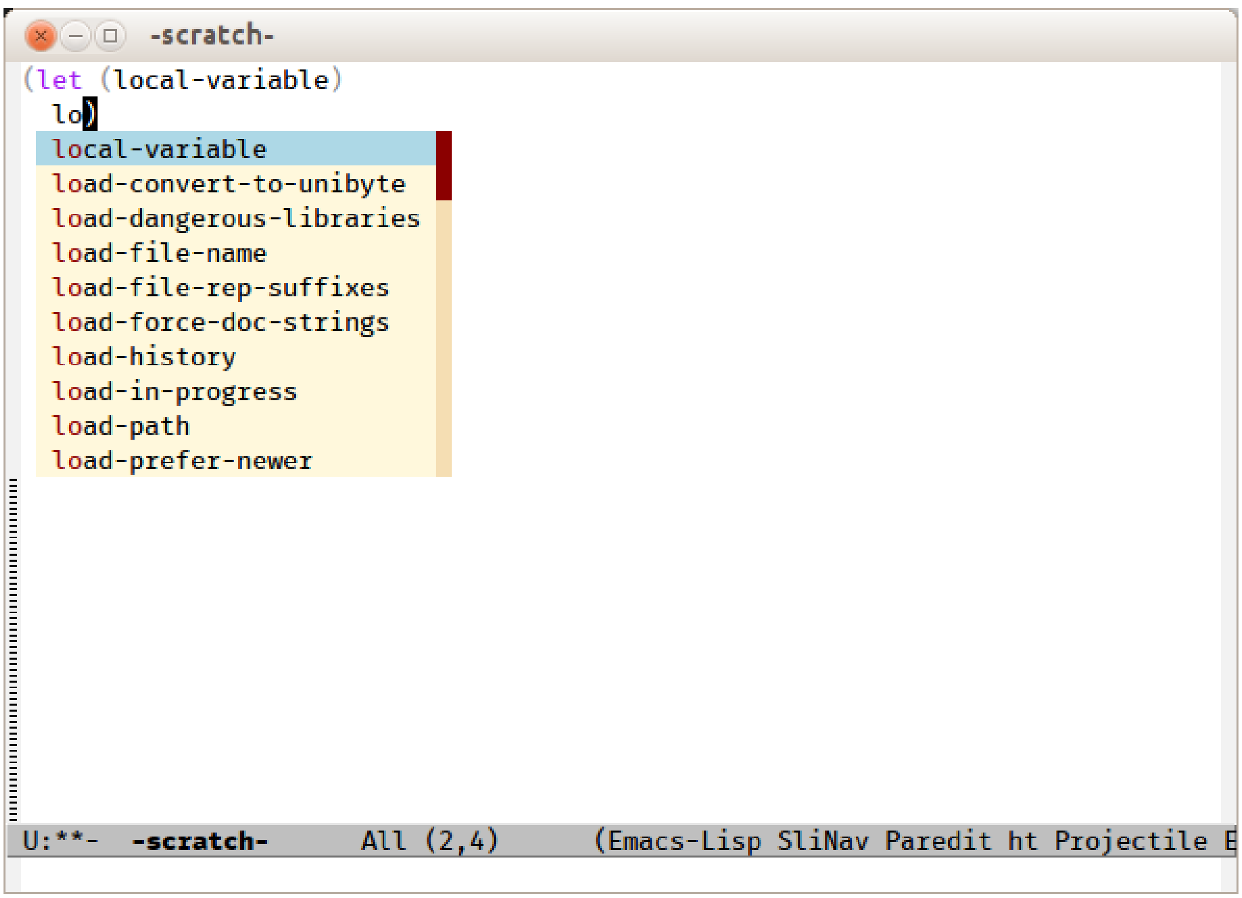
```
CL-USER> (|(- 2 2))
;; now write anything.
CL-USER> (zerop (- 2 2))
```

With a numbered prefix argument (`C-u 2 M-(`), wrap around this number of s-expressions.

Additionnaly, use `M-x check-parens` to spot malformed s-exps and `C-c C-]` (`slime-close-all-parens-in-sexp`) to insert the required number of closing parenthesis.

Code completion

Use the built-in `C-c TAB` to complete symbols in SLIME. You can get tooltips with company-mode.



In the REPL, it's simply TAB.

Use Emacs' hippie-expand, bound to `M-/`, to complete any string present in other open buffers.

Hiding/showing code

Use `C-x n n` (narrow-to-region) and `C-x n w` to widen back.

See also code folding.

Comments

Insert a comment, comment a region with `M-;`, adjust text with `M-q`.

Evaluating and Compiling Lisp in SLIME

Compile the entire **buffer** by pressing `C-c C-k` (`slime-compile-and-load-file`).

Compile a **region** with `M-x slime-compile-region`.

Compile a **defun** by putting the cursor inside it and pressing `C-c C-c` (`slime-compile-defun`).

To **evaluate** rather than compile:

- evaluate the **sexp** before the point by putting the cursor after its closing paren and pressing `C-x C-e` (`slime-eval-last-expression`). The result is printed in the minibuffer.
- similarly, use `C-c C-p` (`slime-pprint-eval-last-expression`) to eval and pretty-print the expression before point. It shows the result in a new "slime-description" window.
- evaluate a region with `C-c C-r`,
- evaluate a defun with `C-M-x`,
- type `C-c C-e` (`slime-interactive-eval`) to get a prompt that asks for code to eval in the current context. It prints the result in the minibuffer. With a prefix argument, insert the result into the current buffer.
- type `C-c C-j` (`slime-eval-last-expression-in-repl`), when the cursor is after the closing parenthesis of an expression, to send this

expression to the REPL and evaluate it.

See also other commands in the menu.

EVALUATION VS COMPILATION

There are a couple of pragmatic differences when choosing between compiling or evaluating. In general, it is better to *compile* top-level forms, for two reasons:

- Compiling a top-level form highlights warnings and errors in the editor, whereas evaluation does not.
- SLIME keeps track of line-numbers of compiled forms, but when a top-level form is evaluated, the file line number information is lost. That's problematic for code navigation afterwards.

`eval` is still useful to observe results from individual non top-level forms. For example, say you have this function:

```
(defun foo ()
  (let ((f (open "/home/mariano/test.lisp")))
    ...))
```

Go to the end of the OPEN expression and evaluate it (`C-x C-e`), to observe the result:

```
=> #<SB-SYS:FD-STREAM for "file /mnt/e6b00b8f-9dad-4bf4-bd40-34b1e6d31f0a/home/marian/test.lisp" {1003AAAB53}>
```

Or on this example, with the cursor on the last parentheses, press `C-x C-e` to evaluate the `let`:

```
(let ((n 20))
  (loop for i from 0 below n
     do (print i)))
```

You should see numbers printed in the REPL.

See also [eval-in-repl] to send any form to the repl.

Searching Lisp Code

Standard Emacs text search (isearch forward/backward, regexp searches, search/replace)

`C-s` does an incremental search forward (e.g. - as each key is the search string is entered, the source file is searched for the first match. This can make finding specific text much quicker as you only need to type in the unique characters. Repeat searches (using the same search characters) can be done by repeatedly pressing `C-s`

`C-r` does an incremental search backward

`C-s RET` and `C-r RET` both do conventional string searches (forward and backward respectively)

`C-M-s` and `C-M-r` both do regular expression searches (forward and backward respectively)

`M-%` does a search/replace while `C-M-%` does a regular expression search/replace

Finding occurrences (occur, grep)

Use `M-x grep`, `rgrep`, `occur`…

See also interactive versions with helm-swoop, helm-occur, ag.el.

Go to definition

Put the cursor on any symbol and press `M-.` (`slime-edit-definition`) to go to its definition. Press `M-,` to come back.

Go to symbol, list symbols in current source

Use `C-u M-.` (`slime-edit-definition` with a prefix argument, also available as `M-- M-.`) to autocomplete the symbol and navigate to it. This command always asks for a symbol even if the cursor is on one. It works with any loaded definition. Here's a little demonstration video.

You can think of it as a `imenu` completion that always work for any Lisp symbol. Add in [Slime's fuzzy completion] for maximum powerness!

Crossreferencing: find who's calling, referencing, setting a symbol

Slime has nice cross-referencing facilities. For example, you can ask what calls a particular function, what expands a macro, or where a global variable is being used.

Results are presented in a new buffer, listing the places which reference a particular entity. From there, we can press Enter to go to the corresponding source line, or more interestingly we can recompile the place at point by pressing **C-c C-c** on that line. Likewise, **C-c C-k** will recompile all the references. This is useful when modifying macros, inline functions, or constants.

The bindings are the following (they are also shown in Slime's menu):

- **C-c C-w c** (`slime-who-calls`) callers of a function
- **C-c C-w m** (`slime-who-macroexpands`) places where a macro is expanded
- **C-c C-w r** (`slime-who-references`) global variable references
- **C-c C-w b** (`slime-who-bind`) global variable bindings
- **C-c C-w s** (`slime-who-sets`) global variable setters
- **C-c C-w a** (`slime-who-specializes`) methods specialized on a symbol

And when the `slime-asdf` contrib is enabled, **C-c C-w d** (`slime-who-depends-on`) lists dependent ASDF systems

And a general binding: **M-?** or **M-_** (`slime-edit-uses`) combines all of the above, it lists every kind of references.

Lisp Documentation in Emacs - Learning About Lisp Symbols

Argument lists

When you put the cursor on a function, SLIME will show its signature in the minibuffer.

Documentation lookup

The main shortcut to know is:

- **C-c C-d d** shows the symbols' documentation on a new window (same result as using `describe`).

Other bindings which may be useful:

- **C-c C-d f** describes a function
- **C-c C-d h** looks up the symbol documentation in CLHS by opening the web browser. But it works only on symbols, so there are two more bindings:
- **C-c C-d #** for reader macros
- **C-c C-d ~** for format directives

You can enhance the help buffer with the Slime extension slime-doc-contribs. It will show more information in a nice looking buffer.

Inspect

You can call `(inspect 'symbol)` from the REPL or call it with `C-c I` from a source file.

Macroexpand

Use `C-c M-m` to macroexpand a macro call

Consult the Hyper Spec (CLHS) offline

The Common Lisp Hyper Spec is the official online version of the ANSI Common Lisp standard. We can start browsing it from starting points: a shortened table of contents of highlights, a symbols index, a glossary, a master index.

Since January of 2023, we have the Common Lisp Community Spec: https://cl-community-spec.github.io/pages/index.html, a new web rendering of the specification. It is a more modern rendering:

- it has a *search box*
- it has *syntax highlihgting*
- it is hosted on GitHub and we have the right to modify it: https://github.com/fonol/cl-community-spec

If you want other tools to do a quick look-up of symbols on the CLHS, since the official website doesn't have a search bar, you can use: * Xach's website search utility: https://www.xach.com/clhs?q=with-open-file * the l1sp.org website: http://l1sp.org/search?q=with-open-file, * and we can use Duckduckgo's or Brave Search's `!clhs` "bang".

We can **browse the CLHS offline** with [Dash]() on MacOS, [Zeal]() on GNU/Linux and [Velocity]() on Windows.

But we can also browse it offline from Emacs. We have to install a CL package and to configure the Emacs side with one command:

```
(ql:quickload "clhs")
(clhs:install-clhs-use-local)
```

Then add this to your Emacs configuration:

```
(load "~/quicklisp/clhs-use-local.el" 'noerror)
```

Now, you can use `C-c C-d h` to look-up the symbol at point in the HyperSpec. This will open your browser, but look at its URL starting with "file://home/": it opens a local file.

Other commands are available:

- when you want to look-up a reader macro, such as `#'` (sharpsign-quote) or `(` (left-parenthesis), use `M-x common-lisp-hyperspec-lookup-reader-macro`, bound to `C-c C-d #`.
- to look-up a `format` directive, such as `~A`, use `M-x common-lisp-hyperspec-format`, bound to `C-c C-d ~`.
  - of course, you can TAB-complete on Emacs' minibuffer prompt to see all the available format directives.

- you can also look-up glossary terms (for example, you can look-up “function” instead of “defun”), use `M-x common-lisp-hyperspec-glossary-term`, bound to `C-c C-d g`.

Miscellaneous

Synchronizing packages

C-c ~ (`slime-sync-package-and-default-directory`): When run in a buffer with a lisp file it will change the current package of the REPL to the package of that file and also set the current directory of the REPL to the parent directory of the file.

Calling code

C-c C-y (`slime-call-defun`): When the point is inside a defun and C-c C-y is pressed,

(I’ll use [] as an indication where the cursor is)

```
(defun foo ()
 nil[])
```

then `(foo [])` will be inserted into the REPL, so that you can write additional arguments and run it.

If `foo` was in a different package than the package of the REPL, `(package:foo )` or `(package::foo )` will be inserted.

This feature is very useful for testing a function you just wrote.

That works not only for defun, but also for defgeneric, defmethod, defmacro, and define-compiler-macro in the same fashion as for defun.

For defvar, defparameter, defconstant: `[] *foo*` will be inserted (the cursor is positioned before the symbol so that you can easily wrap it into a function call).

For defclass: `(make-instance ‘class-name )`.

Inserting calls to frames in the debugger

C-y in SLDB on a frame will insert a call to that frame into the REPL, e.g.,

```
(/ 0) =>
…
1: (CCL::INTEGER-/-INTEGER 1 0)
…
```

C-y will insert `(CCL::INTEGER-/-INTEGER 1 0)`.

(thanks to Slime tips)

Exporting symbols

C-c x (*slime-export-symbol-at-point*) from the `slime-package-fu` contrib: takes the symbol at point and modifies the `:export` clause of the corresponding defpackage form. It also exports the symbol. When called with a negative argument (C-u C-c x) it will remove the symbol from `:export` and unexport it.

M-x slime-export-class does the same but with symbols defined by a structure or a class, like accessors, constructors, and so on. It works on structures only on SBCL and Clozure CL so far. Classes should work everywhere with MOP.

Customization

There are different styles of how symbols are presented in `defpackage`, the default is to use uninterned symbols (`#:foo`). This can be changed:

to use keywords:

```
(setq slime-export-symbol-representation-function
      (lambda (n) (format ":%s" n)))
```

or strings:

```
(setq slime-export-symbol-representation-function
 (lambda (n) (format "\"%s\"" (upcase n))))
```

Project Management

ASDF is the de-facto build facility. It is shipped in most Common Lisp implementations.

- ASDF
- ASDF best practices

Searching Quicklisp libraries

From the REPL, we can use `,ql` to install a package known by name already.

In addition, we can use the Quicklisp-systems Slime extension to search, browse and load Quicklisp systems from Emacs.

Questions/Answers

utf-8 encoding

You might want to set this to your init file:

```
(set-language-environment "UTF-8")
(setenv "LC_CTYPE" "en_US.UTF-8")
```

and for Sly:

```
(setf sly-lisp-implementations
      '((sbcl ("/usr/local/bin/sbcl") :coding-system utf-8-unix)
        ))
```

This will avoid getting `ascii stream decoding errors` when you have non-ascii characters in files you evaluate with SLIME.

Default cut/copy/paste keybindings

I am so used to C-c, C-v and friends to copy and paste text that the default Emacs shortcuts don't make any sense to me.

Luckily, you have a solution! Install [cua-mode] and you can continue to use these shortcuts.

```
;; C-z=Undo, C-c=Copy, C-x=Cut, C-v=Paste (needs cua.el)
(require 'cua) (CUA-mode t)
```

Appendix

All Slime REPL shortcuts

Here is the reference of all Slime shortcuts that work in the REPL.

To see them, go in a REPL, type `C-h m` and go to the Slime REPL map section.

```
REPL mode defined in ‘slime-repl.el’:
Major mode for interacting with a superior Lisp.
key             binding
                -

C-c             Prefix Command
C-j             slime-repl-newline-and-indent
RET             slime-repl-return
C-x             Prefix Command
ESC             Prefix Command
SPC             slime-space
  (that binding is currently shadowed by another mode)
,               slime-handle-repl-shortcut
DEL             backward-delete-char-untabify
<C-down>        slime-repl-forward-input
<C-return>      slime-repl-closing-return
<C-up>          slime-repl-backward-input
<return>        slime-repl-return

C-x C-e         slime-eval-last-expression

C-c C-c         slime-interrupt
C-c C-n         slime-repl-next-prompt
C-c C-o         slime-repl-clear-output
C-c C-p         slime-repl-previous-prompt
```

```
C-c C-s         slime-complete-form
C-c C-u         slime-repl-kill-input
C-c C-z         other-window
C-c ESC         Prefix Command
C-c I           slime-repl-inspect

M-RET           slime-repl-closing-return
M-n             slime-repl-next-input
M-p             slime-repl-previous-input
M-r             slime-repl-previous-matching-input
M-s             previous-line

C-c C-z         run-lisp
  (that binding is currently shadowed by another mode)

C-M-x           lisp-eval-defun

C-M-q           indent-sexp

C-M-q           prog-indent-sexp
  (that binding is currently shadowed by another mode)

C-c M-e         macrostep-expand
C-c M-i         slime-fuzzy-complete-symbol
C-c M-o         slime-repl-clear-buffer
```

All other Slime shortcuts

Here are all the default keybindings defined by Slime mode.

To see them, go in a .lisp file, type `C-h m` and go to the Slime section.

```
Commands to compile the current buffer’s source file and visually
highlight any resulting compiler notes and warnings:
C-c C-k - Compile and load the current buffer’s file.
C-c M-k - Compile (but not load) the current buffer’s file.
C-c C-c - Compile the top-level form at point.

Commands for visiting compiler notes:
M-n - Goto the next form with a compiler note.
M-p - Goto the previous form with a compiler note.
C-c M-c - Remove compiler-note annotations in buffer.

Finding definitions:
M-.
```

```
- Edit the definition of the function called at point.
M-,
- Pop the definition stack to go back from a definition.

Documentation commands:
C-c C-d C-d - Describe symbol.
C-c C-d C-a - Apropos search.
C-c M-d - Disassemble a function.

Evaluation commands:
C-M-x   - Evaluate top-level from containing point.
C-x C-e - Evaluate sexp before point.
C-c C-p - Evaluate sexp before point, pretty-print result.

Full set of commands:
key             binding
                -

C-c             Prefix Command
C-x             Prefix Command
ESC             Prefix Command
SPC             slime-space

C-c C-c         slime-compile-defun
C-c C-j         slime-eval-last-expression-in-repl
C-c C-k         slime-compile-and-load-file
C-c C-s         slime-complete-form
C-c C-y         slime-call-defun
C-c ESC         Prefix Command
C-c C-]         slime-close-all-parens-in-sexp
C-c x           slime-export-symbol-at-point
C-c ~           slime-sync-package-and-default-directory

C-M-a           slime-beginning-of-defun
C-M-e           slime-end-of-defun
M-n             slime-next-note
M-p             slime-previous-note

C-M-,           slime-previous-location
C-M-.           slime-next-location

C-c TAB         completion-at-point
C-c RET         slime-expand-1
C-c C-p         slime-pprint-eval-last-expression
C-c C-u         slime-undefine-function
C-c ESC         Prefix Command

C-c C-b         slime-interrupt
```

```
C-c C-d         slime-doc-map
C-c C-e         slime-interactive-eval
C-c C-l         slime-load-file
C-c C-r         slime-eval-region
C-c C-t         slime-toggle-fancy-trace
C-c C-v         Prefix Command
C-c C-w         slime-who-map
C-c C-x         Prefix Command
C-c C-z         slime-switch-to-output-buffer
C-c ESC         Prefix Command
C-c :           slime-interactive-eval
C-c <           slime-list-callers
C-c >           slime-list-callees
C-c E           slime-edit-value
C-c I           slime-inspect

C-x C-e         slime-eval-last-expression
C-x 4           Prefix Command
C-x 5           Prefix Command

C-M-x           slime-eval-defun
M-,             slime-pop-find-definition-stack
M-.             slime-edit-definition
M-?             slime-edit-uses
M-_             slime-edit-uses

C-c M-c         slime-remove-notes
C-c M-e         macrostep-expand
C-c M-i         slime-fuzzy-complete-symbol
C-c M-k         slime-compile-file
C-c M-q         slime-reindent-defun

C-c M-m         slime-macroexpand-all

C-c C-v C-d     slime-describe-presentation-at-point
C-c C-v TAB     slime-inspect-presentation-at-point
C-c C-v C-n     slime-next-presentation
C-c C-v C-p     slime-previous-presentation
C-c C-v C-r     slime-copy-presentation-at-point-to-repl
C-c C-v C-w     slime-copy-presentation-at-point-to-kill-ring
C-c C-v ESC     Prefix Command
C-c C-v SPC     slime-mark-presentation
C-c C-v d       slime-describe-presentation-at-point
C-c C-v i       slime-inspect-presentation-at-point
C-c C-v n       slime-next-presentation
C-c C-v p       slime-previous-presentation
C-c C-v r       slime-copy-presentation-at-point-to-repl
C-c C-v w       slime-copy-presentation-at-point-to-kill-ring
```

```
C-c C-v C-SPC   slime-mark-presentation

C-c C-w C-a     slime-who-specializes
C-c C-w C-b     slime-who-binds
C-c C-w C-c     slime-who-calls
C-c C-w RET     slime-who-macroexpands
C-c C-w C-r     slime-who-references
C-c C-w C-s     slime-who-sets
C-c C-w C-w     slime-calls-who
C-c C-w a       slime-who-specializes
C-c C-w b       slime-who-binds
C-c C-w c       slime-who-calls
C-c C-w d       slime-who-depends-on
C-c C-w m       slime-who-macroexpands
C-c C-w r       slime-who-references
C-c C-w s       slime-who-sets
C-c C-w w       slime-calls-who

C-c C-d C-a     slime-apropos
C-c C-d C-d     slime-describe-symbol
C-c C-d C-f     slime-describe-function
C-c C-d C-g     common-lisp-hyperspec-glossary-term
C-c C-d C-p     slime-apropos-package
C-c C-d C-z     slime-apropos-all
C-c C-d #       common-lisp-hyperspec-lookup-reader-macro
C-c C-d a       slime-apropos
C-c C-d d       slime-describe-symbol
C-c C-d f       slime-describe-function
C-c C-d g       common-lisp-hyperspec-glossary-term
C-c C-d h       slime-documentation-lookup
C-c C-d p       slime-apropos-package
C-c C-d z       slime-apropos-all
C-c C-d ~       common-lisp-hyperspec-format
C-c C-d C-#     common-lisp-hyperspec-lookup-reader-macro
C-c C-d C-~     common-lisp-hyperspec-format

C-c C-x c       slime-list-connections
C-c C-x n       slime-next-connection
C-c C-x p       slime-prev-connection
C-c C-x t       slime-list-threads

C-c M-d         slime-disassemble-symbol
C-c M-p         slime-repl-set-package

C-x 5 .         slime-edit-definition-other-frame

C-x 4 .         slime-edit-definition-other-window
```

```
C-c C-v M-o     slime-clear-presentations
```

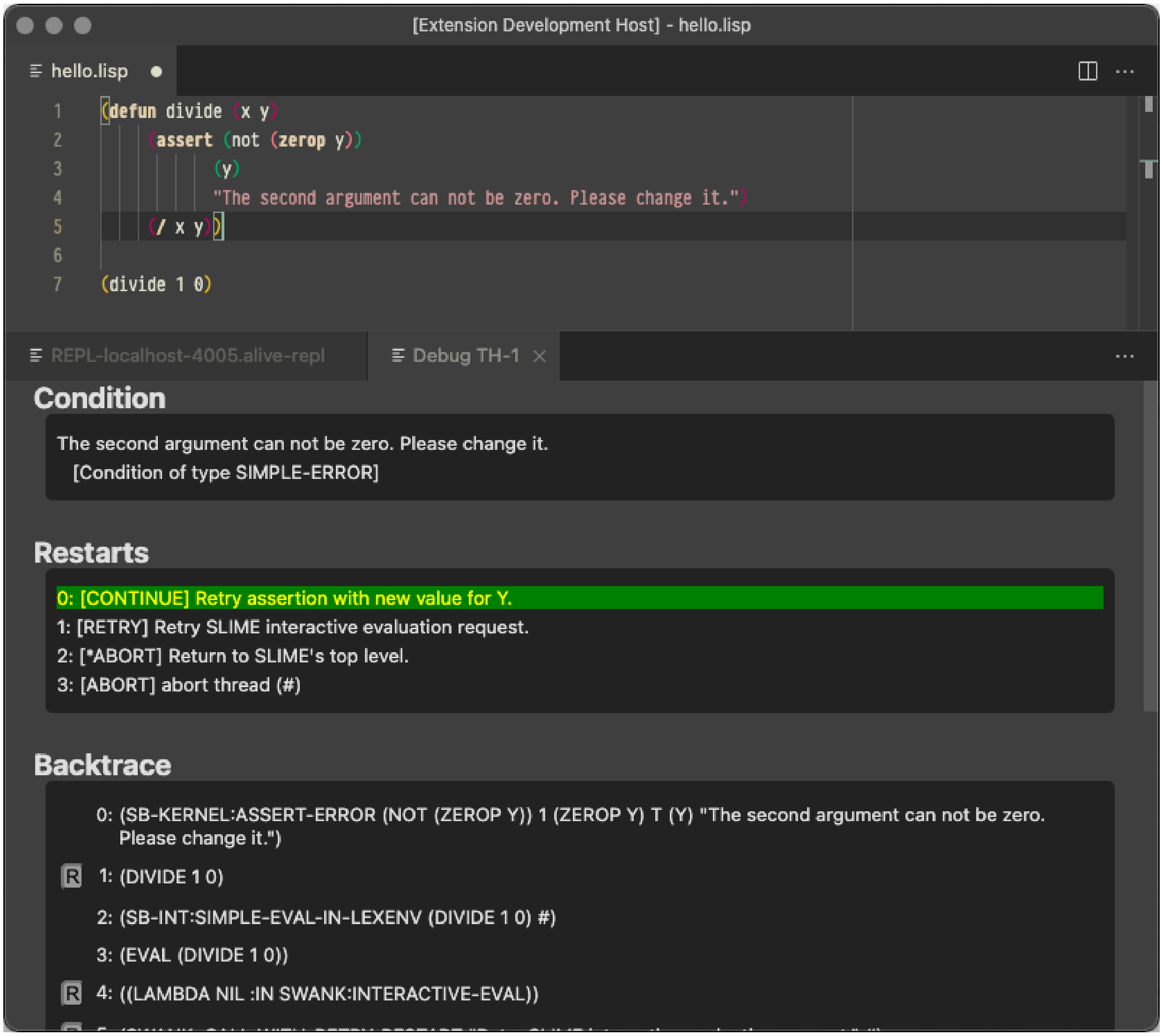
See also

- Common Lisp REPL exploration guide - a concise and curated set of highlights to find one's way in the REPL.

Using VSCode with Alive

The Alive extension makes VSCode a powerful Common Lisp development platform. Alive hooks directly into the Swank server that Emacs Slime uses and is fully compatible with VSCode's ability to develop remotely in containers, WSL, Remote machines, etc. It has no dependencies beyond a version of Common Lisp running on the target platform that can run the Swank server. It currently supports:

- Syntax highlighting
- Code completion
- Code formatter
- Jump to definition
- Snippets
- REPL integration
- Interactive Debugger
- REPL history
- Inline evaluation
- Macro expand
- Disassemble
- Inspector
- Hover Text
- Rename function args and let bindings
- Code folding

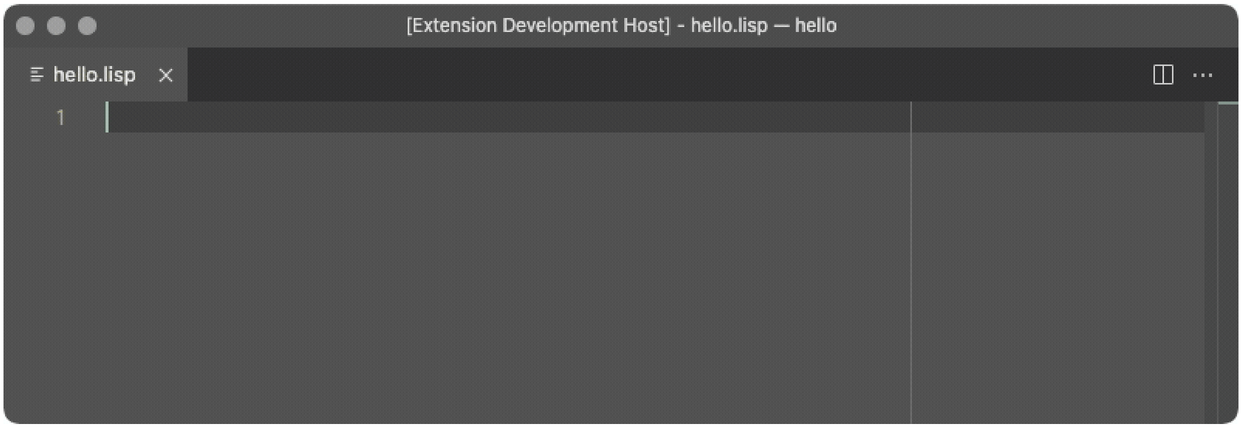


Prerequisites

The Alive extension in VSCode is compatible with ANSI Common Lisp, and these instructions should work for any of them as long as the Alive REPL starts up successfully. The examples all use SBCL.

- VsCode with command line installed running the Alive extension.
- SBCL

Connect VSCode to a REPL



1. Inside of VSCode, open a lisp file that you want to edit.
   - If you don't have one, create a new one called `hello.lisp`
2. Inside of VSCode, open the Command Palette on the menu at the top where it says `View/Command Palette` and start an instance of SBCL running a Swank server attached to your VSCode REPL by choosing: `Alive: Start REPL And Attach`.
   - You will see a small window pop up that says `REPL Connected`
   - If you don't get a `REPL Connected` message, open up VSCode's Output on the menu at the top where it says `View:Output` and choose `Swank Trace` from the pulldown. This output is the output from the running lisp image and will get you started on figuring out what might be going wrong.

Congrats, You now have a VSCode instance running a REPL attached to a Swank server running on port 4005 of a running SBCL image. You can now evaluate statements in your file and they will be processed in your running SBCL instance.

To disconnect your REPL and shut down your SBCL instance, open the Command Palette on the menu at the top where it says `View/Command Palette` and choose: `Alive: Detach from REPL`

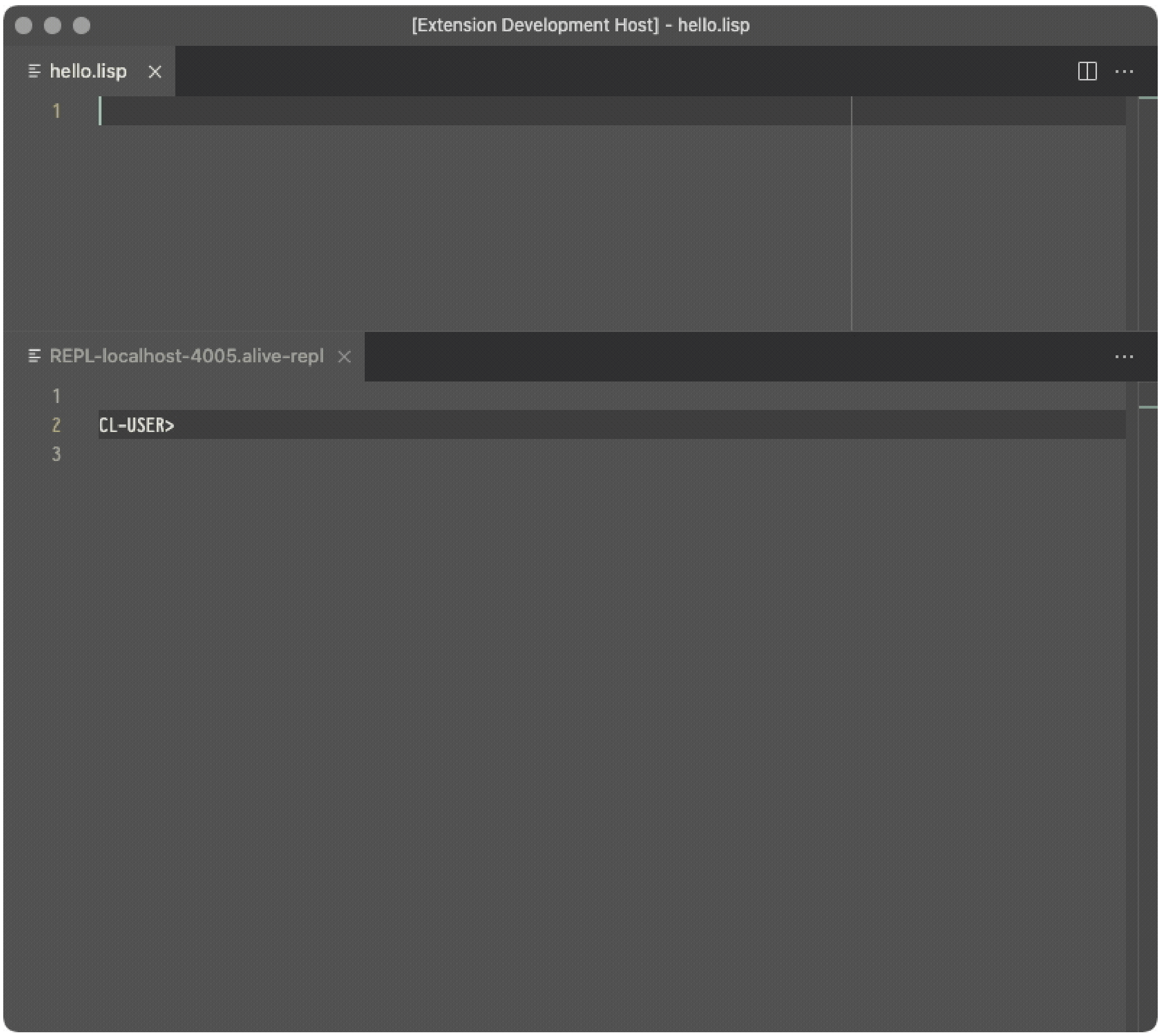
There are keybindings for every operation, feel free to explore and modify those as needed.

Recipes

All recipes assume you have a file open in VSCode running with an attached REPL unless otherwise stated.

When evaluating an expression, you choose the expression to evaluate by putting your cursor anywhere in or immediately following the s-expression that you wish to evaluate.

Evaluate a statement in-line



1. In your open file in your editor window, enter:

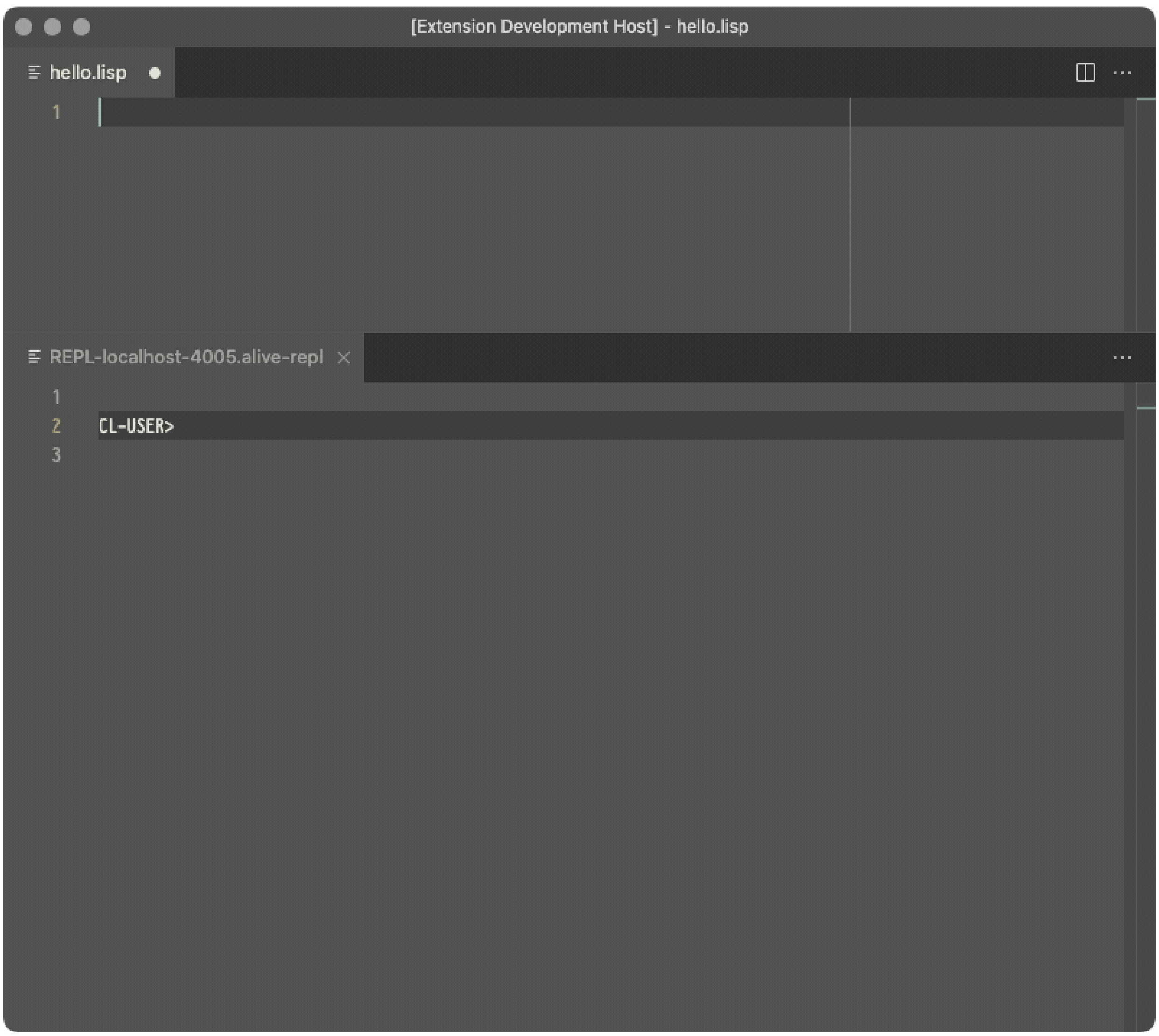
```
(+ 2 2)
```

2. Open the Command Palette on the menu at the top `View/Command Palette` and choose `Alive: Inline Eval`

3. You will see a small pop up that says `=> 4 (3 bits, #x4, #o4, #b100)`, which is the result

   Evaluating a statement in-line is exactly the same as sending it to the REPL. The only difference is how it is displayed.

Evaluate a statement



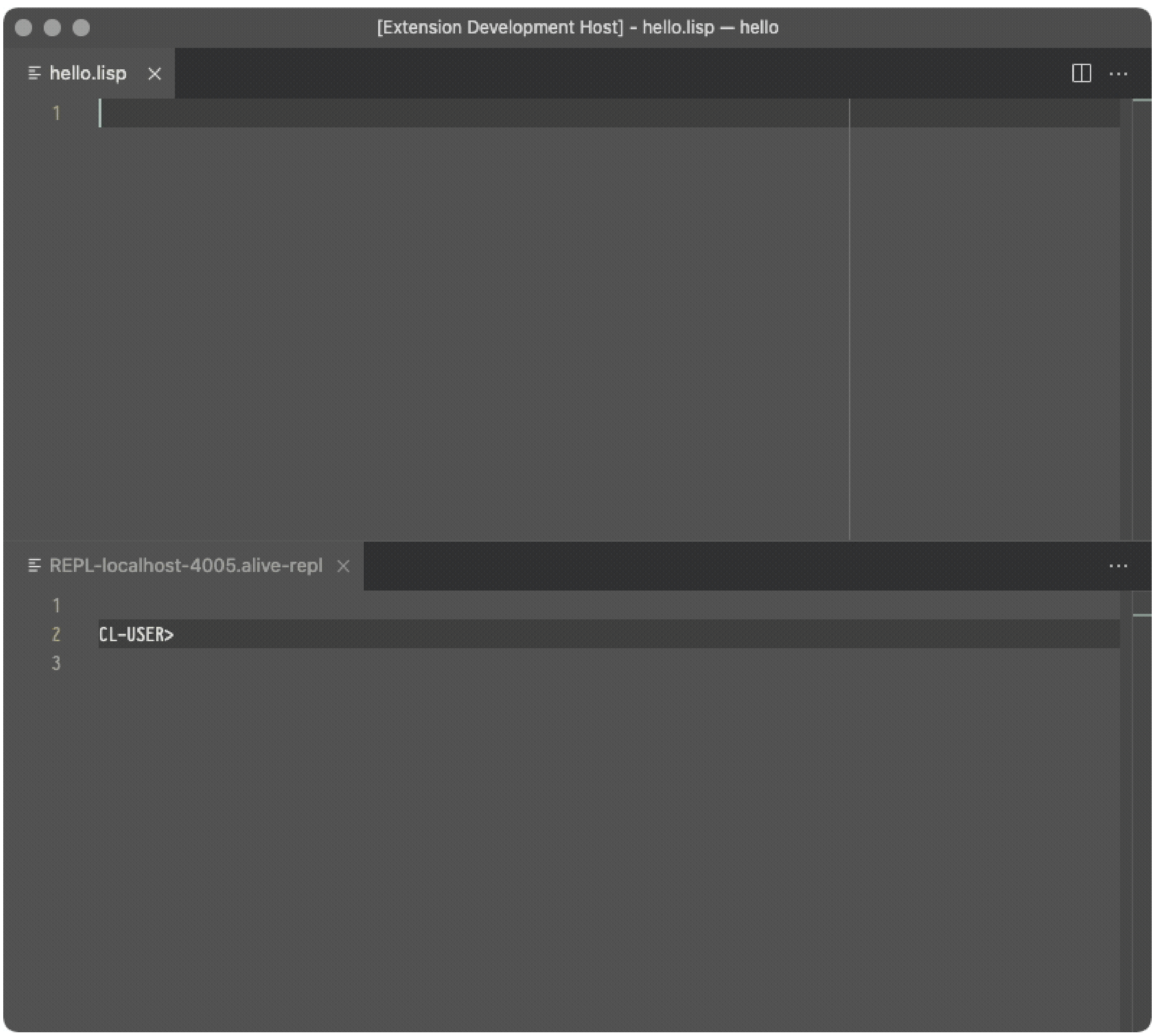
1. In your open file in your editor window, enter:

```
(+ 2 2)
```

2. Open the Command Palette on the menu at the top `View/Command Palette` and choose `Alive: Send To REPL`
3. You will see the expression show up in the REPL along with the result.

```
CL-USER>
(+ 2 2)
4
CL-USER>
```

Compile a file



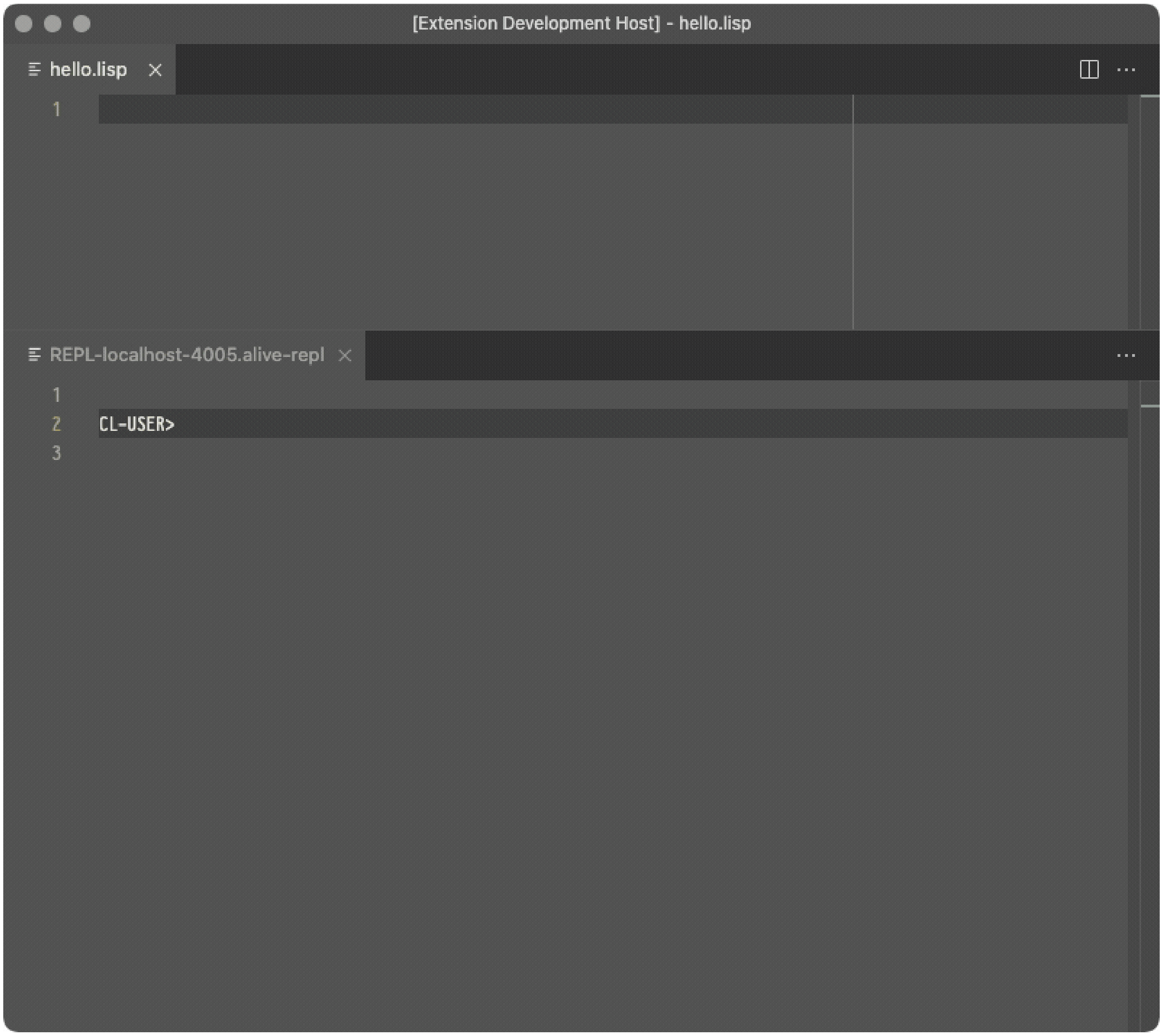
1. In your open file in your editor window, enter:

```
(+ 2 2)
```

2. Open the Command Palette on the menu at the top `View/Command Palette` and choose `Alive: Compile`
3. You will see details about the compile in your repl, and a fasl file in your filesystem.

```
CL-USER>

; compiling file "/Users/jason/Desktop/hello.lisp" (written 14 SEP 2021 04:24:37 AM):


; wrote /Users/jason/Desktop/hello.fasl

; compilation finished in 0:00:00.001
```

Use the Interactive Debugger to abort



1. In your open file in your editor window, enter:

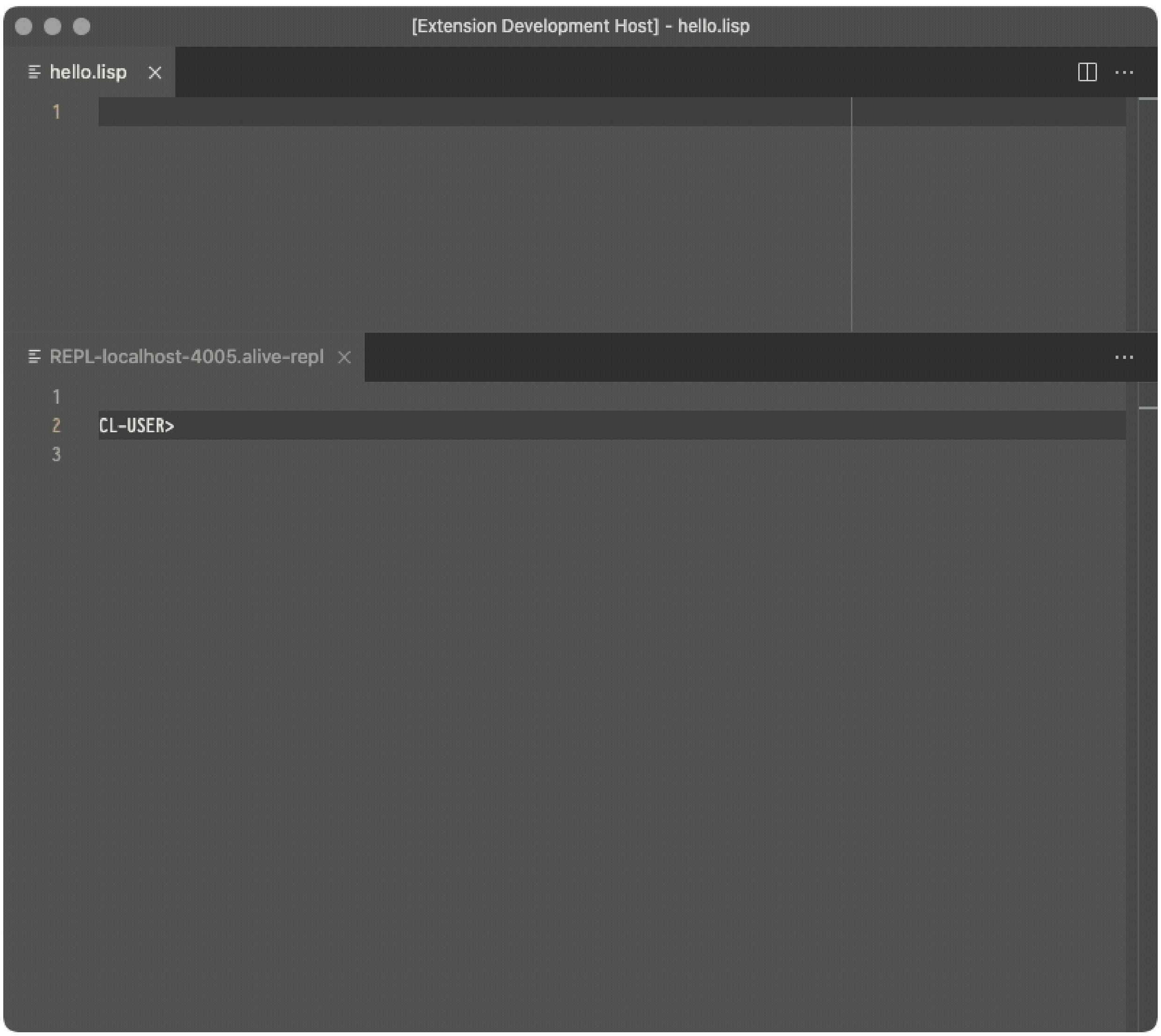
```
(defun divide (x y)
  (/ x y))
```

2. Put your cursor after the last parenthesis if it isn't already there. Open the Command Palette on the menu at the top `View/Command Palette` and choose `Alive: Inline Eval` to load your `define` function into your image.
3. In your open file, add another new line and enter:

```
(divide 1 0)
```

4. Put your cursor after the last parenthesis if it isn't already there. Open the Command Palette on the menu at the top `View/Command Palette` and choose `Alive: Inline Eval` to run your divide function in your image.
5. You will see the Interactive Debugger pop up. In the `Restarts` section, choose option 2 to Abort.
6. You're now back to your editor and still-running REPL and can continue like it never happened.

Use the Interactive Debugger to fix a problem at runtime



1. In your open file in your editor window, enter:

```lisp
(defun divide (x y)
  (assert (not (zerop y))
          (y)
          "The second argument can not be zero.")
  (/ x y))
```

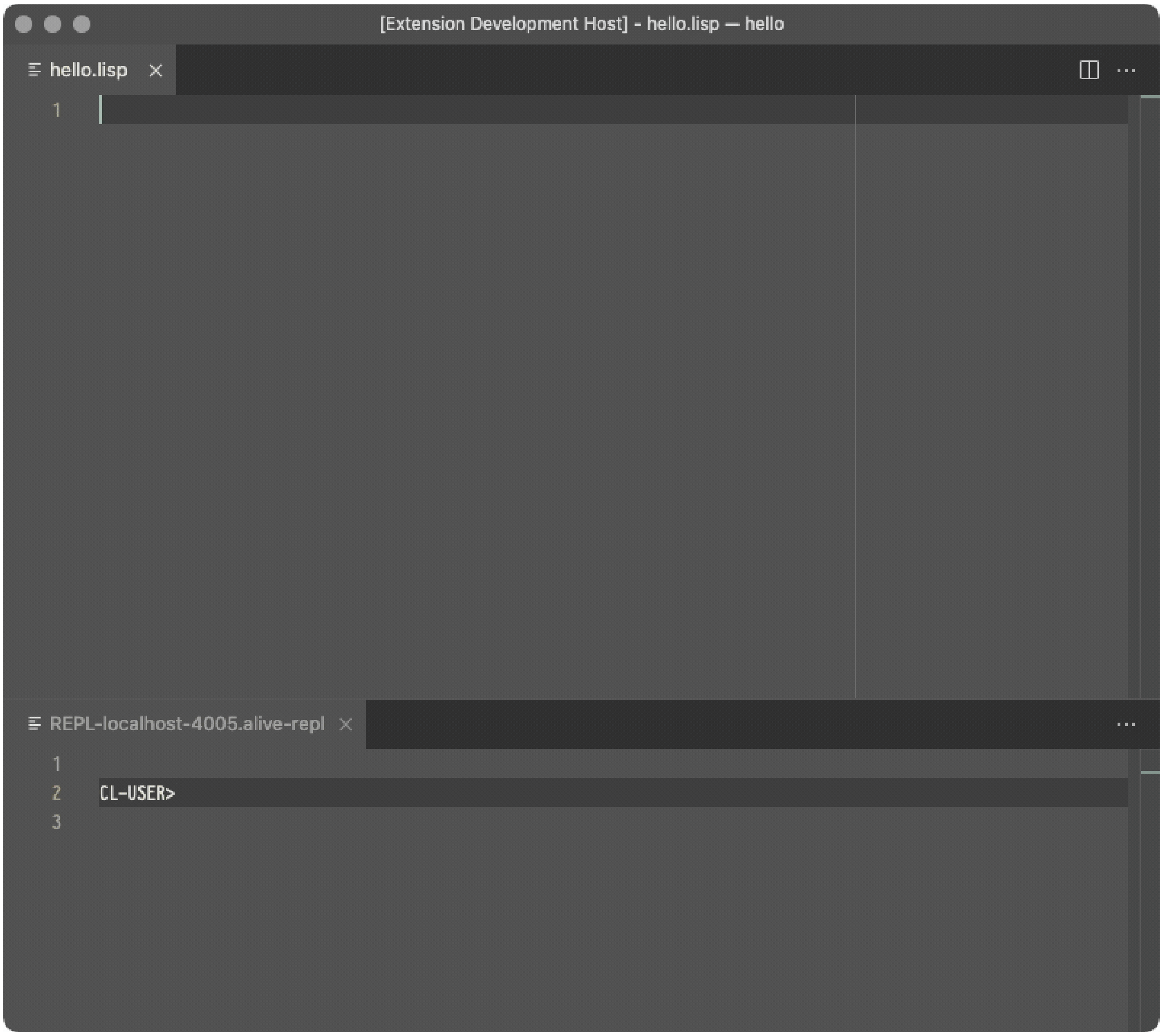
2. Put your cursor after the last parenthesis if it isn't already there. Open the Command Palette on the menu at the top `View/Command Palette` and choose `Alive: Inline Eval` to load your `define` function into your image.
3. In your open file, add another new line and enter:

```lisp
(divide 1 0)
```

4. Put your cursor after the last parenthesis if it isn't already there. Open the Command Palette on the menu at the top `View/Command Palette` and choose `Alive: Inline Eval` to run your divide function in your image.
5. You will see the Interactive Debugger pop up. In the `Restarts` section, choose option 0 to "Retry assertion with new value for Y".
6. In the popup menu, enter `y'
7. In the next popup menu, enter `1`
8. You should now see a small pop up that says `=> 1 (1 bit, #x1, #o1, #b1)`, which is the result of the new value. You're now back to your editor and still-running REPL after crashing out into the debugger, having it let you change the value that caused the crash, and then proceeding like you never typed that bad `0` value.

More ideas for what can be done with the debugger can be found on the error handling page.

Expand a macro



1. In your open file in your editor window, enter:

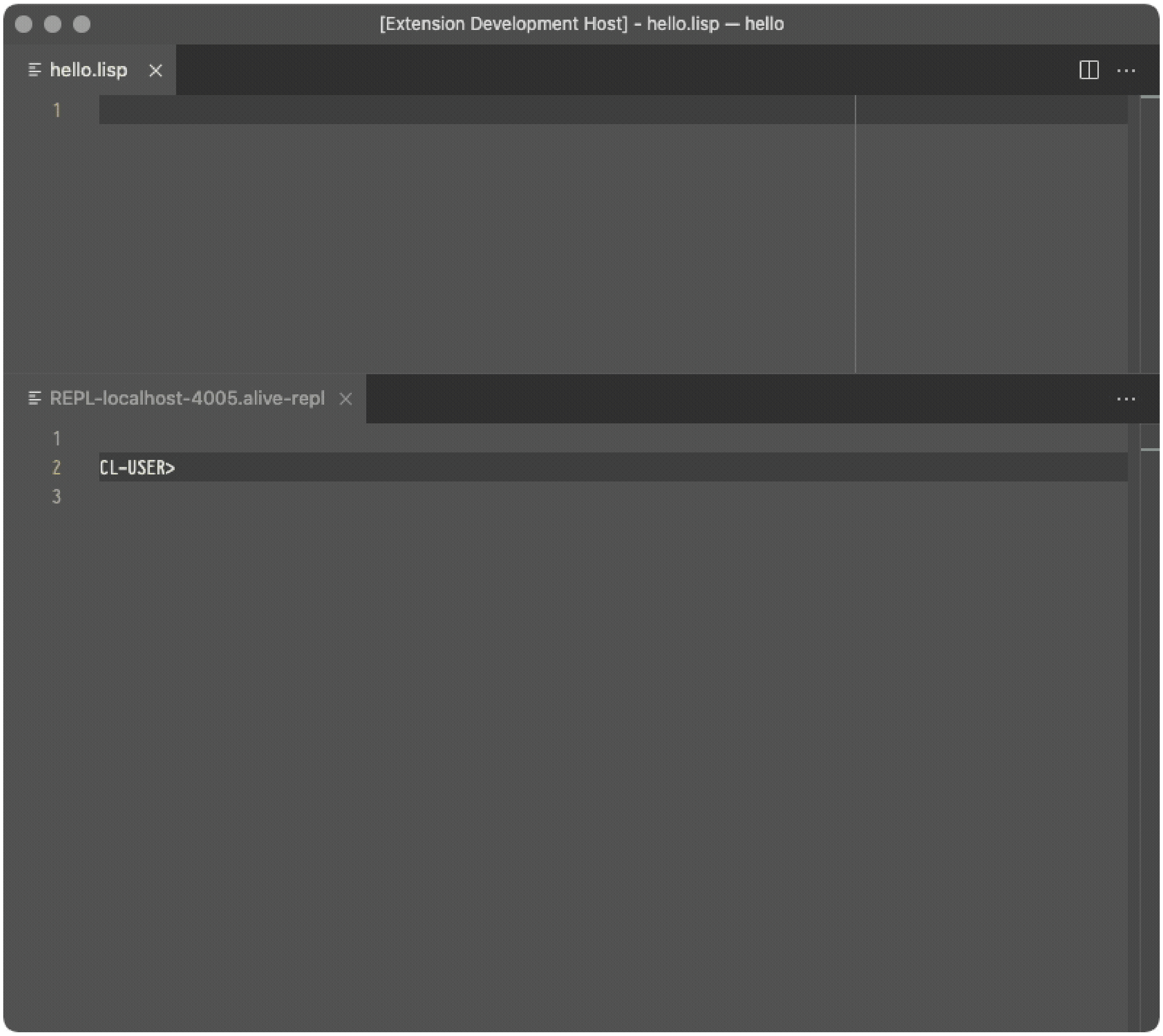
```
(loop for x in '(a b c d e) do
     (print x))
```

2. Put your cursor after the last parenthesis if it isn't already there. Open the Command Palette on the menu at the top `View/Command Palette` and choose `Alive: Macro Expand` to expand the for-loop macro.
3. You should see something like this:

```
(BLOCK NIL
  (LET ((X NIL)
        (#:LOOP-LIST-559
```

```
        (SB-KERNEL:THE* (LIST :USE-ANNOTATIONS T
                              :SOURCE-FORM '(A B C D E))
                        '(A B C D E))))
   (DECLARE (IGNORABLE #:LOOP-LIST-559)
            (IGNORABLE X))
   (TAGBODY
    SB-LOOP::NEXT-LOOP
     (SETQ X (CAR #:LOOP-LIST-559))
     (SETQ #:LOOP-LIST-559 (CDR #:LOOP-LIST-559))
     (PRINT X)
     (IF (ENDP #:LOOP-LIST-559)
         (GO SB-LOOP::END-LOOP))
     (GO SB-LOOP::NEXT-LOOP)
    SB-LOOP::END-LOOP)))
```

Disassemble a function



1. In your open file in your editor window, enter:

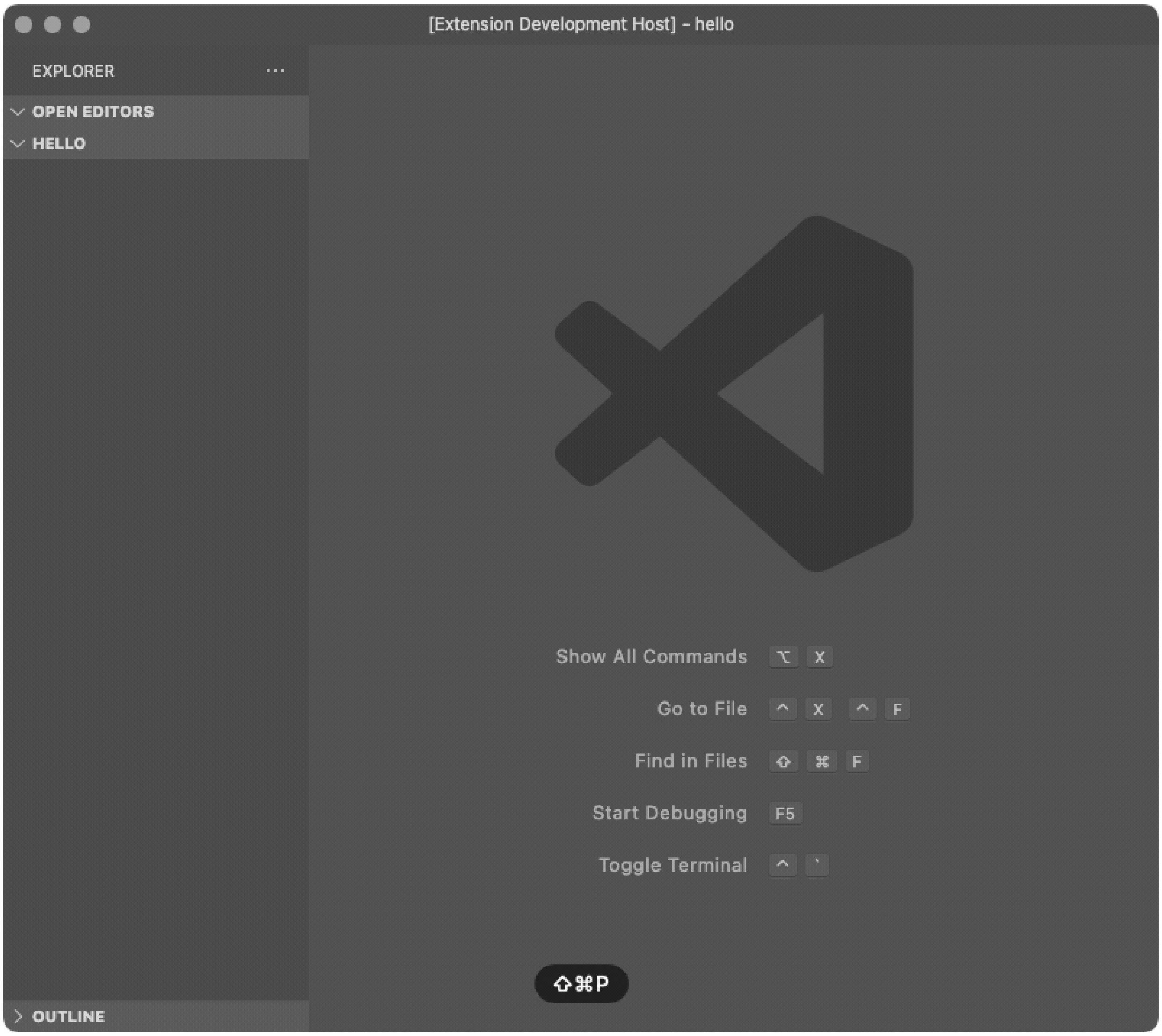
```
(defun hello (name)
  (format t "Hello, ~A~%" name))
```

2. Put your cursor after the last parenthesis if it isn't already there. Open the Command Palette on the menu at the top `View/Command Palette` and choose `Alive: Inline Eval` to load the function into your image.
3. Put your cursor after the last parenthesis if it isn't already there. Open the Command Palette on the menu at the top `View/Command Palette` and choose `Alive: Disassemble` print out the machine code of your compiled function.
4. It will start something like this:

```
; disassembly for HELLO
; Size: 172 bytes. Origin: #x70052478B4                        ;
HELLO
; 8B4:       AC0A40F9         LDR R2, [THREAD, #16]            ;
binding-stack-pointer
; 8B8:       4C0F00F9         STR R2, [CFP, #24]
; 8BC:       AC4642F9         LDR R2, [THREAD, #1160]          ;
tls: *STANDARD-OUTPUT*
; 8C0:       9F8501F1         CMP R2, #97
; 8C4:       61000054         BNE L0
; 8C8:       4AFDFF58         LDR R0, #x7005247870             ;
'*STANDARD-OUTPUT*
; 8CC:       4C1140F8         LDR R2, [R0, #1]
; 8D0: L0:   4C1700F9         STR R2, [CFP, #40]
; 8D4:       E0031BAA         MOV NL0, CSP
; 8D8:       7A0701F8         STR CFP, [CSP], #16
; 8DC:       EAFCFF58         LDR R0, #x7005247878             ;
"Hello, "
; 8E0:       4B1740F9         LDR R1, [CFP, #40]
; 8E4:       B6FBFF58         LDR LEXENV, #x7005247858         ;
#<SB-KERNEL:FDEFN WRITE-STRING>
; 8E8:       970080D2         MOVZ NARGS, #4
; 8EC:       FA0300AA         MOV CFP, NL0
; 8F0:       DE9240F8         LDR LR, [LEXENV, #9]
; 8F4:       C0033FD6         BLR LR
; 8F8:       3B039B9A         CSEL CSP, OCFP, CSP, EQ
; 8FC:       E0031BAA         MOV NL0, CSP
; 900:       7A0701F8         STR CFP, [CSP], #16
; 904:       4A2F42A9         LDP R0, R1, [CFP, #32]
; 908:       D6FAFF58         LDR LEXENV, #x7005247860         ;
#<SB-KERNEL:FDEFN PRINC>
; 90C:       970080D2         MOVZ NARGS, #4
; 910:       FA0300AA         MOV CFP, NL0
; 914:       DE9240F8         LDR LR, [LEXENV, #9]
; 918:       C0033FD6         BLR LR
; 91C:       3B039B9A         CSEL CSP, OCFP, CSP, EQ
; 920:       E0031BAA         MOV NL0, CSP
; 924:       7A0701F8         STR CFP, [CSP], #16
; 928:       2A4981D2         MOVZ R0, #2633
; 92C:       4B1740F9         LDR R1, [CFP, #40]
; 930:       D6F9FF58         LDR LEXENV, #x7005247868         ;
#<SB-KERNEL:FDEFN WRITE-CHAR>
; 934:       970080D2         MOVZ NARGS, #4
; 938:       FA0300AA         MOV CFP, NL0
; 93C:       DE9240F8         LDR LR, [LEXENV, #9]
; 940:       C0033FD6         BLR LR
; 944:       3B039B9A         CSEL CSP, OCFP, CSP, EQ
; 948:       EA031DAA         MOV R0, NULL
```

```
; 94C:       FB031AAA        MOV CSP, CFP
; 950:       5A7B40A9        LDP CFP, LR, [CFP]
; 954:       BF0300F1        CMP NULL, #0
; 958:       C0035FD6        RET
; 95C:       E00120D4        BRK #15                                   ;
Invalid argument count trap
```

Create a skeleton Common Lisp system



This recipe creates a new Common Lisp System, so it does not need a running REPL.

1. Create a folder called `experiment` for your new project

2. Open vscode in your newly created directory

```
cd experiment
code .
```

3. Create new Common Lisp System.

- Inside of VSCode, Open Command Palette on the menu at the top `View/Command Palette` and generate a system skeleton: `Alive: System Skeleton`
- The previous command should have generated the following directory structure:
  - experiment.asd
  - src/
    - app.lisp
  - test/
    - all.lisp

The content of those files is as follows:

`experiment.asd`:

```
(in-package :asdf-user)

(defsystem "experiment"
  :class :package-inferred-system
  :depends-on ("experiment/src/app")
  :description ""
  :in-order-to ((test-op (load-op "experiment/test/all")))
  :perform (test-op (o c) (symbol-call :test/all :test-suite)))

(defsystem "experiment/test"
  :depends-on ("experiment/test/all"))

(register-system-packages "experiment/src/app" '(:app))
(register-system-packages "experiment/test/all" '(:test/all))
```

`src/app.lisp`:

```lisp
(defpackage :app
  (:use :cl))

(in-package :app)
```

`test/all.lisp`:

```lisp
(defpackage :test/all
  (:use :cl
        :app)
  (:export :test-suite))

(in-package :test/all)

(defun test-suite ()
  (format T "Test Suite~%"))
```

Optional Custom Configurations

Configuring VSCode Alive to work with Quicklisp

Assuming that you have quicklisp installed and configured to load on init, quicklisp just works.

Configuring VSCode Alive to work with CLPM in the default context

Assuming that you have CLPM installed and configured, modify your vscode settings to look like this:

1. Add the following to to your VSCode settings:

```json
"alive.swank.startupCommand":[
  "clpm",
  "exec",
  "--",
  "sbcl",
  "--eval",
  "(asdf:load-system :swank)",
  "--eval",
```

```
    "(swank:create-server)"
  ],
```

This will start up sbcl in the default clpm context

Configuring VSCode Alive to work with CLPM using a bundle clpmfile

Assuming that you have [CLPM]() installed and configured and a bundle configured in the root of your home directory that contains swank as a dev dependency, [modify your vscode settings]() to look like this:

1. Add the following to your VSCode settings:

```
  "alive.swank.startupCommand":[
    "clpm",
    "bundle",
    "exec",
    "--",
    "sbcl",
    "--eval",
    "(asdf:load-system :swank)",
    "--eval",
    "(swank:create-server)"
  ],
```

This will start up sbcl in your bundle's clpm context

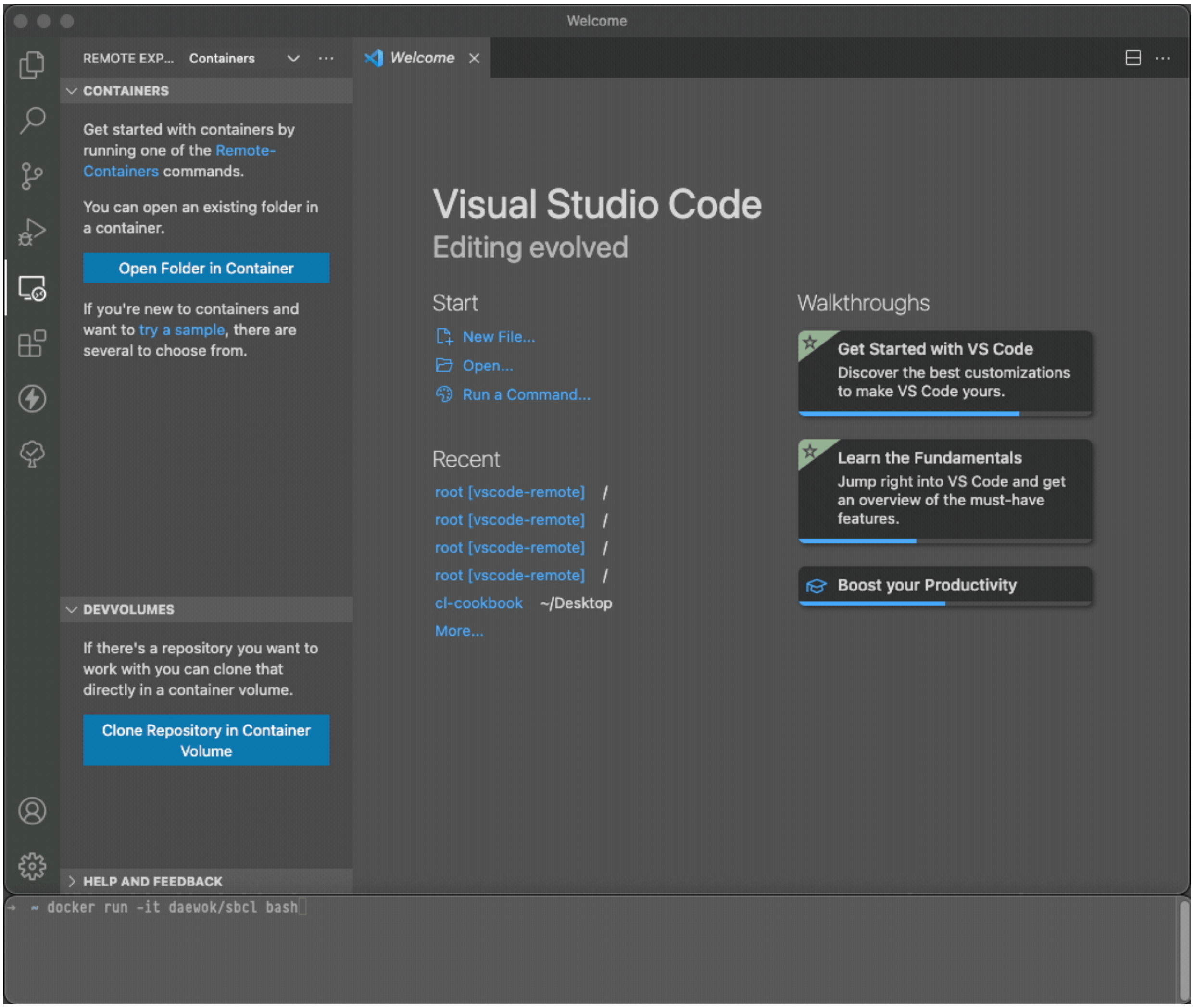
Configuring VSCode Alive to work with Roswell

Assuming that you have [Roswell]() installed, [modify your vscode settings]() to look like this:

```
  "alive.swank.startupCommand": [
    "ros",
    "run",
    "--eval",
    "(require :asdf)",
    "--eval",
    "(asdf:load-system :swank)",
    "--eval",
```

```
    "(swank:create-server)"
  ]
```

Connecting VSCode Alive to a Docker container



These instructions will work for remote connections, wsl connections, and github Codespaces as well using the `Remote - SSH` *and* `Remote - WSL`*, and* `Github Codespaces` *extensions, respectively assuming you have the extensions installed. For this example, make sure you have the [Containers extension installed and configured](#).*

1. Pull a docker image that has sbcl installed, in this example, we'll use the latest clfoundations sbcl.

```
docker pull clfoundation/sbcl
```

2. Run bash in the docker image to start it up and keep it running.

```
docker run -it clfoundation/sbcl bash
```

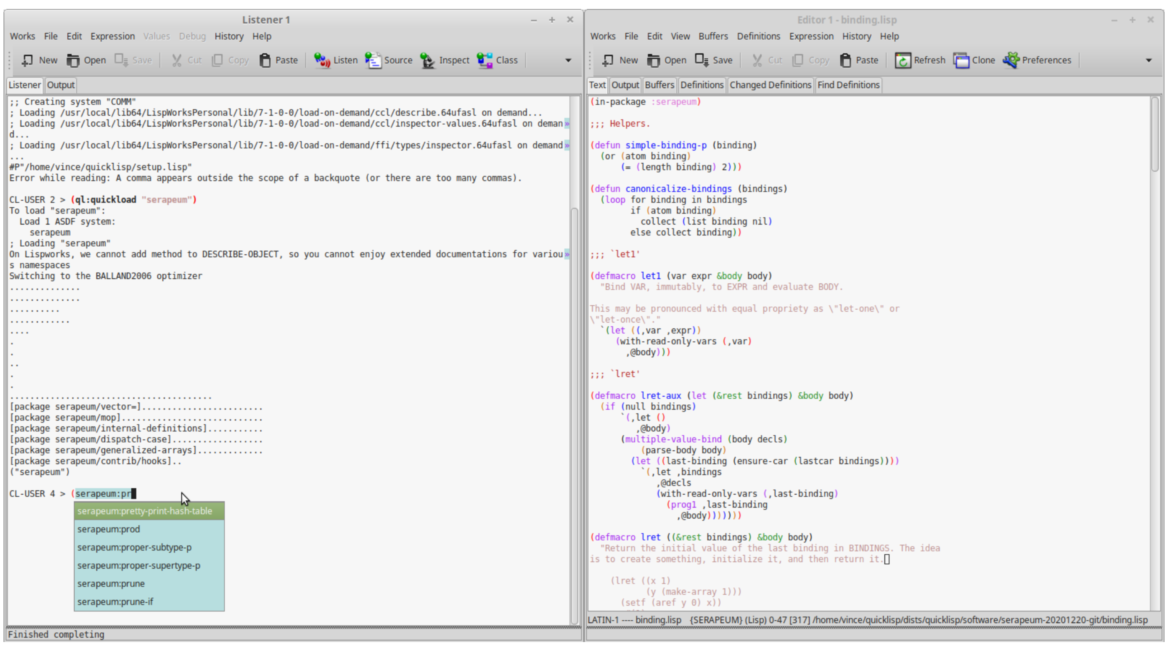
3. In the VSCode Side Bar, click the `Remote Explorer` icon.
4. In the list of Dev Containers, right click on clfoundation/sbcl and choose `Attach to Container`.
5. In the VSCode Side Bar of the new VSCode window that opens up, click on `Explorer`. *You may need to tell it to view the files in your container if it isn't already showing them.*
6. Once you're viewing the files in the container, right click in the VSCode `Side Bar` and choose `New File`. Name the file `hello.lisp`
7. In the VSCode Site Bar, click the `Extensions` icon
8. Click the `Install in Container...` button for the `Alive` plugin
9. Open up your `hello.lisp` file and follow the "Connect VSCode to a REPL" instructions at the beginning of these recipes
10. You now have VSCode running a REPL hooked to a Slime server running on an SBCL image in a docker container.

LispWorks review

[LispWorks](http://www.lispworks.com/) is a Common Lisp implementation that comes with its own Integrated Development Environment (IDE) and its share of unique features, such as the CAPI GUI toolkit. It is **proprietary** and provides a **free limited version**.

Here, we will mainly explore its IDE, asking ourselves what it can offer to a seasoned lisper used to Emacs and Slime. The short answer is: more graphical tools, such as an easy to use graphical stepper, a tracer, a code coverage browser or again a class browser. Setting and using breakpoints was easier than on Slime.

LispWorks also provides more integrated tools (the Process browser lists all processes running in the Lisp image and we can stop, break or debug them) and presents many information in the form of graphs (for example, a graph of function calls or a graph of all the created windows).



LispWorks’ listener and editor in the Mate desktop environment

LispWorks features

We can see a matrix of LispWorks features by edition and platform here: http://www.lispworks.com/products/features.html.

We highlight:

- 32-bit, 64-bit and ARM support on Windows, MacOS, Linux, Solaris, FreeBSD,
- CAPI portable GUI toolkit: provides native look-and-feel on Windows, Cocoa, GTK+ and Motif.
  - comes with a graphical “Interface Builder” (think QtCreator) (though not available on MacOS (nor on mobile))
- LispWorks for mobile runtime, for Android and iOS,
- optimized application delivery: LispWorks can use a tree shaker to remove unused lisp code from the delivered applicatiion, thus shipping lighter binaries than existing open-source implementations.
- ability to deliver a dynamic library,
- a Java interface, allowing to call from Lisp to Java or the other way around,
- an Objective-C and Cocoa interface, with drag and drop and multi-touch support,
- a Foreign Language Interface,
- TCP/UDP sockets with SSL & IPv6 support,
- natived threads and symmetric multiprocessing, unicode support, and all other Common Lisp features, and all other LispWorks Enterprise features.

And, of course, a built-in IDE.

LispWorks is used in diverse areas of the industry. They maintain a list of success stories. As for software that we can use ourselves, we find ScoreCloud amazing (a music notation software: you play an instrument, sing or whistle and it writes the music) or OpenMusic (opensource composition environment).

Free edition limitations

The download instructions and the limitations are given on the download page.

The limitations are the following:

- There is a **heap size limit** which, if exceeded, causes the image to exit. A warning is provided when the limit is approached.

What does it prevent us to do? As an illustration, we can not load this set of libraries together in the same image:

```lisp
(ql:quickload '("alexandria" "serapeum" "bordeaux-threads"
    "lparallel" "dexador" "hunchentoot" "quri"
    "cl-ppcre" "mito"))
```

- There is a **time limit of 5 hours** for each session, after which LispWorks Personal exits, possibly without saving your work or performing cleanups such as removing temporary files. You are warned after 4 hours of use.

- It is **impossible to build a binary**. Indeed, the functions save-image, deliver (*the* function to create a stand-alone executable), and load-all-patches are not available.

- **Initialization files are not loaded**. If you are used to initializing Quicklisp from your `~/.sbclrc` on Emacs, you'll have to load an init file manually every time you start LispWorks (`(load #p"~/.your-init-file`)).

For the record, the snippet provided by Quicklisp to put in one's startup file is the following:

```lisp
;; provided you installed quicklisp in ~/quicklisp/
(let ((quicklisp-init (merge-pathnames "quicklisp/setup.lisp"
                                       (user-homedir-pathname))))
  (when (probe-file quicklisp-init)
    (load quicklisp-init)))
```

You'll have to paste it to the listener window (with the `C-y` key, y as "yank").

- Layered products that are part of LispWorks Professional and Enterprise Editions (CLIM, KnowledgeWorks, Common SQL and LispWorks ORB) are not included. But **we can try the CAPI toolkit**.

The installation process requires you to fill an HTML form to receive a download link, then to run a first script that makes you accept the terms and the licence, then to run a second script that installs the software.

Licencing model

LispWorks actually comes in four paid editions. It's all explained by themselves here: http://www.lispworks.com/products/lispworks.html. In short, there is:

- a Hobbyist edition with `save-image` and `load-all-patches`, to apply updates of minor versions, without the obvious limitations, for non-

commercial and non-academic use,
- a HobbyistDV edition with the `deliver` function to create executables (still for non-commercial and non-academic uses),
- a Professional edition, with the `deliver` function, for commercial and academic uses,
- an Enterprise one, with their enterprise modules: the Common SQL interface, LispWorks ORB, KnowledgeWorks.

At the time of writing, the licence of the hobbyist edition costs 750 USD, the pro version the double. They are bought for a LW version, per platform. They have no limit of time.

NB: Please double check their upstream resources and don't hesitate to contact them.

LispWorks IDE

The LispWorks IDE is self-contained, but it is also possible to use LispWorks-the-implementation from Emacs and Slime (see below). The IDE runs *inside* the Common Lisp image, unlike Emacs which is an external program that communicates with the Lisp image through Swank and Slime. User code runs in the same process.

The editor

The editor offers what's expected: a TAB-completion pop-up, syntax highlighting, Emacs-like keybindings (including the `M-x` extended command). The menus help the discovery.

We personally found the editing experience a bit "raw". For example:

- indention after a new line is not automatic, one has to press TAB again.
- the auto-completion is not fuzzy.
- there are no plugins similar to ~~Paredit~~ (there is a brand new (2021) [Paredit for LispWorks]) or Lispy, nor a Vim layer.

We also had an issue, in that the go-to-source function bound to `M-.` did not work out for built-in Lisp symbols. Apparently, LispWorks doesn't provide much source code, and mostly code of the editor. Some other commercial Lisps, like Allegro CL, provide more source code

The editor provides an interesting tab: Changed Definitions. It lists the functions and methods that were redefined since, at our choosing: the first edit of the session, the last save, the last compile.

See also:

- the Editor User Guide.

Keybindings

Most of the keybindings are similar to Emacs, but not all. Here are some differences:

- to **compile a function**, use `C-S-c` (control, shift and c), instead of C-c C-c.
- to **compile the current buffer**, use `C-S-b` (instead of C-c C-k).

Similar ones include:

- `C-g` to cancel what you're doing,
- `C-x C-s` to save the current buffer,
- `M-w` and `C-y` to copy and paste,
- `M-b`, `M-f`, `C-a`, `C-e`… to move around words, to go to the beginning or the end of the line,
- `C-k` to kill until the end of the line, `C-w` to kill a selected region,
- `M-.` to find the source of a symbol,
- `C-x C-e` to evaluate the current defun,
- …

Some useful functions don't have a keybinding by default, for example:

- clear the REPL with `M-x Clear Listener`
- `Backward Kill Line`

It is possible to use **classical keybindings**, à la KDE/Gnome. Go to the Preferences menu, Environment and in the Emulation tab.

There is **no Vim layer**.

Searching keybindings by name

It is possible to search for a keybinding associated to a function, or a function name from its keybinding, with the menu (Help -> Editing -> Key to Command / Command to Key) or with `C-h` followed by a key, as in Emacs. For example type `C-h k` then enter a keybinding to get the command name. See more with `C-h ?`.

Tweaking the IDE

It is possible to change keybindings. The editor’s state is accessible from the `editor` package, and the editor is built with the CAPI framework, so we can use the `capi` interface too. Useful functions include:

`

```
editor:bind-key
editor:defcommand
editor:current-point
editor:with-point  ;; save point location
editor:move-point
editor:*buffer-list*
editor:*in-listener* ;; returns T when we are in the REPL
```

…

Here’s how you can bind keys:

```
;; Indent new lines.
;; By default, the point is not indented after a Return.
(editor:bind-key "Indent New Line" #\Return :mode "Lisp")

;; Insert pairs.
(editor:bind-key "Insert Parentheses For Selection" #\( :mode "Lisp")
(editor:bind-key "Insert Double Quotes For Selection"
   #\"
  :mode "Lisp")
```

Here’s how to define a new command. We make the `)` key to go past the next closing parenthesis.

```
(editor:defcommand "Move Over ()" (p)
  "Move past the next close parenthesis.
Any indentation preceeding the parenthesis is deleted."
  "Move past the next close parenthesis."
  ;; thanks to Thomas Hermann
  ;; https://github.com/ThomasHermann/LispWorks/blob/master/editor.li
  (declare (ignore p))
```

```
  (let ((point (editor:current-point)))
    (editor:with-point ((m point))
      (cond ((editor::forward-up-list m)
         (editor:move-point point m)
             (editor::point-before point)
             (loop (editor:with-point ((back point))
                     (editor::back-to-indentation back)
                     (unless (editor:point= back point)
                       (return)))
                   (editor::delete-indentation point))
         (editor::point-after point))
        (t (editor:editor-error))))))

(editor:bind-key "Move Over ()" #\) :mode "Lisp")
```

And here's how you can change indentation for special forms:

```
(editor:setup-indent "if" 1 4 1)
```

See also:

- a list of LispWork keybindings: https://www.nicklevine.org/declarative/lectures/additional/key-binds.html

The listener

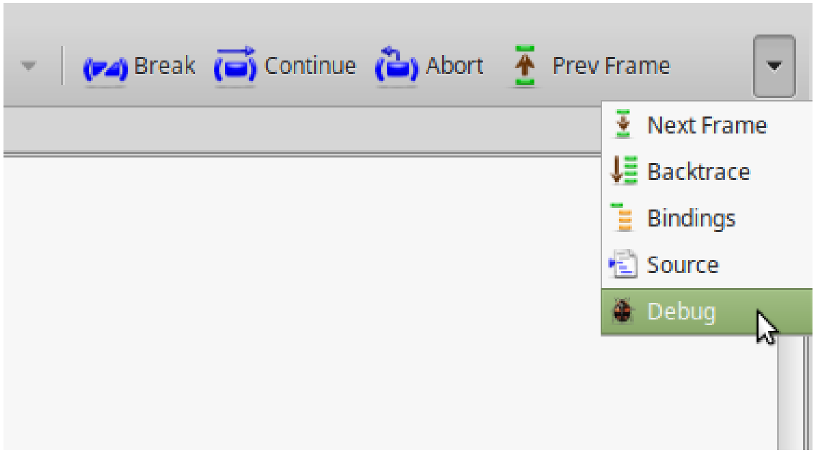
The listener is the REPL we are expecting to find, but it has a slight difference from Slime.

It doesn't evaluate the input line by line or form by form, instead it parses the input while typing. So we get some errors instantly. For example, we type `(abc`. So far so good. Once we type a colon to get `(abc:`, an error message is printed just above our input:

```
Error while reading: Reader cannot find package ABC.

CL-USER 1 > (abc:
```

Indeed, now `abc:` references a package, but such a package doesn't exist.

Its interactive debugger is primarily textual but you can also interact with it with graphical elements. For example, you can use the Abort button of the menu bar, which brings you back to the top level. You can invoke the graphical debugger to see the stacktraces and interact with them. See the Debugger button at the very end of the toolbar.



If you see the name of your function in the stacktraces (you will if you wrote and compiled your code in a file, and not directly wrote it in the REPL), you can double-click on its name to go back to the editor and have it highlight the part of your code that triggered the error.

NB: this is equivalent of pressing `M-v` in Slime.

It is possible to choose the graphical debugger to appear by default, instead of the textual one.

The listener provides some helper commands, not unlike Slime's ones starting with a comma `,`:

```
CL-USER 1 > :help

:bug-form <subject> &key <filename>
         Print out a bug report form, optionally to a file.
:get <variable> <command identifier>
               Get a previous command (found by its number or a symbol/subform within it) and put it in a variable.
:help    Produce this list.
:his &optional <n1> <n2>
          List the command history, optionally the last n1 or range n1 to n2.
:redo &optional <command identifier>
              Redo a previous command, found by its number or a symbol/subform within it.
:use <new> <old> &optional <command identifier>
                Do variant of a previous command, replacing old symbol/subform with new symbol/subform.
```

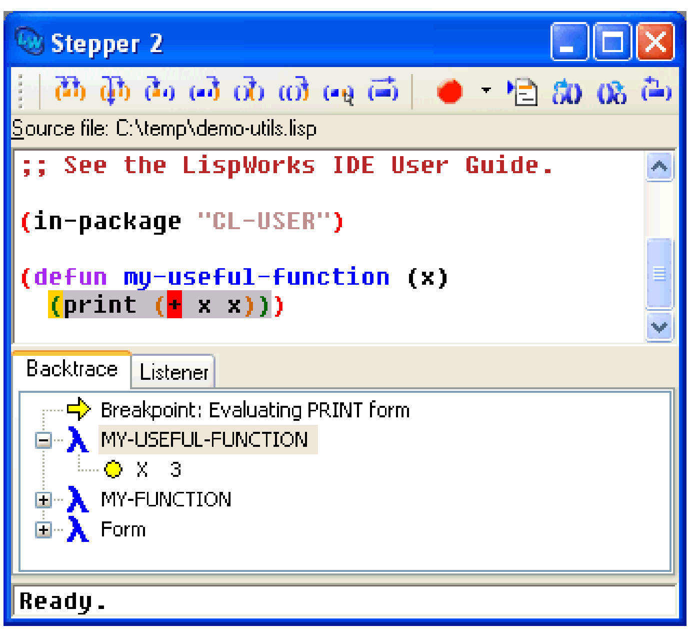
The stepper. Breakpoints.

The stepper is one of the areas where LispWorks shines.

When your are writing code in the editor window, you can set breakpoints with the big red “Breakpoint” button (or by calling `M-x Stepper Breakpoint`). This puts a red mark in your code.

The next time your code is executed, you’ll get a comprehensive Stepper pop-up window showing:

- your source code, with an indicator showing what expression is being evaluated
- a lower pane with two tabs:
  - the backtrace, showing the intermediate variables, thus showing their evolution during the execution of the program
  - the listener, in the context of this function call, where you can evaluate expressions
- and the menu bar with the stepper controls: you can step into the next expression, step on the next function call, continue execution until the position of the cursor, continue the execution until the next breakpoint, etc.



That’s not all. The non-visual, REPL-oriented stepper is also nice. It shows the forms that are being evaluated and their results.

In this example, we use `:s` to "step" though the current form and its subforms. We are using the usual listener, we can write any Lisp code after the prompt (the little -> here), and we have access to the local variables (X).

```
CL-USER 4 > (defun my-abs (x)
              (cond ((> x 0) x) ((< x 0) (- x)) (t 0)))
CL-USER 5 > (step (my-abs -5))
(MY-ABS -5) -> :s
   -5 -> :s
   -5
   (COND ((> X 0) X) ((< X 0) (- X)) (T 0)) <=> (IF (> X 0) (PROGN X)
   ;; Access to the local variables:
   (IF (> X 0) (PROGN X) (IF (< X 0) (- X) (PROGN 0))) -> (format t '
Is X equal to -5? yes
   (IF (> X 0) (PROGN X) (IF (< X 0) (- X) (PROGN 0))) -> :s
      (> X 0) -> :s
         X -> :s
         -5
         0 -> :s
         0
      NIL
      (IF (< X 0) (- X) (PROGN 0)) -> :s
         (< X 0) -> :s
            X -> :s
            -5
            0 -> :s
            0
         T
         (- X) -> :s
            X -> :s
            -5
         5
      5
   5
5
```

Here are the available stepper commands (see `:?`):

```
:s          Step this form and all of its subforms (optional +ve
integer arg)
:st      Step this form without stepping its subforms
:si       Step this form without stepping its arguments if it is a
function call
:su      Step up out of this form without stepping its subforms
:sr      Return a value to use for this form
```

```
:sq       Quit from the current stepper level
:bug-form <subject> &key <filename>
          Print out a bug report form, optionally to a file.
:get <variable> <command identifier>
                Get a previous command (found by its number or a
symbol/subform within it) and put it in a variable.
:help     Produce this list.
:his &optional <n1> <n2>
           List the command history, optionally the last n1 or range
n1 to n2.
:redo &optional <command identifier>
               Redo a previous command, found by its number or a
symbol/subform within it.
:use <new> <old> &optional <command identifier>
                Do variant of a previous command, replacing old
symbol/subform with new symbol/subform.
```

The class browser

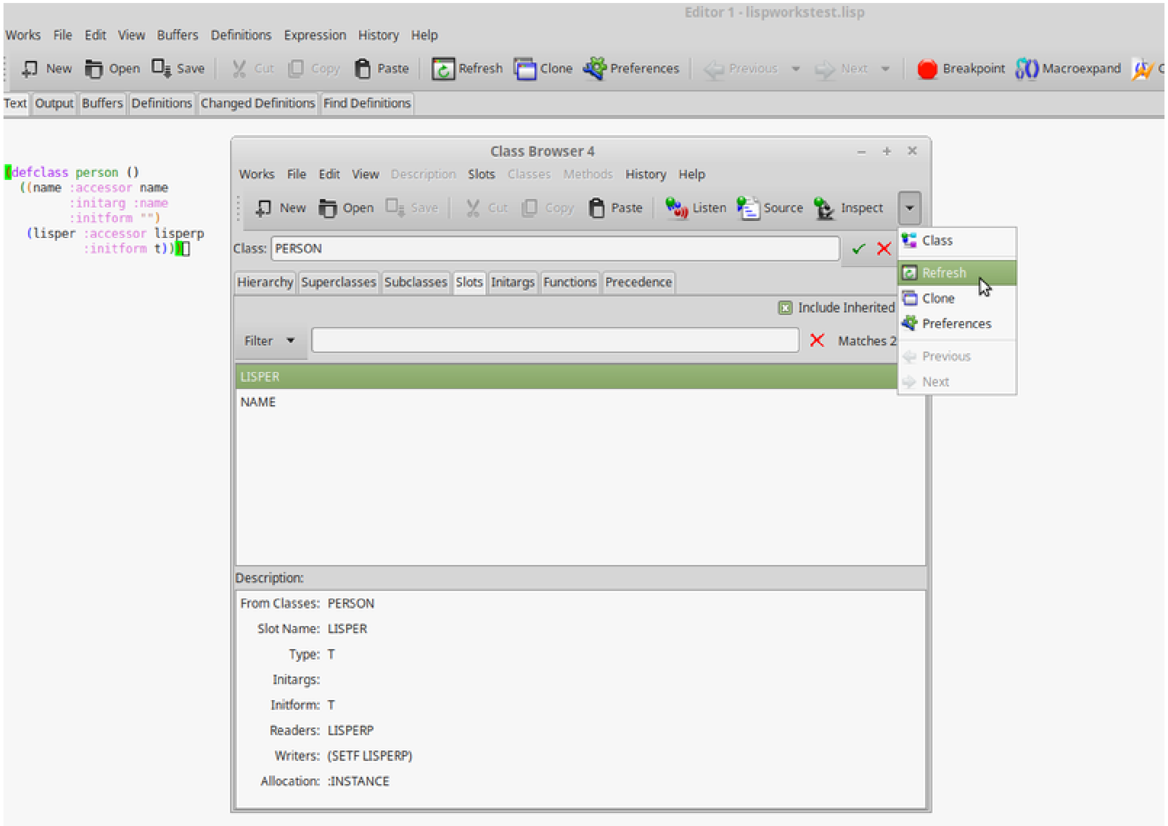
The class browser allows us to examine a class’s slots, parent classes, available methods, and some more.

Let’s create a simple class:

```
(defclass person ()
  ((name :accessor name
         :initarg :name
         :initform "")
   (lisper :accessor lisperp
           :initform t)))
```

Now call the class browser:

- use the “Class” button from the listener,
- or use the menu Expression -> Class,
- or put the cursor on the class and call `M-x Describe class`.



It is composed of several panes:

- the **class hierarchy**, showing the superclasses on the left and the subclasses on the right, with their description to the bottom,
- the **superclasses viewer**, in the form of a simple schema, and the same for subclasses,
- the **slots pane** (the default),
- the available **initargs**,
- the existing **generic functions** for that class
- and the **class precedence list**.

The Functions pane lists all methods applicable to that class, so we can discover public methods provided by the CLOS object system: `initialize-instance`, `print-object`, `shared-initialize`, etc. We can double-click on them to go to their source. We can choose not to include the inherited methods too (see the “include inherited” checkbox).

You’ll find buttons on the toolbar (for example, Inspect a generic function) and more actions on the Methods menu, such as a way to see the **functions calls**, a menu to **undefine** or **trace** a function.

See more:

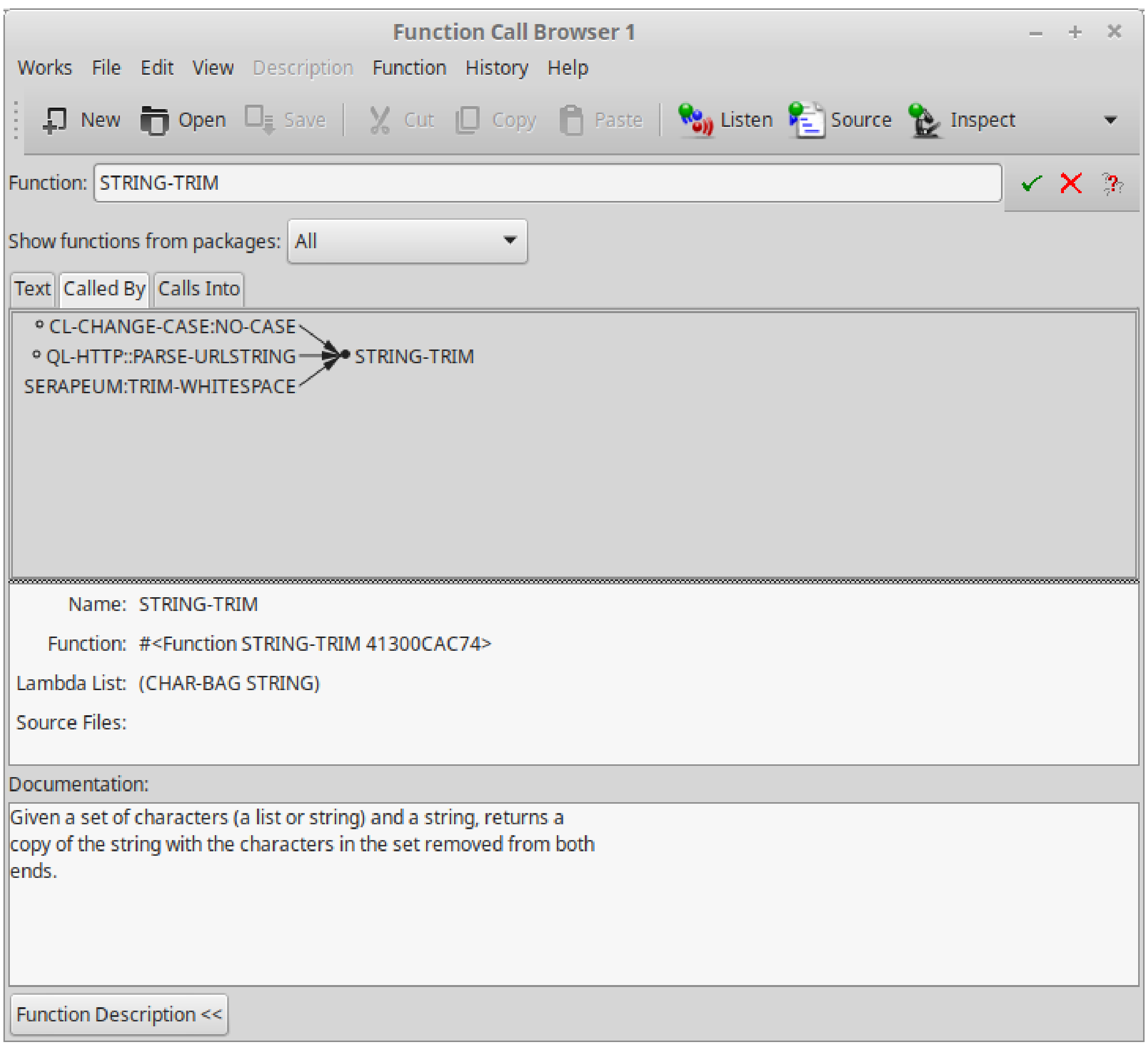
- [Chapter 8 of the documentation: the Class Browser]

The function call browser

The function call browser allows us to see a graph of the callers and the callees of a function. It provides several ways to filter the displayed information and to further inspect the call stack.

NB: The Slime functions to find such cross-references are `slime-who-[calls, references, binds, sets, depends-on, specializes, macroexpands]`.

After loading a couple packages, here's a simple example showing who calls the `string-trim` function.



The function call browser

It shows functions from all packages, but there is a select box to restrict it further, for example to the “current and used” or only to the current packages.

Double click on a function shown in the graph to go to its source. Again, as in many LispWorks views, the Function menu allows to further manipulate selected functions: trace, undefine, listen (paste the object to the Listener)…

The Text tab shows the same information, but textually, the callers and callees side by side.

We can see cross references for compiled code, and we must ensure the feature is on. When we compile code, LispWorks shows a compilation output likes this:

```
;;; Safety = 3, Speed = 1, Space = 1, Float = 1, Interruptible = 1
;;; Compilation speed = 1, Debug = 2, Fixnum safety = 3
;;; Source level debugging is on
;;; Source file recording is  on
;;; Cross referencing is on
```

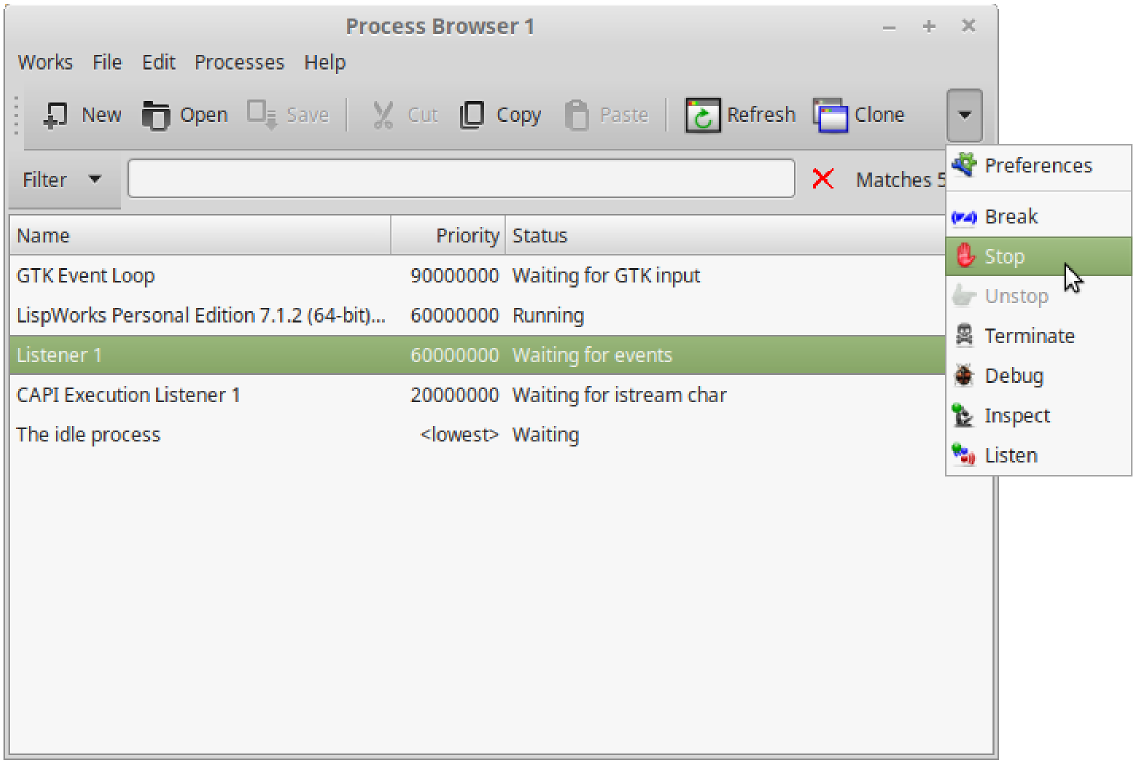
We see that cross referencing is on. Otherwise, activate it with `(toggle-source-debugging t)`.

See more:

- [Chapter 15: the function call browser]

The Process Browser

The Process Browser shows us a list of all threads running. The input area allows to filter by name. It accepts regular expressions. Then we can stop, inspect, listen, break into these processes.



“The process browser”

See more:

- Chapter 28: the Process Browser

Saving images

Saving images with LispWorks is different than with SBCL:

- we can save an image now, or schedule snapshots later in time
- the new created image becomes the default core image for our LispWorks environment
- the REPL session is saved
- the windows configuration is saved
- threads are saved

So, effectively, we can save an image and have our development environment back to the same state, effectively allowing to take snapshots of our current work and to continue where we left of.

For example, we can start a game from the REPL, play a little bit in its window, save an image, and when restored we will get the game and its state back.

Misc

We like the `Search Files` functionality. It is like a recursive `grep`, but we get a typical LispWorks graphical window that displays the results, allows to double-click on them and that offers some more actions.

Last but not least, have a look at the **compilation conditions browser**. LispWorks puts all warnings and errors into a special browser when we compile a system. From now on we can work on fixing them and see them disappear from the browser. That helps keeping track of warnings and errors during development.

Using LispWorks from Emacs and Slime

To do that, you have two possibilities. The first one is to start LispWorks normally, start a Swank server and connect to it from Emacs (Swank is the backend part of Slime).

First, let's load the dependencies:

```
(ql:quickload "swank")
;; or
(load "~/.emacs.d/elpa/slime-20xx/swank-loader.lisp")
```

Start a server:

```
(swank:create-server :port 9876)
;; Swank started at port: 9876.
9876
```

From Emacs, run `M-x slime-connect`, choose `localhost` and `9876` for the port.

You should be connected. Check with: `(lisp-implementation-type)`. You are now able to use LispWorks' features:

```
(setq button
      (make-instance 'capi:push-button
                     :data "Button"))
(capi:contain button)
```

The second possibility is to create a non-GUI LispWorks image, with Swank loaded, and to run this image from SLIME or SLY. For example, to create a so-called `console` image with multiprocessing enabled:

```
(in-package "CL-USER")
(load-all-patches)
(save-image "~/lw-console"
            :console t
            :multiprocessing t
            :environment nil)
```

and run LispWorks like this to create the new image ~/lw-console:

```
lispworks-7-0-0-x86-linux -build /tmp/resave.lisp
```

However: `console` is implemented **only for Windows and Mac**.

See LispWorks' documentation.

See also

- LispWorks IDE User Guide (check out the sections we didn't cover)
- LispWorks on Wikipedia
- the Awesome LispWorks list
- Real Image-based approach in Common Lisp - differences between SBCL and LispWorks.

Functions

Named functions: `defun`

Creating named functions is done with the `defun` keyword. It follows this model:

```
(defun <name> (list of arguments)
  "docstring"
  (function body))
```

The return value is the value returned by the last expression of the body (see below for more). There is no “return xx” statement.

So, for example:

```
(defun hello-world ()
  ;;               ^^ no arguments
  (print "hello world!"))
```

Call it:

```
(hello-world)
;; "hello world!"  <-- output
;; "hello world!"  <-- a string is returned.
```

Arguments

Base case: required arguments

Add in arguments like this:

```
(defun hello (name)
  "Say hello to `name'."
  (format t "hello ~a !~&" name))
;; HELLO
```

(where `~a` is the most used `format` directive to print a variable *aesthetically* and `~&` prints a newline)

Call the function:

```
(hello "me")
;; hello me !  <-- this is printed by `format`
;; NIL         <-- return value: `format t` prints a string
;;                 to standard output and returns nil.
```

If you don't specify the right amount of arguments, you'll be trapped into the interactive debugger with an explicit error message:

(hello)

invalid number of arguments: 0

Optional arguments: `&optional`

Optional arguments are declared after the `&optional` keyword in the lambda list. They are ordered, they must appear one after another.

This function:

```
(defun hello (name &optional age gender) …)
```

must be called like this:

```
(hello "me") ;; a value for the required argument,
             ;; zero optional arguments
(hello "me" "7")  ;; a value for age
(hello "me" 7 :h) ;; a value for age and gender
```

Named parameters: `&key`

It is not always convenient to remember the order of the arguments. It is thus possible to supply arguments by name: we declare them using `&key <name>`, we set them with `:name <value>` in the function call, and we use `name` as a regular variable in the function body. They are `nil` by default.

```lisp
(defun hello (name &key happy)
  "If `happy' is `t', print a smiley"
  (format t "hello ~a " name)
  (when happy
    (format t ":)~&")))
```

The following calls are possible:

```lisp
(hello "me")
(hello "me" :happy t)
(hello "me" :happy nil) ;; useless, equivalent to (hello "me")
```

and this is not valid: `(hello "me" :happy)`:

> odd number of &KEY arguments

A similar example of a function declaration, with several key parameters:

```lisp
(defun hello (name &key happy lisper cookbook-contributor-p) …)
```

it can be called with zero or more key parameters, in any order:

```lisp
(hello "me" :lisper t)
(hello "me" :lisper t :happy t)
(hello "me" :cookbook-contributor-p t :happy t)
```

Last but not least, you would quickly realize it, but we can choose the keys programmatically (they can be variables):

```lisp
(let ((key :happy)
      (val t))
  (hello "me" key val))
;; hello me :)
;; NIL
```

Mixing optional and key parameters

It is generally a style warning, but it is possible.

```lisp
(defun hello (&optional name &key happy)
  (format t "hello ~a " name)
```

```
  (when happy
    (format t ":)~&")))
```

In SBCL, this yields:

```
; in: DEFUN HELLO
;     (SB-INT:NAMED-LAMBDA HELLO
;         (&OPTIONAL NAME &KEY HAPPY)
;        (BLOCK HELLO (FORMAT T "hello ~a " NAME) (WHEN HAPPY
(FORMAT T ":)~&"))))
;
; caught STYLE-WARNING:
;      &OPTIONAL and &KEY found in the same lambda list:
(&OPTIONAL (NAME "John") &KEY
;                                                      HAPPY)
;
; compilation unit finished
;   caught 1 STYLE-WARNING condition
```

We can call it:

```
(hello "me" :happy t)
;; hello me :)
;; NIL
```

Default values to key parameters

In the lambda list, use pairs to give a default value to an optional or a key argument, like `(happy t)` below:

```
(defun hello (name &key (happy t))
```

Now `happy` is true by default.

Was a key parameter specified?

You can skip this tip for now if you want, but come back later to it as it can turn handy.

We saw that a default key parameter is `nil` by default (`(defun hello (name &key happy) …)`). But how can be distinguish between “the value is

NIL by default" and "the user wants it to be NIL"?

We saw how to use a tuple to set its default value:

```
&key (:happy t)
```

To answer our question, use a triple like this:

```
&key (happy t happy-p)
```

where `happy-p` serves as a *predicate* variable (using `-p` is only a convention, give it the name you want) to know if the key was supplied. If it was, then it will be `T`.

So now, we will print a sad face if `:happy` was explicitely set to NIL. We don't print it by default.

```
(defun hello (name &key (happy nil happy-p))
  (format t "Key supplied? ~a~&" happy-p)
  (format t "hello ~a " name)
  (when happy-p
    (if happy
      (format t ":)")
      (format t ":("))))
```

Variable number of arguments: `&rest`

Sometimes you want a function to accept a variable number of arguments. Use `&rest <variable>`, where `<variable>` will be a list.

```
(defun mean (x &rest numbers)
    (/ (apply #'+ x numbers)
       (1+ (length numbers))))

(mean 1)
(mean 1 2)  ;; => 3/2 (yes, it is printed as a ratio)
(mean 1 2 3 4 5) ;;  => 3
```

Defining key arguments, and allowing more: `&allow-other-keys`

Observe:

```
(defun hello (name &key happy)
  (format t "hello ~a~&" name))

(hello "me" :lisper t)
;; => Error: unknown keyword argument
```

whereas

```
(defun hello (name &key happy &allow-other-keys)
  (format t "hello ~a~&" name))

(hello "me" :lisper t)
;; hello me
```

We might need `&allow-other-keys` when passing around arguments or with higher level manipulation of functions.

Here's a real example. We define a function to open a file that always uses `:if-exists :supersede`, but still passes any other keys to the `open` function.

```
(defun open-supersede (f &rest other-keys &key &allow-other-keys
  (apply #'open f :if-exists :supersede other-keys))
```

In the case of a duplicated `:if-exists` argument, our first one takes precedence.

Return values

The return value of the function is the value returned by the last executed form of the body.

There are ways for non-local exits (`return-from <function name> <value>`), but they are usually not needed.

Common Lisp has also the concept of multiple return values.

Multiple return values: `values`, `multiple-value-bind` and `nth-value`

Returning multiple values is *not* like returning a tuple or a list of results ;) This is a common misconception.

Multiple values are specially useful and powerful because a change in them needs little to no refactoring.

```
(defun foo (a b c)
  a)
```

This function returns `a`.

```
(defvar *res* (foo :a :b :c))
;; :A
```

We use `values` to return multiple values:

```
(defun foo (a b c)
  (values a b c))
```

```
(setf *res* (foo :a :b :c))
;; :A
```

Observe here that `*res*` *is still* `:A`.

All functions that use the return value of `foo` need *not* to change, they still work. If we had returned a list or an array, this would be different.

multiple-value-bind

We destructure multiple values with `multiple-value-bind` (or `mvb`+TAB in Slime for short) and we can get one given its position with `nth-value`:

```lisp
(multiple-value-bind (res1 res2 res3)
    (foo :a :b :c)
  (format t "res1 is ~a, res2 is ~a, res2 is ~a~&"
     res1 res2 res3))
;; res1 is A, res2 is B, res2 is C
;; NIL
```

Its general form is

```lisp
(multiple-value-bind (var-1 .. var-n) expr
  body)
```

The variables `var-n` are not available outside the scope of `multiple-value-bind`.

With **nth-value**:

```lisp
(nth-value 0 (values :a :b :c))  ;; => :A
(nth-value 2 (values :a :b :c))  ;; => :C
(nth-value 9 (values :a :b :c))  ;; => NIL
```

Look here too that `values` is different from a list:

```lisp
(nth-value 0 '(:a :b :c)) ;; => (:A :B :C)
(nth-value 1 '(:a :b :c)) ;; => NIL
```

Note that `(values)` with no values returns… no values at all.

multiple-value-list

While we are at it: multiple-value-list turns multiple values to a list:

```lisp
(multiple-value-list (values 1 2 3))
;; (1 2 3)
```

The reverse is **values-list**, it turns a list to multiple values:

```lisp
(values-list '(1 2 3))
;; 1
;; 2
;; 3
```

Anonymous functions: `lambda`

Anonymous functions are created with `lambda`:

```
(lambda (x) (print x))
```

We can call a lambda with `funcall` or `apply` (see below).

If the first element of an unquoted list is a lambda expression, the lambda is called:

```
((lambda (x) (print x)) "hello")
;; hello
```

Calling functions programmatically: `funcall` and `apply`

`funcall` is to be used with a known number of arguments, when `apply` can be used on a list, for example from `&rest`:

```
(funcall #'+ 1 2)
(apply #'+ '(1 2))
```

Referencing functions by name: single quote `'` or sharpsign-quote `#'`?

In the example above, we used `#'`, but a single quote also works, and we can encounter it in the wild. Which one to use?

It is generally safer to use `#'`, because it respects lexical scope. Observe:

```
(defun foo (x)
  (* x 100))

(flet ((foo (x) (1+ x)))
  (funcall #'foo 1))
;; => 2, as expected
```

```lisp
;; But:

(flet ((foo (x) (1+ x)))
  (funcall 'foo 1))
;; => 100
```

`#'` is actually the shorthand for `(function …)`:

```lisp
(function +)
;; #<FUNCTION +>

(flet ((foo (x) (1+ x)))
  (print (function foo))
  (funcall (function foo) 1))
;; #<FUNCTION (FLET FOO) {1001C0ACFB}>
;; 2
```

Using `function` or the `#'` shorthand allows us to refer to local functions. If we pass instead a symbol to `funcall`, what is called is always the function named by that symbol in the *global environment*.

In addition, `#'` catches the function by value. If the function is redefined, bindings that refered to this function by `#'` will still run its original behaviour.

Let's assign a function to a parameter:

```lisp
(defparameter *foo-caller* #'foo)
(funcall *foo-caller* 1)
;; => 100
```

Now, if we redefine `foo`, the behaviour of `*foo-caller*` will *not* change:

```lisp
(defun foo (x) (1+ x))
;; WARNING: redefining CL-USER::FOO in DEFUN
;; FOO

(funcall *foo-caller* 1)
;; 100  ;; and not 2
```

If we bind the caller with `'foo`, a single quote, the function will be resolved at runtime:

```
(defun foo (x) (* x 100))  ;; back to original behavior.
(defparameter *foo-caller-2* 'foo)
;; *FOO-CALLER-2*
(funcall *foo-caller-2* 1)
;; 100

;; We change the definition:
(defun foo (x) (1+ x))
;; WARNING: redefining CL-USER::FOO in DEFUN
;; FOO

;; We try again:
(funcall *foo-caller-2* 1)
;; 2
```

The behaviour you want depends on your use case. Generally, using sharpsign-quote is less surprising. But if you are running a tight loop and you want live-update mechanisms (think a game or live visualisations), you might want to use a single quote so that your loop picks up the user's new function definition.

Higher order functions: functions that return functions

Writing functions that return functions is simple enough:

```
(defun adder (n)
  (lambda (x) (+ x n)))
;; ADDER
```

Here we have defined the function `adder` which returns an *object* of *type* [`function`](function).

To call the resulting function, we must use `funcall` or `apply`:

```lisp
(adder 5)
;; #<CLOSURE (LAMBDA (X) :IN ADDER) {100994ACDB}>
(funcall (adder 5) 3)
;; 8
```

Trying to call it right away leads to an illegal function call:

```lisp
((adder 3) 5)
In: (ADDER 3) 5
    ((ADDER 3) 5)
Error: Illegal function call.
```

Indeed, CL has different *namespaces* for functions and variables, i.e. the same *name* can refer to different things depending on its position in a form that's evaluated.

```lisp
;; The symbol foo is bound to nothing:
CL-USER> (boundp 'foo)
NIL
CL-USER> (fboundp 'foo)
NIL
;; We create a variable:
CL-USER> (defparameter foo 42)
FOO
* foo
42
;; Now foo is "bound":
CL-USER> (boundp 'foo)
T
;; but still not as a function:
CL-USER> (fboundp 'foo)
NIL
;; So let's define a function:
CL-USER> (defun foo (x) (* x x))
FOO
;; Now the symbol foo is bound as a function too:
CL-USER> (fboundp 'foo)
T
;; Get the function:
CL-USER> (function foo)
#<FUNCTION FOO>
;; and the shorthand notation:
```

```
* #'foo
#<FUNCTION FOO>
;; We call it:
(funcall (function adder) 5)
#<CLOSURE (LAMBDA (X) :IN ADDER) {100991761B}>
;; and call the lambda:
(funcall (funcall (function adder) 5) 3)
8
```

To simplify a bit, you can think of each symbol in CL having (at least) two “cells” in which information is stored. One cell - sometimes referred to as its *value cell* - can hold a value that is *bound* to this symbol, and you can use `boundp` to test whether the symbol is bound to a value (in the global environment). You can access the value cell of a symbol with `symbol-value`.

The other cell - sometimes referred to as its *function cell* - can hold the definition of the symbol’s (global) function binding. In this case, the symbol is said to be *fbound* to this definition. You can use `fboundp` to test whether a symbol is fbound. You can access the function cell of a symbol (in the global environment) with `symbol-function`.

Now, if a *symbol* is evaluated, it is treated as a *variable* in that its value cell is returned (just `foo`). If a *compound form*, i.e. a *cons*, is evaluated and its *car* is a symbol, then the function cell of this symbol is used (as in `(foo 3)`).

In Common Lisp, as opposed to Scheme, it is *not* possible that the car of the compound form to be evaluated is an arbitrary form. If it is not a symbol, it *must* be a *lambda expression*, which looks like `(lambda`*lambda-list* _form*_`)`.

This explains the error message we got above - `(adder 3)` is neither a symbol nor a lambda expression.

If we want to be able to use the symbol `*my-fun*` in the car of a compound form, we have to explicitly store something in its *function cell* (which is normally done for us by the macro `defun`):

```
;;; continued from above
CL-USER> (fboundp '*my-fun*)
NIL
CL-USER> (setf (symbol-function '*my-fun*) (adder 3))
#<CLOSURE (LAMBDA (X) :IN ADDER) {10099A5EFB}>
CL-USER> (fboundp '*my-fun*)
T
CL-USER> (*my-fun* 5)
8
```

Read the CLHS section about [form evaluation](#) for more.

Closures

Closures allow to capture lexical bindings:

```
(let ((limit 3)
      (counter -1))
    (defun my-counter ()
      (if (< counter limit)
          (incf counter)
          (setf counter 0))))

(my-counter)
0
(my-counter)
1
(my-counter)
2
(my-counter)
3
(my-counter)
0
```

Or similarly:

```
(defun repeater (n)
  (let ((counter -1))
     (lambda ()
       (if (< counter n)
         (incf counter)
```

```lisp
         (setf counter 0)))))

(defparameter *my-repeater* (repeater 3))
;; *MY-REPEATER*
(funcall *my-repeater*)
0
(funcall *my-repeater*)
1
(funcall *my-repeater*)
2
(funcall *my-repeater*)
3
(funcall *my-repeater*)
0
```

See more on Practical Common Lisp.

`setf` functions

A function name can also be a list of two symbols with `setf` as the first one, and where the first argument is the new value:

```lisp
(defun (setf <name>) (new-value <other arguments>)
  body)
```

This mechanism is particularly used for CLOS methods.

A silly example:

```lisp
(defparameter *current-name* ""
  "A global name.")

(defun hello (name)
  (format t "hello ~a~&" name))

(defun (setf hello) (new-value)
  (hello new-value)
  (setf *current-name* new-value)
  (format t "current name is now ~a~&" new-value))
```

```
(setf (hello) "Alice")
;; hello Alice
;; current name is now Alice
;; NIL
```

Currying

Concept

A related concept is that of *currying* which you might be familiar with if you're coming from a functional language. After we've read the last section that's rather easy to implement:

```
CL-USER> (defun curry (function &rest args)
           (lambda (&rest more-args)
               (apply function (append args more-args))))
CURRY
CL-USER> (funcall (curry #'+ 3) 5)
8
CL-USER> (funcall (curry #'+ 3) 6)
9
CL-USER> (setf (symbol-function 'power-of-ten) (curry #'expt 10)
#<Interpreted Function "LAMBDA (FUNCTION &REST ARGS)" {482DB969}
CL-USER> (power-of-ten 3)
1000
```

With the Alexandria library

Now that you know how to do it, you may appreciate using the implementation of the Alexandria library (in Quicklisp).

```
(ql:quickload "alexandria")


(defun adder (foo bar)
  "Add the two arguments."
  (+ foo bar))

(defvar add-one (alexandria:curry #'adder 1) "Add 1 to the argum
```

```lisp
(funcall add-one 10)  ;; => 11

(setf (symbol-function 'add-one) add-one)
(add-one 10)  ;; => 11
```

Documentation

- functions: http://www.lispworks.com/documentation/HyperSpec/Body/t_fn.htm#function
- ordinary lambda lists: http://www.lispworks.com/documentation/HyperSpec/Body/03_da.htm
- multiple-value-bind: http://clhs.lisp.se/Body/m_multip.htm

Data structures

We hope to give here a clear reference of the common data structures. To really learn the language, you should take the time to read other resources. The following resources, which we relied upon, also have many more details:

- Practical CL, by Peter Seibel
- CL Recipes, by E. Weitz, full of explanations and tips,
- the CL standard with a nice TOC, functions reference, extensive descriptions, more examples and warnings (i.e: everything). PDF mirror
- a Common Lisp quick reference

Don't miss the appendix and when you need more data structures, have a look at the awesome-cl list and Quickdocs.

Lists

Building lists. Cons cells, lists.

A list is also a sequence, so we can use the functions shown below.

The list basic element is the cons cell. We build lists by assembling cons cells.

```
(cons 1 2)
;; => (1 . 2) ;; representation with a point, a dotted pair.
```

It looks like this:

```
[o|o]--- 2
 |
 1
```

If the `cdr` of the first cell is another cons cell, and if the `cdr` of this last one is `nil`, we build a list:

```
(cons 1 (cons 2 nil))
;; => (1 2)
```

It looks like this:

```
[o|o]---[o|/]
 |       |
 1       2
```

(ascii art by draw-cons-tree).

See that the representation is not a dotted pair ? The Lisp printer understands the convention.

Finally we can simply build a literal list with `list`:

```
(list 1 2)
;; => (1 2)
```

or by calling quote:

```
'(1 2)
;; => (1 2)
```

which is shorthand notation for the function call `(quote (1 2))`.

Circular lists

A cons cell car or cdr can refer to other objects, including itself or other cells in the same list. They can therefore be used to define self-referential structures such as circular lists.

Before working with circular lists, tell the printer to recognise them and not try to print the whole list by setting *print-circle* to `T`:

```
(setf *print-circle* t)
```

A function which modifies a list, so that the last `cdr` points to the start of the list is:

```
(defun circular! (items)
  "Modifies the last cdr of list ITEMS, returning a circular lis
  (setf (cdr (last items)) items))

(circular! (list 1 2 3))
;; => #1=(1 2 3 . #1#)

(fifth (circular! (list 1 2 3)))
;; => 2
```

The list-length function recognises circular lists, returning `nil`.

The reader can also create circular lists, using Sharpsign Equal-Sign notation. An object (like a list) can be prefixed with `#n=` where `n` is an unsigned decimal integer (one or more digits). The label `#n#` can be used to refer to the object later in the expression:

```
'#42=(1 2 3 . #42#)
;; => #1=(1 2 3 . #1#)
```

Note that the label given to the reader (`n=42`) is discarded after reading, and the printer defines a new label (`n=1`).

Further reading

- Let over Lambda section on cyclic expressions

car/cdr or first/rest (and second… to tenth)

```
(car (cons 1 2)) ;; => 1
(cdr (cons 1 2)) ;; => 2
(first (cons 1 2)) ;; => 1
(first '(1 2 3)) ;; => 1
(rest '(1 2 3)) ;; => (2 3)
```

We can assign *any* new value with `setf`.

last, butlast, nbutlast (&optional n)

return the last cons cell in a list (or the nth last cons cells).

```
(last '(1 2 3))
;; => (3)
(car (last '(1 2 3)) ) ;; or (first (last …))
;; => 3
(butlast '(1 2 3))
;; => (1 2)
```

In Alexandria, `lastcar` is equivalent of `(first (last …))`:

```
(alexandria:lastcar '(1 2 3))
;; => 3
```

reverse, nreverse

`reverse` and `nreverse` return a new sequence.

`nreverse` is destructive. The N stands for **non-consing**, meaning it doesn't need to allocate any new cons cells. It *might* (but in practice, does) reuse and modify the original sequence:

```
(defparameter mylist '(1 2 3))
;; => (1 2 3)
(reverse mylist)
;; => (3 2 1)
mylist
;; => (1 2 3)
(nreverse mylist)
;; => (3 2 1)
mylist
;; => (1) in SBCL but implementation dependent.
```

append

`append` takes any number of list arguments and returns a new list containing the elements of all its arguments:

```
(append (list 1 2) (list 3 4))
;; => (1 2 3 4)
```

The new list shares some cons cells with the `(3 4)`:

http://gigamonkeys.com/book/figures/after-append.png

`nconc` is the recycling equivalent.

push (item, place)

`push` prepends *item* to the list that is stored in *place*, stores the resulting list in *place*, and returns the list.

```
(defparameter mylist '(1 2 3))
(push 0 mylist)
;; => (0 1 2 3)
```

```
(defparameter x ’(a (b c) d))
;; => (A (B C) D)
(push 5 (cadr x))
;; => (5 B C)
x
;; => (A (5 B C) D)
```

`push` is equivalent to `(setf place (cons item place ))` except that the subforms of *place* are evaluated only once, and *item* is evaluated before *place*.

There is no built-in function to **add to the end of a list**. It is a more costly operation (have to traverse the whole list). So if you need to do this: either consider using another data structure, either just `reverse` your list when needed.

pop

a destructive operation.

nthcdr (index, list)

Use this if `first`, `second` and the rest up to `tenth` are not enough.

car/cdr and composites (cadr, caadr…) - accessing lists inside lists

They make sense when applied to lists containing other lists.

```
(caar (list 1 2 3))                   ==> error
(caar (list (list 1 2) 3))            ==> 1
(cadr (list (list 1 2) (list 3 4)))   ==> (3 4)
(caadr (list (list 1 2) (list 3 4))) ==> 3
```

destructuring-bind (parameter*, list)

It binds the parameter values to the list elements. We can destructure trees, plists and even provide defaults.

Simple matching:

```
(destructuring-bind (x y z) (list 1 2 3)
  (list :x x :y y :z z))
;; => (:X 1 :Y 2 :Z 3)
```

Matching inside sublists:

```
(destructuring-bind (x (y1 y2) z) (list 1 (list 2 20) 3)
  (list :x x :y1 y1 :y2 y2 :z z))
;; => (:X 1 :Y1 2 :Y2 20 :Z 3)
```

The parameter list can use the usual `&optional`, `&rest` and `&key` parameters.

```
(destructuring-bind (x (y1 &optional y2) z) (list 1 (list 2) 3)
  (list :x x :y1 y1 :y2 y2 :z z))
;; => (:X 1 :Y1 2 :Y2 NIL :Z 3)

(destructuring-bind (&key x y z) (list :z 1 :y 2 :x 3)
  (list :x x :y y :z z))
;; => (:X 3 :Y 2 :Z 1)
```

The `&whole` parameter is bound to the whole list. It must be the first one and others can follow.

```
(destructuring-bind (&whole whole-list &key x y z)
    (list :z 1 :y 2 :x 3)
  (list :x x :y y :z z :whole whole-list))
;; => (:X 3 :Y 2 :Z 1 :WHOLE-LIST (:Z 1 :Y 2 :X 3))
```

Destructuring a plist, giving defaults:

(example from Common Lisp Recipes, by E. Weitz, Apress, 2016)

```
(destructuring-bind (&key a (b :not-found) c
                     &allow-other-keys)
    '(:c 23 :d "D" :a #\A :foo :whatever)
  (list a b c))
;; => (#\A :NOT-FOUND 23)
```

If this gives you the will to do pattern matching, see pattern matching.

Predicates: null, listp

`null` is equivalent to `not`, but considered better style.

`listp` tests whether an object is a cons cell or nil.

and sequences' predicates.

ldiff, tailp, list*, make-list, fill, revappend, nreconc, consp, atom

```
(make-list 3 :initial-element "ta")
;; => ("ta" "ta" "ta")

(make-list 3)
;; => (NIL NIL NIL)
(fill * "hello")
;; => ("hello" "hello" "hello")
```

member (elt, list)

Returns the tail of `list` beginning with the first element satisfying `eqlity`.

Accepts `:test`, `:test-not`, `:key` (functions or symbols).

```
(member 2 '(1 2 3))
;; (2 3)
```

Replacing objects in a tree: subst, sublis

subst and `subst-if` search and replace occurences of an element or subexpression in a tree (when it satisfies the optional `test`):

```
(subst 'one 1 '(1 2 3))
;; => (ONE 2 3)

(subst  '(1 . one) '(1 . 1) '((1 . 1) (2 . 2)) :test #'equal)
;; ((1 . ONE) (2 . 2))
```

sublis allows to replace many objects at once. It substitutes the objects given in `alist` and found in `tree` with their new values given in the alist:

```
(sublis '((x . 10) (y . 20))
        '(* x (+ x y) (* y y)))
;; (* 10 (+ 10 20) (* 20 20))
```

`sublis` accepts the `:test` and `:key` arguments. `:test` is a function that takes two arguments, the key and the subtree.

```
(sublis '((t . "foo"))
        '("one" 2 ("three" (4 5)))
        :key #'stringp)
;; ("foo" 2 ("foo" (4 5)))
```

Sequences

lists and **vectors** (and thus **strings**) are sequences.

Note: see also the strings page.

Many of the sequence functions take keyword arguments. All keyword arguments are optional and, if specified, may appear in any order.

Pay attention to the `:test` argument. It defaults to `eql` (for strings, use `:equal`).

The `:key` argument should be passed either nil, or a function of one argument. This key function is used as a filter through which the elements of the sequence are seen. For instance, this:

```
(find x y :key 'car)
```

is similar to `(assoc* x y)`: It searches for an element of the list whose car equals x, rather than for an element which equals x itself. If `:key` is omitted or nil, the filter is effectively the identity function.

Example with an alist (see definition below):

```
(defparameter my-alist (list (cons 'foo "foo")
                             (cons 'bar "bar")))
;; => ((FOO . "foo") (BAR . "bar"))
(find 'bar my-alist)
;; => NIL
(find 'bar my-alist :key 'car)
;; => (BAR . "bar")
```

For more, use a `lambda` that takes one parameter.

```
(find 'bar my-alist :key (lambda (it) (car it)))
```

```
(find 'bar my-alist :key ^(car %))
(find 'bar my-alist :key (lm (it) (car it)))
```

Predicates: every, some,…

`every, notevery (test, sequence)`: return nil or t, respectively, as soon as one test on any set of the corresponding elements of sequences returns nil.

```lisp
(defparameter foo '(1 2 3))
(every #'evenp foo)
;; => NIL
(some #'evenp foo)
;; => T
```

with a list of strings:

```lisp
(defparameter str '("foo" "bar" "team"))
(every #'stringp str)
;; => T
(some (lambda (it) (= 3 (length it))) str)
;; => T
```

`some`, `notany` *(test, sequence)*: return either the value of the test, or nil.

Functions

See also sequence functions defined in [Alexandria](): `starts-with`, `ends-with`, `ends-with-subseq`, `length=`, `emptyp`,…

length (sequence)

elt (sequence, index) - find by index

beware, here the sequence comes first.

count (foo sequence)

Return the number of elements in sequence that match *foo*.

Additional paramaters: `:from-end`, `:start`, `:end`.

See also `count-if`, `count-not` *(test-function sequence)*.

subseq (sequence start, [end])

```lisp
(subseq (list 1 2 3) 0)
;; (1 2 3)
(subseq (list 1 2 3) 1 2)
;; (2)
```

However, watch out if the `end` is larger than the list:

```lisp
(subseq (list 1 2 3) 0 99)
;; => Error: the bounding indices 0 and 99
;; are bad for a sequence of length 3.
```

To this end, use `alexandria-2:subseq*`:

```lisp
(alexandria-2:subseq* (list 1 2 3) 0 99)
;; (1 2 3)
```

`subseq` is "setf"able, but only works if the new sequence has the same length of the one to replace.

sort, stable-sort (sequence, test [, key function])

These sort functions are destructive, so one may prefer to copy the sequence with `copy-seq` before sorting:

```lisp
(sort (copy-seq seq) :test #'string<)
```

Unlike `sort`, `stable-sort` guarantees to keep the order of the argument. In theory, the result of this:

```lisp
(sort '((1 :a) (1 :b)) #'< :key #'first)
```

could be either `((1 :A) (1 :B))`, either `((1 :B) (1 :A))`. On my tests, the order is preserved, but the standard does not guarantee it.

find, position (foo, sequence) - get index

also `find-if`, `find-if-not`, `position-if`, `position-if-not` *(test sequence)*. See `:key` and `:test` parameters.

```lisp
(find 20 '(10 20 30))
;; 20
(position 20 '(10 20 30))
;; 1
```

search and mismatch (sequence-a, sequence-b)

`search` searches in sequence-b for a subsequence that matches sequence-a. It returns the *position* in sequence-b, or NIL. It has the `from-end`, `end1`, `end2` and the usual `test` and `key` parameters.

```lisp
(search '(20 30) '(10 20 30 40))
;; 1
(search '("b" "c") '("a" "b" "c"))
;; NIL
(search '("b" "c") '("a" "b" "c") :test #'equal)
;; 1
(search "bc" "abc")
;; 1
```

`mismatch` returns the position where the two sequences start to differ:

```lisp
(mismatch '(10 20 99) '(10 20 30))
;; 2
(mismatch "hellolisper" "helloworld")
;; 5
(mismatch "same" "same")
;; NIL
(mismatch "foo" "bar")
;; 0
```

substitute, nsubstitute[if,if-not]

Return a sequence of the same kind as `sequence` with the same elements, except that all elements equal to `old` are replaced with `new`.

```lisp
(substitute #\o #\x "hellx") ;; => "hello"
(substitute :a :x '(:a :x :x)) ;; => (:A :A :A)
(substitute "a" "x" '("a" "x" "x") :test #'string=)
;; => ("a" "a" "a")
```

sort, stable-sort, merge

(see above)

replace (sequence-a, sequence-b, &key start1, end1)

Destructively replace elements of sequence-a with elements of sequence-b.

The full signature is:

```
(replace sequence1 sequence2
      &rest args
      &key (start1 0) (end1 nil) (start2 0) (end2 nil))
```

Elements are copied to the subseqeuence bounded by START1 and END1, from the subsequence bounded by START2 and END2. If these subsequences are not of the same length, then the shorter length determines how many elements are copied.

```
(replace "xxx" "foo")
"foo"

(replace "xxx" "foo" :start1 1)
"xfo"

(replace "xxx" "foo" :start1 1 :start2 1)
"xoo"

(replace "xxx" "foo" :start1 1 :start2 1 :end2 2)
"xox"
```

remove, delete (foo sequence)

Make a copy of sequence without elements matching foo. Has `:start/end`, `:key` and `:count` parameters.

`delete` is the recycling version of `remove`.

```lisp
(remove "foo" '("foo" "bar" "foo") :test 'equal)
;; => ("bar")
```

see also `remove-if[-not]` below.

remove-duplicates, delete-duplicates (sequence)

remove-duplicates returns a new sequence with uniq elements. `delete-duplicates` may modify the original sequence.

`remove-duplicates` accepts the following, usual arguments: `from-end test test-not start end key`.

```lisp
(remove-duplicates '(:foo :foo :bar))
(:FOO :BAR)

(remove-duplicates '("foo" "foo" "bar"))
("foo" "foo" "bar")

(remove-duplicates '("foo" "foo" "bar") :test #'string-equal)
("foo" "bar")
```

mapping (map, mapcar, remove-if[-not],…)

If you're used to map and filter in other languages, you probably want `mapcar`. But it only works on lists, so to iterate on vectors (and produce either a vector or a list, use `(map 'list function vector)`.

mapcar also accepts multiple lists with `&rest more-seqs`. The mapping stops as soon as the shortest sequence runs out.

`map` takes the output-type as first argument (`'list`, `'vector` or `'string`):

```lisp
(defparameter foo '(1 2 3))
(map 'list (lambda (it) (* 10 it)) foo)
```

`reduce` *(function, sequence)*. Special parameter: `:initial-value`.

```
(reduce '- '(1 2 3 4))
;; => -8
(reduce '- '(1 2 3 4) :initial-value 100)
;; => 90
```

Filter is here called `remove-if-not`.

Flatten a list (Alexandria)

With [Alexandria](), we have the `flatten` function.

Creating lists with variables

That's one use of the `backquote`:

```
(defparameter *var* "bar")
;; First try:
'("foo" *var* "baz") ;; no backquote
;; => ("foo" *VAR* "baz") ;; nope
```

Second try, with backquote interpolation:

```
`("foo" ,*var* "baz")     ;; backquote, comma
;; => ("foo" "bar" "baz") ;; good
```

The backquote first warns we'll do interpolation, the comma introduces the value of the variable.

If our variable is a list:

```
(defparameter *var* '("bar" "baz"))
;; First try:
`("foo" ,*var*)
;; => ("foo" ("bar" "baz")) ;; nested list
`("foo" ,@*var*)            ;; backquote, comma-@ to
;; => ("foo" "bar" "baz")
```

E. Weitz warns that "objects generated this way will very likely share structure (see Recipe 2-7)".

Comparing lists

We can use sets functions.

Set

We show below how to use set operations on lists.

A set doesn't contain twice the same element and is unordered.

Most of these functions have recycling (modifying) counterparts, starting with "n": `nintersection`,… They all accept the usual `:key` and `:test` arguments, so use the test `#'string=` or `#'equal` if you are working with strings.

For more, see functions in [Alexandria](): `setp`, `set-equal`,… and the FSet library, shown in the next section.

`intersection` of lists

What elements are both in list-a and list-b ?

```
(defparameter list-a '(0 1 2 3))
(defparameter list-b '(0 2 4))
(intersection list-a list-b)
;; => (2 0)
```

Remove the elements of list-b from list-a (`set-difference`)

```
(set-difference list-a list-b)
;; => (3 1)
(set-difference list-b list-a)
;; => (4)
```

Join two lists with uniq elements (`union`)

```
(union list-a list-b)
;; => (3 1 0 2 4) ;; order can be different in your lisp
```

Remove elements that are in both lists (`set-exclusive-or`)

```
(set-exclusive-or list-a list-b)
;; => (4 3 1)
```

Add an element to a set (`adjoin`)

A new set is returned, the original set is not modified.

```
(adjoin 3 list-a)
;; => (0 1 2 3)   ;; <-- nothing was changed, 3 was already ther

(adjoin 5 list-a)
;; => (5 0 1 2 3) ;; <-- element added in front.

list-a
;; => (0 1 2 3)  ;; <-- original list unmodified.
```

Check if this is a subset (`subsetp`)

```
(subsetp '(1 2 3) list-a)
;; => T

(subsetp '(1 1 1) list-a)
;; => T

(subsetp '(3 2 1) list-a)
;; => T

(subsetp '(0 3) list-a)
;; => T
```

Fset - immutable data structure

You may want to have a look at the FSet library (in Quicklisp).

Arrays and vectors

Arrays have constant-time access characteristics.

They can be fixed or adjustable. A *simple array* is neither displaced (using `:displaced-to`, to point to another array) nor adjustable (`:adjust-array`), nor does it have a fill pointer (`fill-pointer`, that moves when we add or remove elements).

A **vector** is an array with rank 1 (of one dimension). It is also a *sequence* (see above).

A *simple vector* is a simple array that is also not specialized (it doesn't use `:element-type` to set the types of the elements).

Create an array, one or many dimensions

`make-array` *(sizes-list :adjustable bool)*

`adjust-array` *(array, sizes-list, :element-type, :initial-element)*

Access: aref (array i [j …])

`aref` *(array i j k …)* or `row-major-aref` *(array i)* equivalent to `(aref i i i …)`.

The result is `setf`able.

```
(defparameter myarray (make-array '(2 2 2) :initial-element 1))
myarray
;; => #3A(((1 1) (1 1)) ((1 1) (1 1)))
(aref myarray 0 0 0)
;; => 1
(setf (aref myarray 0 0 0) 9)
;; => 9
(row-major-aref myarray 0)
;; => 9
```

Sizes

`array-total-size` *(array)*: how many elements will fit in the array ?

`array-dimensions` *(array)*: list containing the length of the array's dimensions.

`array-dimension` *(array i)*: length of the *i*th dimension.

`array-rank` number of dimensions of the array.

```
(defparameter myarray (make-array '(2 2 2)))
;; => MYARRAY
myarray
;; => #3A(((0 0) (0 0)) ((0 0) (0 0)))
(array-rank myarray)
;; => 3
(array-dimensions myarray)
;; => (2 2 2)
(array-dimension myarray 0)
;; => 2
(array-total-size myarray)
;; => 8
```

Vectors

Create with `vector` or the reader macro `#()`. It returns a *simple vector.*

```
(vector 1 2 3)
;; => #(1 2 3)
#(1 2 3)
;; => #(1 2 3)
```

`vector-push` *(foo vector)*: replace the vector element pointed to by the fill pointer by foo. Can be destructive.

`vector-push-extend` *(foo vector [extension-num])*t

`vector-pop` *(vector)*: return the element of vector its fill pointer points to.

`fill-pointer` *(vector)*. `setf`able.

and see also the *sequence* functions.

Transforming a vector to a list.

If you're mapping over it, see the `map` function whose first parameter is the result type.

Or use `(coerce vector 'list)`.

Hash Table

Hash Tables are a powerful data structure, associating keys with values in a very efficient way. Hash Tables are often preferred over association lists whenever performance is an issue, but they introduce a little overhead that makes assoc lists better if there are only a few key-value pairs to maintain.

Alists can be used sometimes differently though:

- they can be ordered
- we can push cons cells that have the same key, remove the one in front and we have a stack
- they have a human-readable printed representation
- they can be easily (de)serialized
- because of RASSOC, keys and values in alists are essentially interchangeable; whereas in hash tables, keys and values play very different roles (as usual, see CL Recipes for more).

Creating a Hash Table

Hash Tables are created using the function `make-hash-table`. It has no required argument. Its most used optional keyword argument is `:test`, specifying the function used to test the equality of keys.

Note: see shorter notations in the Serapeum or Rutils libraries. For example, Serapeum has `dict`, and Rutils a `#h` reader macro.

Adding an Element to a Hash Table

If you want to add an element to a hash table, you can use `gethash`, the function to retrieve elements from the hash table, in conjunction with [`setf`](setf).

```
CL-USER> (defparameter *my-hash* (make-hash-table))
*MY-HASH*
CL-USER> (setf (gethash 'one-entry *my-hash*) "one")
"one"
CL-USER> (setf (gethash 'another-entry *my-hash*) 2/4)
1/2
CL-USER> (gethash 'one-entry *my-hash*)
"one"
T
CL-USER> (gethash 'another-entry *my-hash*)
1/2
T
```

With Serapeum's `dict`, we can create a hash-table and add elements to it in one go:

```
(defparameter *my-hash* (dict :one-entry "one"
                              :another-entry 2/4))
;; =>
 (dict
  :ONE-ENTRY "one"
  :ANOTHER-ENTRY 1/2
 )
```

Getting a value from a Hash Table

The function `gethash` takes two required arguments: a key and a hash table. It returns two values: the value corresponding to the key in the hash table (or `nil` if not found), and a boolean indicating whether the key was found in the table. That second value is necessary since `nil` is a valid value in a key-value pair, so getting `nil` as first value from `gethash` does not necessarily mean that the key was not found in the table.

Getting a key that does not exist with a default value

`gethash` has an optional third argument:

```lisp
(gethash 'bar *my-hash* "default-bar")
;; => "default-bar"
;;     NIL
```

Getting all keys or all values of a hash table

The Alexandria library (in Quicklisp) has the functions `hash-table-keys` and `hash-table-values` for that.

```lisp
(ql:quickload "alexandria")
;; […]
(alexandria:hash-table-keys *my-hash*)
;; => (BAR)
```

Testing for the Presence of a Key in a Hash Table

The first value returned by `gethash` is the object in the hash table that's associated with the key you provided as an argument to `gethash` or `nil` if no value exists for this key. This value can act as a generalized boolean if you want to test for the presence of keys.

```lisp
CL-USER> (defparameter *my-hash* (make-hash-table))
*MY-HASH*
CL-USER> (setf (gethash 'one-entry *my-hash*) "one")
"one"
CL-USER> (if (gethash 'one-entry *my-hash*)
           "Key exists"
           "Key does not exist")
"Key exists"
CL-USER> (if (gethash 'another-entry *my-hash*)
           "Key exists"
           "Key does not exist")
"Key does not exist"
```

But note that this does *not* work if `nil` is amongst the values that you want to store in the hash.

```lisp
CL-USER> (setf (gethash 'another-entry *my-hash*) nil)
NIL
CL-USER> (if (gethash 'another-entry *my-hash*)
```

```
            "Key exists"
            "Key does not exist")
"Key does not exist"
```

In this case you'll have to check the *second* return value of `gethash` which will always return `nil` if no value is found and T otherwise.

```
CL-USER> (if (nth-value 1 (gethash 'another-entry *my-hash*))
            "Key exists"
            "Key does not exist")
"Key exists"
CL-USER> (if (nth-value 1 (gethash 'no-entry *my-hash*))
            "Key exists"
            "Key does not exist")
"Key does not exist"
```

Deleting from a Hash Table

Use [remhash] to delete a hash entry. Both the key and its associated value will be removed from the hash table. `remhash` returns T if there was such an entry, `nil` otherwise.

```
CL-USER> (defparameter *my-hash* (make-hash-table))
*MY-HASH*
CL-USER> (setf (gethash 'first-key *my-hash*) 'one)
ONE
CL-USER> (gethash 'first-key *my-hash*)
ONE
T
CL-USER> (remhash 'first-key *my-hash*)
T
CL-USER> (gethash 'first-key *my-hash*)
NIL
NIL
CL-USER> (gethash 'no-entry *my-hash*)
NIL
NIL
CL-USER> (remhash 'no-entry *my-hash*)
NIL
CL-USER> (gethash 'no-entry *my-hash*)
```

```
NIL
NIL
```

Deleting a Hash Table

Use [clrhash]() to delete a hash table. This will remove all of the data from the hash table and return the deleted table.

```
CL-USER> (defparameter *my-hash* (make-hash-table))
*MY-HASH*
CL-USER> (setf (gethash 'first-key *my-hash*) 'one)
ONE
CL-USER> (setf (gethash 'second-key *my-hash*) 'two)
TWO
CL-USER> *my-hash*
#<hash-table :TEST eql :COUNT 2 {10097BF4E3}>
CL-USER> (clrhash *my-hash*)
#<hash-table :TEST eql :COUNT 0 {10097BF4E3}>
CL-USER> (gethash 'first-key *my-hash*)
NIL
NIL
CL-USER> (gethash 'second-key *my-hash*)
NIL
NIL
```

Traversing a Hash Table

If you want to perform an action on each entry (i.e., each key-value pair) in a hash table, you have several options:

You can use [maphash]() which iterates over all entries in the hash table. Its first argument must be a function which accepts *two* arguments, the key and the value of each entry. Note that due to the nature of hash tables you *can't* control the order in which the entries are provided by `maphash` (or other traversing constructs). `maphash` always returns `nil`.

```
CL-USER> (defparameter *my-hash* (make-hash-table))
*MY-HASH*
CL-USER> (setf (gethash 'first-key *my-hash*) 'one)
ONE
```

```
CL-USER> (setf (gethash 'second-key *my-hash*) 'two)
TWO
CL-USER> (setf (gethash 'third-key *my-hash*) nil)
NIL
CL-USER> (setf (gethash nil *my-hash*) 'nil-value)
NIL-VALUE
CL-USER> (defun print-hash-entry (key value)
    (format t "The value associated with the key ~S is ~S~%"
            key value))
PRINT-HASH-ENTRY
CL-USER> (maphash #'print-hash-entry *my-hash*)
The value associated with the key FIRST-KEY is ONE
The value associated with the key SECOND-KEY is TWO
The value associated with the key THIRD-KEY is NIL
The value associated with the key NIL is NIL-VALUE
```

You can also use `with-hash-table-iterator`, a macro which turns (via `macrolet`) its first argument into an iterator that on each invocation returns three values per hash table entry - a generalized boolean that's true if an entry is returned, the key of the entry, and the value of the entry. If there are no more entries, only one value is returned - `nil`.

```
;;; same hash-table as above
CL-USER> (with-hash-table-iterator (my-iterator *my-hash*)
           (loop
              (multiple-value-bind (entry-p key value)
                  (my-iterator)
                (if entry-p
                    (print-hash-entry key value)
                    (return)))))
The value associated with the key FIRST-KEY is ONE
The value associated with the key SECOND-KEY is TWO
The value associated with the key THIRD-KEY is NIL
The value associated with the key NIL is NIL-VALUE
NIL
```

Note the following caveat from the HyperSpec: "It is unspecified what happens if any of the implicit interior state of an iteration is returned outside the dynamic extent of the `with-hash-table-iterator` form such as by returning some closure over the invocation form."

And there's always loop:

```
;;; same hash-table as above
CL-USER> (loop for key being the hash-keys of *my-hash*
           do (print key))
FIRST-KEY
SECOND-KEY
THIRD-KEY
NIL
NIL
CL-USER> (loop for key being the hash-keys of *my-hash*
           using (hash-value value)
           do (format t "The value associated with the key ~S is
                      key value))
The value associated with the key FIRST-KEY is ONE
The value associated with the key SECOND-KEY is TWO
The value associated with the key THIRD-KEY is NIL
The value associated with the key NIL is NIL-VALUE
NIL
CL-USER> (loop for value being the hash-values of *my-hash*
           do (print value))
ONE
TWO

NIL
NIL-VALUE
NIL
CL-USER> (loop for value being the hash-values of *my-hash*
           using (hash-key key)
           do (format t "~&~A -> ~A" key value))
FIRST-KEY -> ONE
SECOND-KEY -> TWO
THIRD-KEY -> NIL
NIL -> NIL-VALUE
NIL
```

Traversing keys or values

To map over keys or values we can again rely on Alexandria with `maphash-keys` and `maphash-values`.

Counting the Entries in a Hash Table

No need to use your fingers - Common Lisp has a built-in function to do it for you: `hash-table-count`.

```
CL-USER> (defparameter *my-hash* (make-hash-table))
*MY-HASH*
CL-USER> (hash-table-count *my-hash*)
0
CL-USER> (setf (gethash 'first *my-hash*) 1)
1
CL-USER> (setf (gethash 'second *my-hash*) 2)
2
CL-USER> (setf (gethash 'third *my-hash*) 3)
3
CL-USER> (hash-table-count *my-hash*)
3
CL-USER> (setf (gethash 'second *my-hash*) 'two)
TWO
CL-USER> (hash-table-count *my-hash*)
3
CL-USER> (clrhash *my-hash*)
#<EQL hash table, 0 entries {48205F35}>
CL-USER> (hash-table-count *my-hash*)
0
```

Printing a Hash Table readably

With print-object (non portable)

It is very tempting to use `print-object`. It works under several implementations, but this method is actually not portable. The standard doesn't permit to do so, so this is undefined behaviour.

```
(defmethod print-object ((object hash-table) stream)
  (format stream "#HASH{~{~{(~a : ~a)~}~^ ~}}"
          (loop for key being the hash-keys of object
                using (hash-value value)
                collect (list key value))))
```

gives:

```
;; WARNING:
;;       redefining  PRINT-OBJECT  (#<STRUCTURE-CLASS  COMMON-
LISP:HASH-TABLE>
;;                                  #<SB-PCL:SYSTEM-CLASS COMMON-
LISP:T>) in DEFMETHOD
;;  #<STANDARD-METHOD  COMMON-LISP:PRINT-OBJECT  (HASH-TABLE  T)
{1006A0D063}>
```

and let’s try it:

```
(let ((ht (make-hash-table)))
  (setf (gethash :foo ht) :bar)
  ht)
;; #HASH{(FOO : BAR)}
```

With a custom function (portable way)

Here’s a portable way.

This snippets prints the keys, values and the test function of a hash-table, and uses `alexandria:alist-hash-table` to read it back in:

```
;; https://github.com/phoe/phoe-toolbox/blob/master/phoe-toolbox
(defun print-hash-table-readably (hash-table
                                  &optional
                                  (stream *standard-output*))
  "Prints a hash table readably using ALEXANDRIA:ALIST-HASH-TABL
  (let ((test (hash-table-test hash-table))
        (*print-circle* t)
        (*print-readably* t))
    (format stream "#.(ALEXANDRIA:ALIST-HASH-TABLE '(~%")
    (maphash (lambda (k v) (format stream "   (~S . ~S)~%" k v))
    (format stream "   ) :TEST '~A)" test)
    hash-table))
```

Example output:

```
#.(ALEXANDRIA:ALIST-HASH-TABLE
'((ONE . 1))
  :TEST 'EQL)
#<HASH-TABLE :TEST EQL :COUNT 1 {10046D4863}>
```

This output can be read back in to create a hash-table:

```
(read-from-string
 (with-output-to-string (s)
   (print-hash-table-readably
    (alexandria:alist-hash-table
     '((a . 1) (b . 2) (c . 3))) s)))
;; #<HASH-TABLE :TEST EQL :COUNT 3 {1009592E23}>
;; 83
```

With Serapeum (readable and portable)

The Serapeum library has the `dict` constructor, the function `pretty-print-hash-table` and the `toggle-pretty-print-hash-table` switch, all which do *not* use `print-object` under the hood.

```
CL-USER> (serapeum:toggle-pretty-print-hash-table)
T
CL-USER> (serapeum:dict :a 1 :b 2 :c 3)
(dict
  :A 1
  :B 2
  :C 3
 )
```

This printed representation can be read back in.

Thread-safe Hash Tables

The standard hash-table in Common Lisp is not thread-safe. That means that simple access operations can be interrupted in the middle and return a wrong result.

Implementations offer different solutions.

With **SBCL**, we can create thread-safe hash tables with the `:synchronized` keyword to `make-hash-table`: http://www.sbcl.org/manual/#Hash-Table-Extensions.

> If nil (the default), the hash-table may have multiple concurrent readers, but results are undefined if a thread writes to the hash-table concurrently with another reader or writer. If t, all concurrent accesses are safe, but note that clhs 3.6 (Traversal Rules and Side Effects) remains in force. See also: sb-ext:with-locked-hash-table.

```
(defparameter *my-hash* (make-hash-table :synchronized t))
```

But, operations that expand to two accesses, like the modify macros (`incf`) or this:

```
(setf (gethash :a *my-hash*) :new-value)
```

need to be wrapped around `sb-ext:with-locked-hash-table`:

> Limits concurrent accesses to HASH-TABLE for the duration of BODY. If HASH-TABLE is synchronized, BODY will execute with exclusive ownership of the table. If HASH-TABLE is not synchronized, BODY will execute with other WITH-LOCKED-HASH-TABLE bodies excluded – exclusion of hash-table accesses not surrounded by WITH-LOCKED-HASH-TABLE is unspecified.

```
(sb-ext:with-locked-hash-table (*my-hash*)
  (setf (gethash :a *my-hash*) :new-value))
```

In **LispWorks**, hash-tables are thread-safe by default. But likewise, there is no guarantee of atomicity *between* access operations, so we can use with-hash-table-locked.

Ultimately, you might like what the **cl-gserver library** proposes. It offers helper functions around hash-tables and its actors/agent system to allow thread-safety. They also maintain the order of updates and reads.

Performance Issues: The Size of your Hash Table

The `make-hash-table` function has a couple of optional parameters which control the initial size of your hash table and how it'll grow if it needs to grow. This can be an important performance issue if you're working with large hash tables. Here's an (admittedly not very scientific) example with [CMUCL](#) pre-18d on Linux:

```
CL-USER> (defparameter *my-hash* (make-hash-table))
*MY-HASH*
CL-USER> (hash-table-size *my-hash*)
65
CL-USER> (hash-table-rehash-size *my-hash*)
1.5
CL-USER> (time (dotimes (n 100000)
                 (setf (gethash n *my-hash*) n)))
Compiling LAMBDA NIL:
Compiling Top-Level Form:

Evaluation took:
  0.27 seconds of real time
  0.25 seconds of user run time
  0.02 seconds of system run time
  0 page faults and
  8754768 bytes consed.
NIL
CL-USER> (time (dotimes (n 100000)
                 (setf (gethash n *my-hash*) n)))
Compiling LAMBDA NIL:
Compiling Top-Level Form:

Evaluation took:
  0.05 seconds of real time
  0.05 seconds of user run time
  0.0 seconds of system run time
  0 page faults and
  0 bytes consed.
NIL
```

The values for `hash-table-size` and `hash-table-rehash-size` are implementation-dependent. In our case, CMUCL chooses and initial size of 65, and it will increase the size of the hash by 50 percent whenever it needs to grow. Let's see how often we have to re-size the hash until we reach the final size…

```
CL-USER> (log (/ 100000 65) 1.5)
18.099062
CL-USER> (let ((size 65))
           (dotimes (n 20)
             (print (list n size))
             (setq size (* 1.5 size))))
(0 65)
(1 97.5)
(2 146.25)
(3 219.375)
(4 329.0625)
(5 493.59375)
(6 740.3906)
(7 1110.5859)
(8 1665.8789)
(9 2498.8184)
(10 3748.2275)
(11 5622.3413)
(12 8433.512)
(13 12650.268)
(14 18975.402)
(15 28463.104)
(16 42694.656)
(17 64041.984)
(18 96062.98)
(19 144094.47)
NIL
```

The hash has to be re-sized 19 times until it's big enough to hold 100,000 entries. That explains why we saw a lot of consing and why it took rather long to fill the hash table. It also explains why the second run was much faster - the hash table already had the correct size.

Here's a faster way to do it: If we know in advance how big our hash will be, we can start with the right size:

```
CL-USER> (defparameter *my-hash* (make-hash-table :size 100000))
*MY-HASH*
CL-USER> (hash-table-size *my-hash*)
100000
CL-USER> (time (dotimes (n 100000)
                 (setf (gethash n *my-hash*) n)))
```

```
Compiling LAMBDA NIL:
Compiling Top-Level Form:

Evaluation took:
  0.04 seconds of real time
  0.04 seconds of user run time
  0.0 seconds of system run time
  0 page faults and
  0 bytes consed.
NIL
```

That's obviously much faster. And there was no consing involved because we didn't have to re-size at all. If we don't know the final size in advance but can guess the growth behaviour of our hash table we can also provide this value to `make-hash-table`. We can provide an integer to specify absolute growth or a float to specify relative growth.

```
CL-USER> (defparameter *my-hash* (make-hash-table :rehash-size 1
*MY-HASH*
CL-USER> (hash-table-size *my-hash*)
65
CL-USER> (hash-table-rehash-size *my-hash*)
100000
CL-USER> (time (dotimes (n 100000)
                 (setf (gethash n *my-hash*) n)))
Compiling LAMBDA NIL:
Compiling Top-Level Form:

Evaluation took:
  0.07 seconds of real time
  0.05 seconds of user run time
  0.01 seconds of system run time
  0 page faults and
  2001360 bytes consed.
NIL
```

Also rather fast (we only needed one re-size) but much more consing because almost the whole hash table (minus 65 initial elements) had to be built during the loop.

Note that you can also specify the `rehash-threshold` while creating a new hash table. One final remark: Your implementation is allowed to *completely ignore* the values provided for `rehash-size` and `rehash-threshold`…

Alist

Definition

An association list is a list of cons cells.

This simple example:

```
(defparameter *my-alist* (list (cons 'foo "foo")
                               (cons 'bar "bar")))
;; => ((FOO . "foo") (BAR . "bar"))
```

looks like this:

```
[o|o]---[o|/]
 |       |
 |      [o|o]---"bar"
 |       |
 |      BAR
 |
[o|o]---"foo"
 |
FOO
```

Construction

We can construct an alist like its representation:

```
(setf *my-alist* '((:foo . "foo")
                   (:bar . "bar")))
```

The constructor `pairlis` associates a list of keys and a list of values:

```
(pairlis '(:foo :bar)
         '("foo" "bar"))
;; => ((:BAR . "bar") (:FOO . "foo"))
```

Alists are just lists, so you can have the same key multiple times in the same alist:

```
(setf *alist-with-duplicate-keys*
  '((:a . 1)
    (:a . 2)
    (:b . 3)
    (:a . 4)
    (:c . 5)))
```

Access

To get a key, we have `assoc` (use `:test 'equal` when your keys are strings, as usual). It returns the whole cons cell, so you may want to use `cdr` or `second` to get the value, or even `assoc-value list key` from `Alexandria`.

```
(assoc :foo *my-alist*)
;; (:FOO . "foo")
(cdr *)
;; "foo"

(alexandria:assoc-value *my-alist* :foo)
;; "foo"
;; (:FOO . "FOO")
;; It actually returned 2 values.
```

There is `assoc-if`, and `rassoc` to get a cons cell by its value:

```
(rassoc "foo" *my-alist*)
;; NIL
;; bummer! The value "foo" is a string, so use:
(rassoc "foo" *my-alist* :test #'equal)
;; (:FOO . "foo")
```

If the alist has repeating (duplicate) keys, you can use `remove-if-not`, for example, to retrieve all of them.

```
(remove-if-not
  (lambda (entry)
      (eq :a entry))
```

```
  *alist-with-duplicate-keys*
  :key #'car)
```

Insert and remove entries

To add a key, we `push` another cons cell:

```
(push (cons 'team "team") *my-alist*)
;; => ((TEAM . "team") (FOO . "foo") (BAR . "bar"))
```

We can use `pop` and other functions that operate on lists, like `remove`:

```
(remove :team *my-alist*)
;; ((:TEAM . "team") (FOO . "foo") (BAR . "bar"))
;; => didn't remove anything
(remove :team *my-alist* :key 'car)
;; ((FOO . "foo") (BAR . "bar"))
;; => returns a copy
```

Remove only one element with `:count`:

```
(push (cons 'bar "bar2") *my-alist*)
;; ((BAR . "bar2") (TEAM . "team") (FOO . "foo") (BAR . "bar"))
;; => twice the 'bar key

(remove 'bar *my-alist* :key 'car :count 1)
;; ((TEAM . "team") (FOO . "foo") (BAR . "bar"))

;; because otherwise:
(remove 'bar *my-alist* :key 'car)
;; ((TEAM . "team") (FOO . "foo"))
;; => no more 'bar
```

Update entries

Replace a value:

```
*my-alist*
;; => '((:FOO . "foo") (:BAR . "bar"))
(assoc :foo *my-alist*)
```

```lisp
;; => (:FOO . "foo")
(setf (cdr (assoc :foo *my-alist*)) "new-value")
;; => "new-value"
*my-alist*
;; => '((:foo . "new-value") (:BAR . "bar"))
```

Replace a key:

```lisp
*my-alist*
;; => '((:FOO . "foo") (:BAR . "bar")))
(setf (car (assoc :bar *my-alist*)) :new-key)
;; => :NEW-KEY
*my-alist*
;; => '((:FOO . "foo") (:NEW-KEY . "bar")))
```

In the Alexandria library, see more functions like `hash-table-alist`, `alist-plist`,…

Plist

A property list is simply a list that alternates a key, a value, and so on, where its keys are symbols (we can not set its `:test`). More precisely, it first has a cons cell whose `car` is the key, whose `cdr` points to the following cons cell whose `car` is the value.

For example this plist:

```lisp
(defparameter my-plist (list 'foo "foo" 'bar "bar"))
```

looks like this:

```
[o|o]---[o|o]---[o|o]---[o|/]
 |       |       |       |
FOO     "foo"   BAR     "bar"
```

We access an element with `getf (list elt)` (it returns the value) (the list comes as first element),

we remove an element with `remf`.

```lisp
(defparameter my-plist (list 'foo "foo" 'bar "bar"))
;; => (FOO "foo" BAR "bar")
(setf (getf my-plist 'foo) "foo!!!")
;; => "foo!!!"
```

Structures

Structures offer a way to store data in named slots. They support single inheritance.

Classes provided by the Common Lisp Object System (CLOS) are more flexible however structures may offer better performance (see for example the SBCL manual).

Creation

Use `defstruct`:

```lisp
(defstruct person
   id name age)
```

At creation slots are optional and default to `nil`.

To set a default value:

```lisp
(defstruct person
   id
   (name "john doe")
   age)
```

Also specify the type after the default value:

```lisp
(defstruct person
  id
  (name "john doe" :type string)
  age)
```

We create an instance with the generated constructor `make-` + `<structure-name>`, so `make-person`:

```
(defparameter *me* (make-person))
*me*
#S(PERSON :ID NIL :NAME "john doe" :AGE NIL)
```

note that printed representations can be read back by the reader.

With a bad name type:

```
(defparameter *bad-name* (make-person :name 123))
```

```
Invalid initialization argument:
  :NAME
in call for class #<STRUCTURE-CLASS PERSON>.
   [Condition of type SB-PCL::INITARG-ERROR]
```

We can set the structure's constructor so as to create the structure without using keyword arguments, which can be more convenient sometimes. We give it a name and the order of the arguments:

```
(defstruct (person (:constructor create-person (id name age)))
     id
     name
     age)
```

Our new constructor is `create-person`:

```
(create-person 1 "me" 7)
#S(PERSON :ID 1 :NAME "me" :AGE 7)
```

However, the default `make-person` does *not* work any more:

```
(make-person :name "me")
;; debugger:
obsolete structure error for a structure of type PERSON
[Condition of type SB-PCL::OBSOLETE-STRUCTURE]
```

Slot access

We access the slots with accessors created by `<name-of-the-struct>-` + `slot-name`:

```
(person-name *me*)
;; "john doe"
```

we then also have `person-age` and `person-id`.

Setting

Slots are `setf`-able:

```
(setf (person-name *me*) "Cookbook author")
(person-name *me*)
;; "Cookbook author"
```

Predicate

A predicate function is generated:

```
(person-p *me*)
T
```

Single inheritance

Use single inheritance with the `:include <struct>` argument:

```
(defstruct (female (:include person))
     (gender "female" :type string))
(make-female :name "Lilie")
;; #S(FEMALE :ID NIL :NAME "Lilie" :AGE NIL :GENDER "female")
```

Note that the CLOS object system is more powerful.

Limitations

After a change, instances are not updated.

If we try to add a slot (`email` below), we have the choice to lose all instances, or to continue using the new definition of `person`. But the effects of redefining a structure are undefined by the standard, so it is best to re-compile and re-run the changed code.

```lisp
(defstruct person
       id
       (name "john doe" :type string)
       age
       email)
```

gives an error and we drop in the debugger:

```
attempt to redefine the STRUCTURE-OBJECT class PERSON
incompatibly with the current definition
   [Condition of type SIMPLE-ERROR]

Restarts:
 0: [CONTINUE] Use the new definition of PERSON, invalidating
already-loaded code and instances.
 1: [RECKLESSLY-CONTINUE] Use the new definition of PERSON as
if it were compatible, allowing old accessors to use new
instances and allowing new accessors to use old instances.
 2: [CLOBBER-IT] (deprecated synonym for RECKLESSLY-CONTINUE)
 3: [RETRY] Retry SLIME REPL evaluation request.
 4: [*ABORT] Return to SLIME's top level.
 5: [ABORT] abort thread (#<THREAD "repl-thread" RUNNING
{1002A0FFA3}>)
```

If we choose restart `0`, to use the new definition, we lose access to `*me*`:

```lisp
*me*
obsolete structure error for a structure of type PERSON
   [Condition of type SB-PCL::OBSOLETE-STRUCTURE]
```

There is also very little introspection. Portable Common Lisp does not define ways of finding out defined super/sub-structures nor what slots a structure has.

The Common Lisp Object System (which came after into the language) doesn’t have such limitations. See the [CLOS section](#).

- [structures on the hyperspec](#)
- David B. Lamkins, [“Successful Lisp, How to Understand and Use Common Lisp”](#).

Tree

`tree-equal`, `copy-tree`. They descend recursively into the car and the cdr of the cons cells they visit.

Sycamore - purely functional weight-balanced binary trees

https://github.com/ndantam/sycamore

Features:

- Fast, purely functional weight-balanced binary trees.
  - Leaf nodes are simple-vectors, greatly reducing tree height.
- Interfaces for tree Sets and Maps (dictionaries).
- Ropes
- Purely functional pairing heaps
- Purely functional amortized queue.

Controlling how much of data to print (`*print-length*`, `*print-level*`)

Use `*print-length*` and `*print-level*`.

They are both `nil` by default.

If you have a very big list, printing it on the REPL or in a stacktrace can take a long time and bring your editor or even your server down. Use `*print-length*` to choose the maximum of elements of the list to print, and to show there is a rest with a `...` placeholder:

```
(setf *print-length* 2)
(list :A :B :C :D :E)
;; (:A :B ...)
```

And if you have a very nested data structure, set `*print-level*` to choose the depth to print:

```
(let ((*print-level* 2))
  (print '(:a (:b (:c (:d :e))))))
;; (:A (:B #))             <= *print-level* in action
```

```
;; (:A (:B (:C (:D :E))))
;; => the list is returned,
;; the let binding is not in effect anymore.
```

`*print-length*` will be applied at each level.

Reference: the HyperSpec.

Appendix A - generic and nested access of alists, plists, hash-tables and CLOS slots

The solutions presented below might help you getting started, but keep in mind that they'll have a performance impact and that error messages will be less explicit.

- the access library (battle tested, used by the Djula templating system) has a generic `(access my-var :elt)` (blog post). It also has `accesses` (plural) to access and set nested values.
- rutils as a generic `generic-elt` or `?`,

Appendix B - accessing nested data structures

Sometimes we work with nested data structures, and we might want an easier way to access a nested element than intricated "getf" and "assoc" and all. Also, we might want to just be returned a `nil` when an intermediary key doesn't exist.

The `access` library given above provides this, with `(accesses var key1 key2…)`.

Strings

The most important thing to know about strings in Common Lisp is probably that they are arrays and thus also sequences. This implies that all concepts that are applicable to arrays and sequences also apply to strings. If you can't find a particular string function, make sure you've also searched for the more general array or sequence functions. We'll only cover a fraction of what can be done with and to strings here.

ASDF3, which is included with almost all Common Lisp implementations, includes Utilities for Implementation- and OS- Portability (UIOP), which defines functions to work on strings (`strcat`, `string-prefix-p`, `string-enclosed-p`, `first-char`, `last-char`, `split-string`, `stripln`).

Some external libraries available on Quicklisp bring some more functionality or some shorter ways to do.

- str defines `trim`, `words`, `unwords`, `lines`, `unlines`, `concat`, `split`, `shorten`, `repeat`, `replace-all`, `starts-with?`, `ends-with?`, `blankp`, `emptyp`, …
- Serapeum is a large set of utilities with many string manipulation functions.
- cl-change-case has functions to convert strings between camelCase, param-case, snake_case and more. They are also included into `str`.
- mk-string-metrics has functions to calculate various string metrics efficiently (Damerau-Levenshtein, Hamming, Jaro, Jaro-Winkler, Levenshtein, etc),
- and `cl-ppcre` can come in handy, for example `ppcre:replace-regexp-all`. See the regexp section.

Last but not least, when you'll need to tackle the `format` construct, don't miss the following resources:

- the official CLHS documentation
- a quick reference

- a CLHS summary on HexstreamSoft
- the list of all format directives at the end of this document.
- plus a Slime tip: type `C-c C-d ~` plus a letter of a format directive to open up its documentation. Use TAB-completion to list them all. Again more useful with `ivy-mode` or `helm-mode`.

Creating strings

A string is created with double quotes, all right, but we can recall these other ways:

- using `format nil` doesn't *print* but returns a new string (see more examples of `format` below):

```
(defparameter *person* "you")
(format nil "hello ~a" *person*) ;; => "hello you"
```

- `make-string count` creates a string of the given length. The `:initial-element` character is repeated `count` times:

```
(make-string 3 :initial-element #\♥) ;; => "♥♥♥"
```

Accessing Substrings

As a string is a sequence, you can access substrings with the SUBSEQ function. The index into the string is, as always, zero-based. The third, optional, argument is the index of the first character which is not a part of the substring, it is not the length of the substring.

```
(defparameter *my-string* (string "Groucho Marx"))
*MY-STRING*
(subseq *my-string* 8)
"Marx"
(subseq *my-string* 0 7)
"Groucho"
(subseq *my-string* 1 5)
"rouc"
```

You can also manipulate the substring if you use SUBSEQ together with SETF.

```
* (defparameter *my-string* (string "Harpo Marx"))
*MY-STRING*
* (subseq *my-string* 0 5)
"Harpo"
* (setf (subseq *my-string* 0 5) "Chico")
"Chico"
* *my-string*
"Chico Marx"
```

But note that the string isn't "stretchable". To cite from the HyperSpec: "If the subsequence and the new sequence are not of equal length, the shorter length determines the number of elements that are replaced." For example:

```
* (defparameter *my-string* (string "Karl Marx"))
*MY-STRING*
* (subseq *my-string* 0 4)
"Karl"
* (setf (subseq *my-string* 0 4) "Harpo")
"Harpo"
* *my-string*
"Harp Marx"
* (subseq *my-string* 4)
" Marx"
* (setf (subseq *my-string* 4) "o Marx")
"o Marx"
* *my-string*
"Harpo Mar"
```

Accessing Individual Characters

You can use the function CHAR to access individual characters of a string. CHAR can also be used in conjunction with SETF.

```
* (defparameter *my-string* (string "Groucho Marx"))
*MY-STRING*
* (char *my-string* 11)
#\x
* (char *my-string* 7)
```

```
#\Space
* (char *my-string* 6)
#\o
* (setf (char *my-string* 6) #\y)
#\y
* *my-string*
"Grouchy Marx"
```

Note that there's also SCHAR. If efficiency is important, SCHAR can be a bit faster where appropriate.

Because strings are arrays and thus sequences, you can also use the more generic functions AREF and ELT (which are more general while CHAR might be implemented more efficiently).

```
* (defparameter *my-string* (string "Groucho Marx"))
*MY-STRING*
* (aref *my-string* 3)
#\u
* (elt *my-string* 8)
#\M
```

Each character in a string has an integer code. The range of recognized codes and Lisp's ability to print them is directed related to your implementation's character set support, e.g. ISO-8859-1, or Unicode. Here are some examples in SBCL of UTF-8 which encodes characters as 1 to 4 8 bit bytes. The first example shows a character outside the first 128 chars, or what is considered the normal Latin character set. The second example shows a multibyte encoding (beyond the value 255). Notice the Lisp reader can round-trip characters by name.

```
* (stream-external-format *standard-output*)

:UTF-8
* (code-char 200)

#\LATIN_CAPITAL_LETTER_E_WITH_GRAVE
* (char-code #\LATIN_CAPITAL_LETTER_E_WITH_GRAVE)

200
```

```
* (code-char 2048)
#\SAMARITAN_LETTER_ALAF

* (char-code #\SAMARITAN_LETTER_ALAF)
2048
```

Check out the UTF-8 Wikipedia article for the range of supported characters and their encodings.

Remove or replace characters from a string

There’s a slew of (sequence) functions that can be used to manipulate a string and we’ll only provide some examples here. See the sequences dictionary in the HyperSpec for more.

`remove` one character from a string:

```
* (remove #\o "Harpo Marx")
"Harp Marx"
* (remove #\a "Harpo Marx")
"Hrpo Mrx"
* (remove #\a "Harpo Marx" :start 2)
"Harpo Mrx"
* (remove-if #'upper-case-p "Harpo Marx")
"arpo arx"
```

Replace one character with `substitute` (non destructive) or `replace` (destructive):

```
* (substitute #\u #\o "Groucho Marx")
"Gruuchu Marx"
* (substitute-if #\_ #'upper-case-p "Groucho Marx")
"_roucho _arx"
* (defparameter *my-string* (string "Zeppo Marx"))
*MY-STRING*
* (replace *my-string* "Harpo" :end1 5)
"Harpo Marx"
* *my-string*
"Harpo Marx"
```

Concatenating Strings

The name says it all: CONCATENATE is your friend. Note that this is a generic sequence function and you have to provide the result type as the first argument.

```
* (concatenate 'string "Karl" " " "Marx")
"Karl Marx"
* (concatenate 'list "Karl" " " "Marx")
(#\K #\a #\r #\l #\Space #\M #\a #\r #\x)
```

With UIOP, use `strcat`:

```
* (uiop:strcat "karl" " " marx")
```

or with the library `str`, use `concat`:

```
* (str:concat "foo" "bar")
```

If you have to construct a string out of many parts, all of these calls to CONCATENATE seem wasteful, though. There are at least three other good ways to construct a string piecemeal, depending on what exactly your data is. If you build your string one character at a time, make it an adjustable VECTOR (a one-dimensional ARRAY) of type character with a fill-pointer of zero, then use VECTOR-PUSH-EXTEND on it. That way, you can also give hints to the system if you can estimate how long the string will be. (See the optional third argument to VECTOR-PUSH-EXTEND.)

```
* (defparameter *my-string* (make-array 0
                              :element-type 'character
                              :fill-pointer 0
                              :adjustable t))
*MY-STRING*
* *my-string*
""
* (dolist (char '(#\Z #\a #\p #\p #\a))
    (vector-push-extend char *my-string*))
NIL
* *my-string*
"Zappa"
```

If the string will be constructed out of (the printed representations of) arbitrary objects, (symbols, numbers, characters, strings, …), you can use FORMAT with an output stream argument of NIL. This directs FORMAT to return the indicated output as a string.

```
* (format nil "This is a string with a list ~A in it"
          '(1 2 3))
"This is a string with a list (1 2 3) in it"
```

We can use the looping constructs of the FORMAT mini language to emulate CONCATENATE.

```
* (format nil "The Marx brothers are:~{ ~A~}."
          '("Groucho" "Harpo" "Chico" "Zeppo" "Karl"))
"The Marx brothers are: Groucho Harpo Chico Zeppo Karl."
```

FORMAT can do a lot more processing but it has a relatively arcane syntax. After this last example, you can find the details in the CLHS section about formatted output.

```
* (format nil "The Marx brothers are:~{ ~A~^,~}."
          '("Groucho" "Harpo" "Chico" "Zeppo" "Karl"))
"The Marx brothers are: Groucho, Harpo, Chico, Zeppo, Karl."
```

Another way to create a string out of the printed representation of various object is using WITH-OUTPUT-TO-STRING. The value of this handy macro is a string containing everything that was output to the string stream within the body to the macro. This means you also have the full power of FORMAT at your disposal, should you need it.

```
* (with-output-to-string (stream)
    (dolist (char '(#\Z #\a #\p #\p #\a #\, #\Space))
      (princ char stream))
    (format stream "~S - ~S" 1940 1993))
"Zappa, 1940 - 1993"
```

Processing a String One Character at a Time

Use the MAP function to process a string one character at a time.

```
* (defparameter *my-string* (string "Groucho Marx"))
*MY-STRING*
* (map 'string (lambda (c) (print c)) *my-string*)
#\G
#\r
#\o
#\u
#\c
#\h
#\o
#\Space
#\M
#\a
#\r
#\x
"Groucho Marx"
```

Or do it with LOOP.

```
* (loop for char across "Zeppo"
        collect char)
(#\Z #\e #\p #\p #\o)
```

Reversing a String by Word or Character

Reversing a string by character is easy using the built-in REVERSE function (or its destructive counterpart NREVERSE).

```
*(defparameter *my-string* (string "DSL"))
*MY-STRING*
* (reverse *my-string*)
"LSD"
```

There’s no one-liner in CL to reverse a string by word (like you would do it in Perl with split and join). You either have to use functions from an external library like SPLIT-SEQUENCE or you have to roll your own solution.

Here’s an attempt with the `str` library:

```
* (defparameter *singing* "singing in the rain")
*SINGING*
* (str:words *SINGING*)
("singing" "in" "the" "rain")
* (reverse *)
("rain" "the" "in" "singing")
* (str:unwords *)
"rain the in singing"
```

And here’s another one with no external dependencies:

```
* (defun split-by-one-space (string)
    "Returns a list of substrings of string
    divided by ONE space each.
    Note: Two consecutive spaces will be seen as
    if there were an empty string between them."
    (loop for i = 0 then (1+ j)
          as j = (position #\Space string :start i)
          collect (subseq string i j)
          while j))
SPLIT-BY-ONE-SPACE
* (split-by-one-space "Singing in the rain")
("Singing" "in" "the" "rain")
* (split-by-one-space "Singing in the  rain")
("Singing" "in" "the" "" "rain")
* (split-by-one-space "Cool")
("Cool")
* (split-by-one-space " Cool ")
("" "Cool" "")
* (defun join-string-list (string-list)
    "Concatenates a list of strings
and puts spaces between the elements."
    (format nil "~{~A~^ ~}" string-list))
JOIN-STRING-LIST
* (join-string-list '("We" "want" "better" "examples"))
"We want better examples"
* (join-string-list '("Really"))
"Really"
* (join-string-list '())
""
* (join-string-list
   (nreverse
    (split-by-one-space
```

```
    "Reverse this sentence by word")))
"word by sentence this Reverse"
```

Dealing with unicode strings

We'll use here SBCL's string operations. More generally, see SBCL's unicode support.

Sorting unicode strings alphabetically

Sorting unicode strings with `string-lessp` as the comparison function isn't satisfying:

```
(sort '("Aaa" "Éée" "Zzz") #'string-lessp)
;; ("Aaa" "Zzz" "Éée")
```

With SBCL, use `sb-unicode:unicode<`:

```
(sort '("Aaa" "Éée" "Zzz") #'sb-unicode:unicode<)
;; ("Aaa" "Éée" "Zzz")
```

Breaking strings into graphenes, sentences, lines and words

These functions use SBCL's `sb-unicode`: they are SBCL specific.

Use `sb-unicode:sentences` to break a string into sentences according to the default sentence breaking rules.

Use `sb-unicode:lines` to break a string into lines that are no wider than the `:margin` keyword argument. Combining marks will always be kept together with their base characters, and spaces (but not other types of whitespace) will be removed from the end of lines. If `:margin` is unspecified, it defaults to 80 characters

```
(sb-unicode:lines "A first sentence. A second somewhat long one.
;; => ("A first"
       "sentence."
       "A second"
       "somewhat"
```

```
       "long one.")
```

See also `sb-unicode:words` and `sb-unicode:graphenes`.

Tip: you can ensure these functions are run only in SBCL with a feature flag:

```
#+sbcl
(runs on sbcl)
#-sbcl
(runs on other implementations)
```

Controlling Case

Common Lisp has a couple of functions to control the case of a string.

```
* (string-upcase "cool")
"COOL"
* (string-upcase "Cool")
"COOL"
* (string-downcase "COOL")
"cool"
* (string-downcase "Cool")
"cool"
* (string-capitalize "cool")
"Cool"
* (string-capitalize "cool example")
"Cool Example"
```

These functions take the `:start` and `:end` keyword arguments so you can optionally only manipulate a part of the string. They also have destructive counterparts whose names starts with “N”.

```
* (string-capitalize "cool example" :start 5)
"cool Example"
* (string-capitalize "cool example" :end 5)
"Cool example"
* (defparameter *my-string* (string "BIG"))
*MY-STRING*
* (defparameter *my-downcase-string* (nstring-downcase *my-strir
*MY-DOWNCASE-STRING*
```

```
* *my-downcase-string*
"big"
* *my-string*
"big"
```

Note this potential caveat: according to the HyperSpec,

> for STRING-UPCASE, STRING-DOWNCASE, and STRING-CAPITALIZE, string is not modified. However, if no characters in string require conversion, the result may be either string or a copy of it, at the implementation's discretion.

This implies that the last result in the following example is implementation-dependent - it may either be "BIG" or "BUG". If you want to be sure, use COPY-SEQ.

```
* (defparameter *my-string* (string "BIG"))
*MY-STRING*
* (defparameter *my-upcase-string* (string-upcase *my-string*))
*MY-UPCASE-STRING*
* (setf (char *my-string* 1) #\U)
#\U
* *my-string*
"BUG"
* *my-upcase-string*
"BIG"
```

With the format function

The format function has directives to change the case of words:

To lower case: ~(~)

```
(format t "~(~a~)" "HELLO WORLD")
;; => hello world
```

Capitalize every word: ~:(~)

```
(format t "~:(~a~)" "HELLO WORLD")
Hello World
NIL
```

Capitalize the first word: ~@(~)

```
(format t "~@(~a~)" "hello world")
Hello world
NIL
```

To upper case: ~@:(~)

Where we re-use the colon and the @:

```
(format t "~@:(~a~)" "hello world")
HELLO WORLD
NIL
```

Trimming Blanks from the Ends of a String

Not only can you trim blanks, but you can get rid of arbitrary characters. The functions STRING-TRIM, STRING-LEFT-TRIM and STRING-RIGHT-TRIM return a substring of their second argument where all characters that are in the first argument are removed off the beginning and/or the end. The first argument can be any sequence of characters.

```
* (string-trim " " " trim me ")
"trim me"
* (string-trim " et" " trim me ")
"rim m"
* (string-left-trim " et" " trim me ")
"rim me "
* (string-right-trim " et" " trim me ")
" trim m"
* (string-right-trim '(#\Space #\e #\t) " trim me ")
" trim m"
* (string-right-trim '(#\Space #\e #\t #\m) " trim me ")
```

Note: The caveat mentioned in the section about Controlling Case also applies here.

Converting between Symbols and Strings

The function INTERN will “convert” a string to a symbol. Actually, it will check whether the symbol denoted by the string (its first argument) is already accessible in the package (its second, optional, argument which defaults to the current package) and enter it, if necessary, into this package. It is beyond the scope of this chapter to explain all the concepts involved and to address the second return value of this function. See the CLHS chapter about packages for details.

Note that the case of the string is relevant.

```
* (in-package "COMMON-LISP-USER")
#<The COMMON-LISP-USER package, 35/44 internal, 0/9 external>
* (intern "MY-SYMBOL")
MY-SYMBOL
NIL
* (intern "MY-SYMBOL")
MY-SYMBOL
:INTERNAL
* (export 'MY-SYMBOL)
T
* (intern "MY-SYMBOL")
MY-SYMBOL
:EXTERNAL
* (intern "My-Symbol")
|My-Symbol|
NIL
* (intern "MY-SYMBOL" "KEYWORD")
:MY-SYMBOL
NIL
* (intern "MY-SYMBOL" "KEYWORD")
:MY-SYMBOL
:EXTERNAL
```

To do the opposite, convert from a symbol to a string, use SYMBOL-NAME or STRING.

```
* (symbol-name 'MY-SYMBOL)
"MY-SYMBOL"
* (symbol-name 'my-symbol)
"MY-SYMBOL"
* (symbol-name '|my-symbol|)
"my-symbol"
* (string 'howdy)
"HOWDY"
```

Converting between Characters and Strings

You can use COERCE to convert a string of length 1 to a character. You can also use COERCE to convert any sequence of characters into a string. You can not use COERCE to convert a character to a string, though - you’ll have to use STRING instead.

```
* (coerce "a" 'character)
#\a
* (coerce (subseq "cool" 2 3) 'character)
#\o
* (coerce "cool" 'list)
(#\c #\o #\o #\l)
* (coerce '(#\h #\e #\y) 'string)
"hey"
* (coerce (nth 2 '(#\h #\e #\y)) 'character)
#\y
* (defparameter *my-array* (make-array 5 :initial-element #\x))
*MY-ARRAY*
* *my-array*
#(#\x #\x #\x #\x #\x)
* (coerce *my-array* 'string)
"xxxxx"
* (string 'howdy)
"HOWDY"
* (string #\y)
"y"
* (coerce #\y 'string)
#\y can't be converted to type STRING.
   [Condition of type SIMPLE-TYPE-ERROR]
```

Finding an Element of a String

Use `find`, `position`, and their `…-if` counterparts to find characters in a string, with the appropriate `:test` parameter:

```
(find #\t "Tea time." :test #'equal)
#\t
* (find #\t "Tea time." :test #'equalp)
#\T
* (find #\z "Tea time." :test #'equalp)
NIL
* (find-if #'digit-char-p "Tea time.")
#\1
* (find-if #'digit-char-p "Tea time." :from-end t)
#\0

(position #\t "Tea time." :test #'equal)
4   ;; <= the first lowercase t
(position #\t "Tea time." :test #'equalp)
0    ;; <= the first capital T
(position-if #'digit-char-p "Tea time is at 5'00.")
15
(position-if #'digit-char-p "Tea time is at 5'00." :from-end t)
18
```

Or use `count` and friends to count characters in a string:

```
(count #\t "Tea time." :test #'equal)
1  ;; <= equal ignores the capital T
(count #\t "Tea time." :test #'equalp)
2  ;; <= equalp counts the capital T
(count-if #'digit-char-p "Tea time is at 5'00.")
3
(count-if #'digit-char-p "Tea time is at 5'00." :start 18)
1
```

Finding a Substring of a String

The function `search` can find substrings of a string.

```
* (search "we" "If we can't be free we can at least be cheap")
```

```
* (search "we" "If we can't be free we can at least be cheap")
3
* (search "we" "If we can't be free we can at least be cheap"
          :from-end t)
20
* (search "we" "If we can't be free we can at least be cheap"
          :start2 4)
20
* (search "we" "If we can't be free we can at least be cheap"
          :end2 5 :from-end t)
3
* (search "FREE" "If we can't be free we can at least be cheap")
NIL
* (search "FREE" "If we can't be free we can at least be cheap"
          :test #'char-equal)
15
```

Converting a String to a Number

To an integer: parse-integer

CL provides the `parse-integer` function to convert a string representation of an integer to the corresponding numeric value. The second return value is the index into the string where the parsing stopped.

```
(parse-integer "42")
42
2
(parse-integer "42" :start 1)
2
2
(parse-integer "42" :end 1)
4
1
(parse-integer "42" :radix 8)
34
2
(parse-integer " 42 ")
42
3
(parse-integer " 42 is forty-two" :junk-allowed t)
```

```
42
3
(parse-integer " 42 is forty-two")

Error in function PARSE-INTEGER:
   There's junk in this string: " 42 is forty-two".
```

`parse-integer` doesn't understand radix specifiers like `#X`, nor is there a built-in function to parse other numeric types. You could use `read-from-string` in this case.

Extracting many integers from a string: `ppcre:all-matches-as-strings`

We show this in the Regular Expressions chapter but while we are on this topic, you can find it super useful:

```
* (ppcre:all-matches-as-strings "-?\\d+" "42 is 41 plus 1")
;; ("42" "41" "1")

* (mapcar #'parse-integer *)
;; (42 41 1)
```

To any number: read-from-string

Be aware that the full reader is in effect if you're using this function. This can lead to vulnerability issues. You should use a library like `parse-number` or `parse-float` instead.

```
(read-from-string "#X23")
35
4
(read-from-string "4.5")
4.5
3
(read-from-string "6/8")
3/4
3
(read-from-string "#C(6/8 1)")
#C(3/4 1)
```

```
9
(read-from-string "1.2e2")
120.00001
5
(read-from-string "symbol")
SYMBOL
6
(defparameter *foo* 42)
*FOO*
(read-from-string "#.(setq *foo* \"gotcha\")")
"gotcha"
23
*foo*
"gotcha"
```

To a float: the parse-float library

There is no built-in function similar to `parse-integer` to parse other number types. The external library parse-float does exactly that. It doesn't use `read-from-string` so it is safe to use.

```
(ql:quickload "parse-float")
(parse-float:parse-float "1.2e2")
;; 120.00001
;; 5
```

LispWorks also has a parse-float function.

See also parse-number.

Converting a Number to a String

The general function WRITE-TO-STRING or one of its simpler variants PRIN1-TO-STRING or PRINC-TO-STRING may be used to convert a number to a string. With WRITE-TO-STRING, the :base keyword argument may be used to change the output base for a single call. To change the output base globally, set *print-base* which defaults to 10. Remember in Lisp, rational numbers are represented as quotients of two integers even when converted to strings.

```
(write-to-string 250)
"250"
(write-to-string 250.02)
"250.02"
(write-to-string 250 :base 5)
"2000"
(write-to-string (/ 1 3))
"1/3"
*
```

Comparing Strings

The general functions EQUAL and EQUALP can be used to test whether two strings are equal. The strings are compared element-by-element, either in a case-sensitive manner (EQUAL) or not (EQUALP). There's also a bunch of string-specific comparison functions. You'll want to use these if you're deploying implementation-defined attributes of characters. Check your vendor's documentation in this case.

Here are a few examples. Note that all functions that test for inequality return the position of the first mismatch as a generalized boolean. You can also use the generic sequence function MISMATCH if you need more versatility.

```
(string= "Marx" "Marx")
T
(string= "Marx" "marx")
NIL
(string-equal "Marx" "marx")
T
(string< "Groucho" "Zeppo")
0
(string< "groucho" "Zeppo")
NIL
(string-lessp "groucho" "Zeppo")
0
(mismatch "Harpo Marx" "Zeppo Marx" :from-end t :test #'char=)
3
```

String formatting

The `format` function has a lot of directives to print strings, numbers, lists, going recursively, even calling Lisp functions, etc. We'll focus here on a few things to print and format strings.

The need of our examples arise when we want to print many strings and justify them. Let's work with this list of movies:

```
(defparameter movies '(
    (1 "Matrix" 5)
    (10 "Matrix Trilogy swe sub" 3.3)
    ))
```

We want an aligned and justified result like this:

```
 1 Matrix                  5
10 Matrix Trilogy swe sub  3.3
```

We'll use `mapcar` to iterate over our movies and experiment with the format constructs.

```
(mapcar (lambda (it)
          (format t "~a ~a ~a~%" (first it) (second it) (third i
        movies)
```

which prints:

```
1 Matrix 5
10 Matrix Trilogy swe sub 3.3
```

Structure of format

Format directives start with `~`. A final character like `A` or `a` (they are case insensitive) defines the directive. In between, it can accept coma-separated options and parameters.

Print a tilde with `~~`, or 10 with `~10~`.

Other directives include:

- `R`: Roman (e.g., prints in English): `(format t "~R" 20)` => “twenty”.
- `$`: monetary: `(format t "~$" 21982)` => 21982.00
- `D`, `B`, `O`, `X`: Decimal, Binary, Octal, Hexadecimal.
- `F`: fixed-format Floating point.
- `P`: plural: `(format nil "~D famil~:@P/~D famil~:@P" 7 1)` => “7 families/1 family”

Basic primitive: ~A or ~a (Aesthetics)

`(format t "~a" movies)` is the most basic primitive.

```
(format nil "~a" movies)
;; => "((1 Matrix 5) (10 Matrix Trilogy swe sub 3.3))"
```

Newlines: ~% and ~&

`~%` is the newline character. `~10%` prints 10 newlines.

`~&` does not print a newline if the output stream is already at one.

Tabs

with `~T`. Also `~10T` works.

Also `i` for indentation.

Justifying text / add padding on the right

Use a number as parameter, like `~2a`:

```
(format nil "~20a" "yo")
;; "yo                  "

(mapcar (lambda (it)
           (format t "~2a ~a ~a~%" (first it) (second it) (thirc
         movies)
```

```
1  Matrix 5
10 Matrix Trilogy swe sub 3.3
```

So, expanding:

```
(mapcar (lambda (it)
          (format t "~2a ~25a ~2a~%" (first it) (second it) (thi
        movies)
```

```
1  Matrix                     5
10 Matrix Trilogy swe sub    3.3
```

text is justified on the right (this would be with option :).

Justifying on the left: @

Use a `@` as in `~2@A`:

```
(format nil "~20@a" "yo")
;; "                  yo"

(mapcar (lambda (it)
           (format nil "~2@a ~25@a ~2a~%" (first it) (second it)
        movies)
```

```
 1                    Matrix 5
10    Matrix Trilogy swe sub 3.3
```

Justifying decimals

In `~,2F`, 2 is the number of decimals and F the floats directive: `(format t "~,2F" 20.1)` => "20.10".

With `~2,2f`:

```
(mapcar (lambda (it)
          (format t "~2@a ~25a ~2,2f~%" (first it) (second it) (
        movies)
```

```
 1 Matrix                    5.00
10 Matrix Trilogy swe sub    3.30
```

And we're happy with this result.

Iteration

Create a string from a list with iteration construct `~{str~}`:

```
(format nil "~{~A, ~}" '(a b c))
;; "A, B, C, "
```

using `~^` to avoid printing the comma and space after the last element:

```
(format nil "~{~A~^, ~}" '(a b c))
;; "A, B, C"
```

`~:{str~}` is similar but for a list of sublists:

```
(format nil "~:{~S are ~S. ~}" '((pigeons birds) (dogs mammals))
;; "PIGEONS are BIRDS. DOGS are MAMMALS. "
```

`~@{str~}` is similar to `~{str~}`, but instead of using one argument that is a list, all the remaining arguments are used as the list of arguments for the iteration:

```
(format nil "~@{~S are ~S. ~}" 'pigeons 'birds 'dogs 'mammals)
;; "PIGEONS are BIRDS. DOGS are MAMMALS. "
```

Formatting a format string (~v, ~?)

Sometimes you want to justify a string, but the length is a variable itself. You can't hardcode its value as in `(format nil "~30a" "foo")`. Enters the `v` directive. We can use it in place of the comma-separated prefix parameters:

```
(let ((padding 30))
    (format nil "~va" padding "foo"))
;; "foo                           "
```

Other times, you would like to insert a complete format directive at run time. Enters the `?` directive.

```
(format nil "~?" "~30a" '("foo"))
;;                        ^ a list
```

or, using `~@?`:

```
(format nil "~@?" "~30a" "foo" )
;;                        ^ not a list
```

Of course, it is always possible to format a format string beforehand:

```
(let* ((length 30)
      (directive (format nil "~~~aa" length)))
 (format nil directive "foo"))
```

Conditional Formatting

Choose one value out of many options by specifying a number:

```
(format nil "~[dog~;cat~;bird~:;default~]" 0)
;; "dog"

(format nil "~[dog~;cat~;bird~:;default~]" 1)
;; "cat"
```

If the number is out of range, the default option (after `~:;`) is returned:

```
(format nil "~[dog~;cat~;bird~:;default~]" 9)
;; "default"
```

Combine it with `~:*` to implement irregular plural:

```
(format nil "I saw ~r el~:*~[ves~;f~:;ves~]." 0)
;; => "I saw zero elves."
(format nil "I saw ~r el~:*~[ves~;f~:;ves~]." 1)
;; => "I saw one elf."
(format nil "I saw ~r el~:*~[ves~;f~:;ves~]." 2)
;; => "I saw two elves."
```

Capturing what is is printed into a stream

Inside `(with-output-to-string (mystream) …)`, everything that is printed into the stream `mystream` is captured and returned as a string:

```
(defun greet (name &key (stream t))
   ;; by default, print to standard output.
   (format stream "hello ~a" name))

(let ((output (with-output-to-string (stream)
                (greet "you" :stream stream))))
   (format t "Output is: '~a'. It is indeed a ~a, aka a string.~
;; Output is: 'hello you'. It is indeed a (SIMPLE-ARRAY CHARACTE
;; NIL
```

Cleaning up strings

The following examples use the cl-slug library which, internally, iterates over the characters of the string and uses `ppcre:regex-replace-all`.

```
(ql:quickload "cl-slug")
```

Then it can be used with the `slug` prefix.

Its main function is to transform a string to a slug, suitable for a website's url:

```
(slug:slugify "My new cool article, for the blog (V. 2).")
;; "my-new-cool-article-for-the-blog-v-2"
```

Removing accentuated letters

Use `slug:asciify` to replace accentuated letters by their ascii equivalent:

```
(slug:asciify "ñ é ß ğ ö")
;; => "n e ss g o"
```

This function supports many (western) languages:

```
slug:*available-languages*
((:TR . "Türkçe (Turkish)") (:SV . "Svenska (Swedish)") (:FI .
"Suomi (Finnish)")
 (:UK . "українська (Ukrainian)") (:RU . "Ру́сский (Russian)")
(:RO . "Română (Romanian)")
 (:RM . "Rumàntsch (Romansh)") (:PT . "Português (Portuguese)")
(:PL . "Polski (Polish)")
 (:NO . "Norsk (Norwegian)") (:LT . "Lietuvių (Lithuanian)")
(:LV . "Latviešu (Latvian)")
 (:LA . "Lingua Latīna (Latin)") (:IT . "Italiano (Italian)")
(:EL . "ελληνικά (Greek)")
 (:FR . "Français (French)") (:EO . "Esperanto") (:ES .
"Español (Spanish)") (:EN . "English")
 (:DE . "Deutsch (German)") (:DA . "Dansk (Danish)") (:CS .
"Čeština (Czech)")
 (:CURRENCY . "Currency"))
```

Removing punctuation

Use `(str:remove-punctuation s)` or `(str:no-case s)` (same as `(cl-change-case:no-case s)`):

```
(str:remove-punctuation "HEY! What's up ??")
;; "HEY What s up"

(str:no-case "HEY! What's up ??")
;; "hey what s up"
```

They strip the punctuation with one ppcre unicode regexp (`(ppcre:regex-replace-all "[^\\p{L}\\p{N}]+"` where `p{L}` is the “letter” category and `p{N}` any kind of numeric character).

Appendix

All format directives

All directives are case-insensivite: `~A` is the same as `~a`.

```
$ - Monetary Floating-Point
% - Newline
& - Fresh-line
( - Case Conversion
```

```
) - End of Case Conversion
* - Go-To
/ - Call Function
; - Clause Separator
< - Justification
< - Logical Block
> - End of Justification
? - Recursive Processing
A - Aesthetic
B - Binary
C - Character
D - Decimal
E - Exponential Floating-Point
F - Fixed-Format Floating-Point
G - General Floating-Point
I - Indent
Missing and Additional FORMAT Arguments
Nesting of FORMAT Operations
Newline: Ignored Newline
O - Octal
P - Plural
R - Radix
S - Standard
T - Tabulate
W - Write
X - Hexadecimal
[ - Conditional Expression
] - End of Conditional Expression
^ - Escape Upward
_ - Conditional Newline
{ - Iteration
| - Page
} - End of Iteration
~ - Tilde
```

See also

- Pretty_printing_table_data, in ASCII art, a tutorial as a Jupyter notebook.

Numbers

Common Lisp has a rich set of numerical types, including integer, rational, floating point, and complex.

Some sources:

- `Numbers` in Common Lisp the Language, 2nd Edition
- `Numbers, Characters and Strings` in Practical Common Lisp

Introduction

Integer types

Common Lisp provides a true integer type, called `bignum`, limited only by the total memory available (not the machine word size). For example this would overflow a 64 bit integer by some way:

```
* (expt 2 200)
1606938044258990275541962092341162602522202993782792835301376
```

For efficiency, integers can be limited to a fixed number of bits, called a `fixnum` type. The range of integers which can be represented is given by:

```
* most-positive-fixnum
4611686018427387903
* most-negative-fixnum
-4611686018427387904
```

Functions which operate on or evaluate to integers include:

- `isqrt`, which returns the greatest integer less than or equal to the exact positive square root of natural.

```
* (isqrt 10)
3
```

```
* (isqrt 4)
2
```

- `gcd` to find the Greatest Common Denominator
- `lcm` for the Least Common Multiple.

Like other low-level programming languages, Common Lisp provides literal representation for hexadecimals and other radixes up to 36. For example:

```
* #xFF
255
* #2r1010
10
* #4r33
15
* #8r11
9
* #16rFF
255
* #36rz
35
```

Rational types

Rational numbers of type `ratio` consist of two `bignums`, the numerator and denominator. Both can therefore be arbitrarily large:

```
* (/ (1+ (expt 2 100)) (expt 2 100))
1267650600228229401496703205377/1267650600228229401496703205376
```

It is a subtype of the `rational` class, along with `integer`.

Floating point types

See Common Lisp the Language, 2nd Edition, section 2.1.3.

Floating point types attempt to represent the continuous real numbers using a finite number of bits. This means that many real numbers cannot be represented, but are approximated. This can lead to some nasty surprises, particularly when converting between base-10 and the base-2 internal representation. If you are working with floating point numbers then reading What Every Computer Scientist Should Know About Floating-Point Arithmetic is highly recommended.

The Common Lisp standard allows for several floating point types. In order of increasing precision these are: `short-float`, `single-float`, `double-float`, and `long-float`. Their precisions are implementation dependent, and it is possible for an implementation to have only one floating point precision for all types.

The constants `short-float-epsilon`, `single-float-epsilon`, `double-float-epsilon` and `long-float-epsilon` give a measure of the precision of the floating point types, and are implementation dependent.

Floating point literals

When reading floating point numbers, the default type is set by the special variable `*read-default-float-format*`. By default this is `SINGLE-FLOAT`, so if you want to ensure that a number is read as double precision then put a `d0` suffix at the end

```
* (type-of 1.24)
SINGLE-FLOAT

* (type-of 1.24d0)
DOUBLE-FLOAT
```

Other suffixes are `s` (short), `f` (single float), `d` (double float), `l` (long float) and `e` (default; usually single float).

The default type can be changed, but note that this may break packages which assume `single-float` type.

```
* (setq *read-default-float-format* 'double-float)
* (type-of 1.24)
DOUBLE-FLOAT
```

Note that unlike in some languages, appending a single decimal point to the end of a number does not make it a float:

```
* (type-of 10.)
(INTEGER 0 4611686018427387903)

* (type-of 10.0)
SINGLE-FLOAT
```

Floating point errors

If the result of a floating point calculation is too large then a floating point overflow occurs. By default in SBCL (and other implementations) this results in an error condition:

```
* (exp 1000)
; Evaluation aborted on #<FLOATING-POINT-OVERFLOW {10041720B3}>.
```

The error can be caught and handled, or this behaviour can be changed, to return `+infinity`. In SBCL this is:

```
* (sb-int:set-floating-point-modes :traps '(:INVALID :DIVIDE-BY-

* (exp 1000)
#.SB-EXT:SINGLE-FLOAT-POSITIVE-INFINITY


* (/ 1 (exp 1000))
0.0
```

The calculation now silently continues, without an error condition.

A similar functionality to disable floating overflow errors exists in CCL:

```
* (set-fpu-mode :overflow nil)
```

In SBCL the floating point modes can be inspected:

```
* (sb-int:get-floating-point-modes)
```

```
  (sb-int:get-floating-point-modes)
(:TRAPS (:OVERFLOW :INVALID :DIVIDE-BY-ZERO) :ROUNDING-MODE :NEA
 :CURRENT-EXCEPTIONS NIL :ACCRUED-EXCEPTIONS NIL :FAST-MODE NIL)
```

Arbitrary precision

For arbitrary high precision calculations there is the computable-reals library on QuickLisp:

```
* (ql:quickload :computable-reals)
* (use-package :computable-reals)

* (sqrt-r 2)
+1.41421356237309504880...

* (sin-r (/r +pi-r+ 2))
+1.00000000000000000000...
```

The precision to print is set by `*PRINT-PREC*`, by default 20

```
* (setq *PRINT-PREC* 50)
* (sqrt-r 2)
+1.41421356237309504880168872420969807856967187537695...
```

Complex types

There are 5 types of complex number: The real and imaginary parts must be of the same type, and can be rational, or one of the floating point types (short, single, double or long).

Complex values can be created using the `#C` reader macro or the function [`complex`](complex). The reader macro does not allow the use of expressions as real and imaginary parts:

```
* #C(1 1)
#C(1 1)

* #C((+ 1 2) 5)
; Evaluation aborted on #<TYPE-ERROR expected-type: REAL datum:
```

```
* (complex (+ 1 2) 5)
#C(3 5)
```

If constructed with mixed types then the higher precision type will be used for both parts.

```
* (type-of #C(1 1))
(COMPLEX (INTEGER 1 1))

* (type-of #C(1.0 1))
(COMPLEX (SINGLE-FLOAT 1.0 1.0))

* (type-of #C(1.0 1d0))
(COMPLEX (DOUBLE-FLOAT 1.0d0 1.0d0))
```

The real and imaginary parts of a complex number can be extracted using realpart and imagpart:

```
* (realpart #C(7 9))
7
* (imagpart #C(4.2 9.5))
9.5
```

Complex arithmetic

Common Lisp's mathematical functions generally handle complex numbers, and return complex numbers when this is the true result. For example:

```
* (sqrt -1)
#C(0.0 1.0)

* (exp #C(0.0 0.5))
#C(0.87758255 0.47942555)

* (sin #C(1.0 1.0))
#C(1.2984576 0.63496387)
```

Reading numbers from strings

The `parse-integer` function reads an integer from a string.

The parse-float library provides a parser which cannot evaluate arbitrary expressions, so should be safer to use on untrusted input:

```
* (ql:quickload :parse-float)
* (use-package :parse-float)

* (parse-float "23.4e2" :type 'double-float)
2340.0d0
6
```

See the strings section on converting between strings and numbers.

Converting numbers

Most numerical functions automatically convert types as needed. The `coerce` function converts objects from one type to another, including numeric types.

See Common Lisp the Language, 2nd Edition, section 12.6.

Convert float to rational

The `rational` and `rationalize` functions convert a real numeric argument into a rational. `rational` assumes that floating point arguments are exact; `rationalize` exploits the fact that floating point numbers are only exact to their precision, so can often find a simpler rational number.

Convert rational to integer

If the result of a calculation is a rational number where the numerator is a multiple of the denominator, then it is automatically converted to an integer:

```
* (type-of (* 1/2 4))
(INTEGER 0 4611686018427387903)
```

Rounding floating-point and rational numbers

The `ceiling`, `floor`, `round` and `truncate` functions convert floating point or rational numbers to integers. The difference between the result and the input is returned as the second value, so that the input is the sum of the two outputs.

```
* (ceiling 1.42)
2
-0.58000004

* (floor 1.42)
1
0.41999996

* (round 1.42)
1
0.41999996

* (truncate 1.42)
1
0.41999996
```

There is a difference between `floor` and `truncate` for negative numbers:

```
* (truncate -1.42)
-1
-0.41999996

* (floor -1.42)
-2
0.58000004

* (ceiling -1.42)
-1
-0.41999996
```

Similar functions `fceiling`, `ffloor`, `fround` and `ftruncate` return the result as floating point, of the same type as their argument:

```
* (ftruncate 1.3)
1.0
0.29999995

* (type-of (ftruncate 1.3))
SINGLE-FLOAT

* (type-of (ftruncate 1.3d0))
DOUBLE-FLOAT
```

Comparing numbers

See [Common Lisp the Language, 2nd Edition, Section 12.3](#).

The `=` predicate returns `T` if all arguments are numerically equal. Note that comparison of floating point numbers includes some margin for error, due to the fact that they cannot represent all real numbers and accumulate errors.

The constant [`single-float-epsilon`](#) is the smallest number which will cause an = comparison to fail, if it is added to 1.0:

```
* (= (+ 1s0 5e-8) 1s0)
T
* (= (+ 1s0 6e-8) 1s0)
NIL
```

Note that this does not mean that a `single-float` is always precise to within `6e-8`:

```
* (= (+ 10s0 4e-7) 10s0)
T
* (= (+ 10s0 5e-7) 10s0)
NIL
```

Instead this means that `single-float` is precise to approximately seven digits. If a sequence of calculations are performed, then error can accumulate and a larger error margin may be needed. In this case the absolute difference can be compared:

```
* (< (abs (- (+ 10s0 5e-7)
             10s0))
     1s-6)
T
```

When comparing numbers with `=` mixed types are allowed. To test both numerical value and type use `eql`:

```
* (= 3 3.0)
T

* (eql 3 3.0)
NIL
```

Operating on a series of numbers

Many Common Lisp functions operate on sequences, which can be either lists or vectors (1D arrays). See the section on mapping.

Operations on multidimensional arrays are discussed in this section.

Libraries are available for defining and operating on lazy sequences, including "infinite" sequences of numbers. For example

- Clazy which is on QuickLisp.
- folio2 on QuickLisp. Includes an interface to the
- Series package for efficient sequences.
- lazy-seq.

Working with Roman numerals

The `format` function can convert numbers to roman numerals with the `~@r` directive:

```
* (format nil "~@r" 42)
"XLII"
```

There is a gist by tormaroe for reading roman numerals.

Generating random numbers

The `random` function generates either integer or floating point random numbers, depending on the type of its argument.

```
* (random 10)
7

* (type-of (random 10))
(INTEGER 0 4611686018427387903)
* (type-of (random 10.0))
SINGLE-FLOAT
* (type-of (random 10d0))
DOUBLE-FLOAT
```

In SBCL a Mersenne Twister pseudo-random number generator is used. See section 7.13 of the SBCL manual for details.

The random seed is stored in `*random-state*` whose internal representation is implementation dependent. The function `make-random-state` can be used to make new random states, or copy existing states.

To use the same set of random numbers multiple times, `(make-random-state nil)` makes a copy of the current `*random-state*`:

```
* (dotimes (i 3)
    (let ((*random-state* (make-random-state nil)))
      (format t "~a~%"
              (loop for i from 0 below 10 collecting (random 10)

(8 3 9 2 1 8 0 0 4 1)
(8 3 9 2 1 8 0 0 4 1)
(8 3 9 2 1 8 0 0 4 1)
```

This generates 10 random numbers in a loop, but each time the sequence is the same because the `*random-state*` special variable is dynamically bound to a copy of its state before the `let` form.

Other resources:

- The random-state package is available on QuickLisp, and provides a number of portable random number generators.

Bit-wise Operation

Common Lisp also provides many functions to perform bit-wise arithmetic operations. Some commonly used ones are listed below, together with their C/C++ equivalence.

Common Lisp	**C/C++**	**Description**
(logand a b c)	a & b & c	Bit-wise AND of multiple operands
(logior a b c)	a | b | c	Bit-wise OR of multiple operands
(lognot a)	~a	Bit-wise NOT of single operands
(logxor a b c)	a ^ b ^ c	Bit-wise exclusive or (XOR) of multiple operands
(ash a 3)	a << 3	Bit-wise left shift
(ash a -3)	a >> 3	Bit-wise right shift

Negative numbers are treated as two's-complements. If you have forgotten this, please refer to the Wiki page.

For example:

```
* (logior 1 2 4 8)
15
;; Explanation:
```

```
;;   0001
;;   0010
;;   0100
;; | 1000
;; -------
;;   1111

* (logand 2 -3 4)
0

;; Explanation:
;;   0010 (2)
;;   1101 (two's complement of -3)
;; & 0100 (4)
;; -------
;;   0000

* (logxor 1 3 7 15)
10

;; Explanation:
;;   0001
;;   0011
;;   0111
;; ^ 1111
;; -------
;;   1010

* (lognot -1)
0
;; Explanation:
;;   11 -> 00

* (lognot -3)
2
;;   101 -> 010

* (ash 3 2)
12
;; Explanation:
;;   11 -> 1100
```

```
* (ash -5 -2)
-2
;; Explanation
;;   11011 -> 110
```

Please see the CLHS page for a more detailed explanation or other bit-wise functions.

Loop, iteration, mapping

Introduction: loop, iterate, for, mapcar, series

loop is the built-in macro for iteration.

Its simplest form is `(loop (print "hello"))`: this will print forever.

A simple iteration over a list is:

```
(loop for x in '(1 2 3)
  do (print x))
```

It prints what's needed but returns `nil`.

If you want to return a list, use `collect`:

```
(loop for x in '(1 2 3)
  collect (* x 10))
;; (10 20 30)
```

The Loop macro is different than most Lisp expressions in having a complex internal domain-specific language that doesn't use s-expressions. So you need to read Loop expressions with half of your brain in Lisp mode, and the other half in Loop mode. You love it or you hate it.

Think of Loop expressions as having four parts: expressions that set up variables that will be iterated, expressions that conditionally terminate the iteration, expressions that do something on each iteration, and expressions that do something right before the Loop exits. In addition, Loop expressions can return a value. It is very rare to use all of these parts in a given Loop expression, but you can combine them in many ways.

iterate is a popular iteration macro that aims at being simpler, "lispier" and more predictable than `loop`, besides being extensible. However it isn't built-in, so you have to import it:

(ql:quickload “iterate”) (use-package :iterate)

Iterate looks like this:

```
(iter (for i from 1 to 5)
    (collect (* i i)))
```

(if you use loop and iterate in the same package, you might run into name conflicts)

Iterate also comes with `display-iterate-clauses` that can be quite handy:

```
(display-iterate-clauses '(for))
;; FOR PREVIOUS &OPTIONAL INITIALLY BACK     Previous value of a variable
;; FOR FIRST THEN                          Set var on first, and then on subsequent iterations
;; ...
```

Much of the examples on this page that are valid for loop are also valid for iterate, with minor modifications.

for is an extensible iteration macro that is often shorter than loop, that “unlike loop is extensible and sensible, and unlike iterate does not require code-walking and is easier to extend”.

It has the other advantage of having one construct that works for all data structures (lists, vectors, hash-tables…): in doubt, just use `for… over`:

```
(for:for ((x over <your data structure>))
   (print …))
```

You also have to quickload it:

(ql:quickload “for”)

We’ll also give examples with **`mapcar`** and `map`, and eventually with their friends `mapcon`, `mapcan`, `maplist`, `mapc` and `mapl` which E. Weitz categorizes very well in his “Common Lisp Recipes”, chap. 7. The one you are certainly accustomed to from other languages is `mapcar`: it takes a function, one or more lists as arguments, applies the function on each *element* of the lists one by one and returns a list of result.

```
(mapcar (lambda (it) (+ it 10)) '(1 2 3))
(11 12 13)
```

`map` is generic, it accepts list and vectors as arguments, and expects the type for its result as first argument:

```
(map 'vector (lambda (it) (+ it 10)) '(1 2 3))
;; #(11 12 13)
(map 'list (lambda (it) (+ it 10)) #(1 2 3))
;; (11 12 13)
(map 'string (lambda (it) (code-char it)) '#(97 98 99))
;; "abc"
```

The other constructs have their advantages in some situations ;) They either process the *tails* of lists, or *concatenate* the return values, or don't return anything. We'll see some of them.

If you like `mapcar`, use it a lot, and would like a quicker and shorter way to write lambdas, then you might like one of those lambda shorthand libraries.

Here is an example with cl-punch:

```
(mapcar ^(* _ 10) '(1 2 3))
;; (10 20 30)
```

and voilà :) We won't use this more in this recipe, but feel free to do.

Last but not least, you might like **series**, a library that describes itself as combining aspects of sequences, streams, and loops. Series expressions look like operations on sequences (= functional programming), but can achieve the same high level of efficiency as a loop. Series first appeared in "Common Lisp the Language", in the appendix A (it nearly became part of the language). Series looks like this:

```
(collect
  (mapping ((x (scan-range :from 1 :upto 5)))
```

```
    (* x x)))
;; (1 4 9 16 25)
```

`series` is good, but its function names are different from what we find in functional languages today. You might like the [“Generators The Way I Want Them Generated”]() library. It is a lazy sequences library, similar to `series` although younger and not as complete, with a “modern” API with words like `take`, `filter`, `for` or `fold`, and that is easy to use.

```
(range :from 20)
;; #<GTWIWTG::GENERATOR! {1001A90CA3}>

(take 4 (range :from 20))
;; (20 21 22 23)
```

At the time of writing, GTWIWTG is licensed under the GPLv3.

Recipes

Looping forever, return

```
(loop
    (print "hello"))
```

`return` can return a result:

```
(loop for i in '(1 2 3)
     when (> i 1)
     return i)
2
```

Looping a fixed number of times

dotimes

```
(dotimes (n 3)
  (print n))
;; =>
;; 0
```

```
;; 1
;; 2
;; NIL
```

Here `dotimes` returns `nil`. There are two ways to return a value. First, you can set a result form in the lambda list:

```
(dotimes (n 3 :done)
  ;;          ^^^^^ result form. It can be a s-expression.
  (print n))
;; =>
;; 0
;; 1
;; 2
;; :DONE
```

Or you can use `return` with return values:

```
(dotimes (i 3)
   (if (> i 1)
       (return :early-exit!)
       (print i)))
;; =>
;; 0
;; 1
;; :EARLY-EXIT!
```

loop… repeat

```
(loop repeat 10
  do (format t "Hello!~%"))
```

This prints 10 times “hello” and returns `nil`.

```
(loop repeat 10 collect (random 10))
;; (5 1 3 5 4 0 7 4 9 1)
```

with `collect`, this returns a list.

Series

```
(iterate ((n (scan-range :below 10)))
  (print n))
```

Looping an infinite number of times, cycling over a circular list

First, as shown above, we can simply use `(loop ...)` to loop infinitely. Here we show how to loop on a list forever.

We can build an infinite list by setting its last element to the list itself:

```
(loop with list-a = '(1 2 3)
      with infinite-list = (setf (cdr (last list-a)) list-a)
      for item in infinite-list
      repeat 8
      collect item)
;; (1 2 3 1 2 3 1 2)
```

Illustration: `(last '(1 2 3))` is `(3)`, a list, or rather a cons cell, whose `car` is 3 and `cdr` is NIL. See the [data-structures chapter]() for a reminder. This is the representation of `(list 3)`:

```
[o|/]
 |
 3
```

The representation of `(list 1 2 3)`:

```
[o|o]---[o|o]---[o|/]
 |       |       |
 1       2       3
```

By setting the `cdr` of the last element to the list itself, we make it recur on itself.

A notation shortcut is possible with the `#=` syntax:

```
(defparameter *list-a* '#1=(1 2 3 . #1#))
(setf *print-circle* t)  ;; don't print circular lists forever
*list-a*
```

If you need to alternate only between two values, use `for … then`:

```
(loop repeat 4
      for up = t then (not up)
      do (print up))
T
NIL
T
NIL
```

Iterate's for loop

For lists and vectors:

```
(iter (for item in '(1 2 3))
  (print item))
(iter (for i in-vector #(1 2 3))
  (print i))
```

or, a generalized iteration clause for lists and vectors, use `in-sequence` (you'll pay a speed penalty).

Looping over a hash-table is also straightforward:

```
(let ((h (let ((h (make-hash-table)))
           (setf (gethash 'a h) 1)
           (setf (gethash 'b h) 2)
           h)))
  (iter (for (k v) in-hashtable h)
    (print k)))
;; b
;; a
```

In fact, take a look here, or `(display-iterate-clauses '(for))` to know about iterating over

- symbols in-package
- forms - or lines, or whatever-you-wish - in-file, or in-stream
- elements in-sequence - sequences can be vectors or lists

Looping over a list

dolist

```
(dolist (item '(1 2 3))
  (print item))
```

`dolist` returns `nil`.

loop

with `in`, no surprises:

```
(loop for x in '(a b c)
      do (print x))
;; A
;; B
;; C
;; NIL
```

```
(loop for x in '(a b c)
      collect x)
;; (A B C)
```

With `on`, we loop over the cdr of the list:

```
(loop for i on '(1 2 3) do (print i))
;; (1 2 3)
;; (2 3)
;; (3)
```

mapcar

```
(mapcar (lambda (x)
             (print (* x 10)))
         '(1 2 3))
10
20
30
(10 20 30)
```

`mapcar` returns the results of the lambda function as a list.

Series

```
(iterate ((item (scan '(1 2 3))))
  (print item))
```

`scan-sublists` is the equivalent of `loop for ... on`:

```
(iterate ((i (scan-sublists '(1 2 3))))
  (print i))
```

Looping over a vector

loop: `across`

```
(loop for i across #(1 2 3) do (print i))
```

Series

```
(iterate ((i (scan #(1 2 3))))
  (print i))
```

Looping over a hash-table

We create a hash-table:

```
(defparameter h (make-hash-table))
(setf (gethash 'a h) 1)
(setf (gethash 'b h) 2)
```

Looping over keys and values

Looping over keys:

```
(loop for k being the hash-key of h do (print k))
;; b
;; a
```

Looping over values uses the same concept but with the `hash-value` keyword instead of `hash-key`:

```
(loop for k being the hash-value of h do (print k))
;; 1
;; 2
```

Looping over key-values pairs:

```
(loop for k
    being the hash-key
    using (hash-value v) of h
    do (format t "~a ~a~%" k v))
b 2
a 1
```

iterate

Use `in-hashtable`:

```
(iter (for (key value) in-hashtable h)
  (collect (list key value)))
```

for

the same with `for`:

```
(for:for ((it over h))
    (print it))
(A 1)
(B 2)
NIL
```

maphash

The lambda function of `maphash` takes two arguments: the key and the value:

```lisp
(maphash (lambda (key val)
             (format t "key: ~a val:~a~&" key val))
          h)
;; key: A val:1
;; key: B val:2
;; NIL
```

See also with-hash-table-iterator.

dohash

Only because we like this topic, we introduce another library, trivial-do. It has the `dohash` macro, that ressembles `dolist`:

```lisp
(dohash (key value h)
  (format t "key: ~A, value: ~A~%" key value))
```

Series

```lisp
(iterate (((k v) (scan-hash h)))
  (format t "~&~a ~a~%" k v))
```

Looping over two lists in parallel

loop

```lisp
(loop for x in '(a b c)
      for y in '(1 2 3)
      collect (list x y))
;; ((A 1) (B 2) (C 3))
```

To return a flat list, use `nconcing` instead of `collect`:

```lisp
(loop for x in '(a b c)
      for y in '(1 2 3)
      nconcing (list x y))
(A 1 B 2 C 3)
```

If a list is smaller than the other one, loop stops at the end of the small one:

```lisp
(loop for x in '(a b c)
      for y in '(1 2 3 4 5)
      collect (list x y))
;; ((A 1) (B 2) (C 3))
```

We could loop over the biggest list and manually access the elements of the smaller one by index, but it would quickly be inefficient. Instead, we can tell `loop` to extend the short list.

```lisp
(loop for y in '(1 2 3 4 5)
      for x-list = '(a b c) then (cdr x-list)
      for x = (or (car x-list) 'z)
      collect (list x y))
;; ((A 1) (B 2) (C 3) (Z 4) (Z 5))
```

The trick is that the notation `for … = … then (cdr …)` (note the = and the role of `then`) shortens our intermediate list at each iteration (thanks to `cdr`). It will first be `'(a b c)`, the initial value, then we will get the `cdr`: `(2 3)`, then `(3)`, then `NIL`. And both `(car NIL)` and `(cdr NIL)` return `NIL`, so we are good.

mapcar

```lisp
(mapcar (lambda (x y)
          (list x y))
        '(a b c)
        '(1 2 3))
;; ((A 1) (B 2) (C 3))
```

or simply:

```lisp
(mapcar #'list
        '(a b c)
        '(1 2 3))
;; ((A 1) (B 2) (C 3))
```

Return a flat list:

```lisp
(mapcan (lambda (x y)
          (list x y))
```

```
        '(a b c)
        '(1 2 3))
;; (A 1 B 2 C 3)
```

Series

```
(collect
  (#Mlist (scan '(a b c))
          (scan '(1 2 3))))
```

A more efficient way, when the lists are known to be of equal length:

```
(collect
  (mapping (((x y) (scan-multiple 'list
                                  '(a b c)
                                  '(1 2 3))))
    (list x y)))
```

Return a flat list:

```
(collect-append ; or collect-nconc
 (mapping (((x y) (scan-multiple 'list
                                 '(a b c)
                                 '(1 2 3))))
   (list x y)))
```

Nested loops

loop

```
(loop for x from 1 to 3
      collect (loop for y from 1 to x
            collect y))
;; ((1) (1 2) (1 2 3))
```

To return a flat list, use `nconcing` instead of the first `collect`.

iterate

```
(iter outer
   (for i below 2)
   (iter (for j below 3)
      (in outer (collect (list i j)))))
;; ((0 0) (0 1) (0 2) (1 0) (1 1) (1 2))
```

Series

```
(collect
  (mapping ((x (scan-range :from 1 :upto 3)))
    (collect (scan-range :from 1 :upto x))))
```

Computing an intermediate value

Use =.

With `for`:

```
(loop for x from 1 to 3
      for y = (* x 10)
      collect y)
;; (10 20 30)
```

With `with`, the difference being that the value is computed only once:

```
(loop for x from 1 to 3
      for y = (* x 10)
      with z = x
      collect (list x y z))
;; ((1 10 1) (2 20 1) (3 30 1))
```

The HyperSpec defines the `with` clause like this:

with-clause::= with var1 [type-spec] [= form1] {and var2 [type-spec] [= form2]}*

so it turns out we can specify the type before the `=` and chain the `with` with `and`:

```
(loop for x from 1 to 3
      for y integer = (* x 10)
      with z integer = x
      collect (list x y z))
```

```
(loop for x upto 3
      with foo = :foo
      and bar = :bar
      collect (list x foo bar))
```

We can also give `for` a `then` clause that will be called at each iteration:

```
(loop repeat 3
      for intermediate = 10 then (incf intermediate)
      do (print intermediate))
10
11
12
```

Here's a trick to alternate a boolean:

```
(loop repeat 4
      for up = t then (not up)
      do (print up))

T
NIL
T
NIL
```

Loop with a counter

loop

Iterate through a list, and have a counter iterate in parallel. The length of the list determines when the iteration ends. Two sets of actions are defined, one of which is executed conditionally.

```
* (loop for x in '(a b c d e)
      for y from 1
```

```
      when (> y 1)
      do (format t ", ")

      do (format t "~A" x)
      )
```

```
A, B, C, D, E
NIL
```

We could also write the preceding loop using the IF construct.

```
* (loop for x in '(a b c d e)
      for y from 1

      if (> y 1)
      do (format t ", ~A" x)
      else do (format t "~A" x)
      )
```

```
A, B, C, D, E
NIL
```

Series

By iterating on multiple series in parallel, and using an infinite range, we can make a counter.

```
(iterate ((x (scan '(a b c d e)))
          (y (scan-range :from 1)))
  (when (> y 1) (format t ", "))
  (format t "~A" x))
```

Ascending, descending order, limits

loop

`from… to…`:

```
(loop for i from 0 to 10
      do (print i))
;; 0 1 2 3 4 5 6 7 8 9 10
```

`from… below…`: this stops at 9:

```
(loop for i from 0 below 10
      do (print i))
```

Similarly, use `from 10 downto 0` (10…0) and `from 10 above 0` (10…1).

Series

`:from ... :upto`, including the upper limit:

```
(iterate ((i (scan-range :from 0 :upto 10)))
  (print i))
```

`:from ... :below`, excluding the upper limit:

```
(iterate ((i (scan-range :from 0 :below 10)))
  (print i))
```

Steps

loop

with `by`:

```
(loop for i from 1 to 10 by 2
      do (print i))
```

if you use `by (1+ (random 3))`, the random is evaluated only once, as if it was in a closure:

```
(let ((step (random 3)))
   (loop for i from 1 to 10 by (+ 1 step)
      do (print i)))
```

The step must always be a positive number. If you want to count down, see above.

Series

with `:by`:

```
(iterate ((i (scan-range :from 1 :upto 10 :by 2)))
  (print i))
```

Loop and conditionals

loop

with `if`, `else` and `finally`:

```
(loop repeat 10
      for x = (random 100)
      if (evenp x)
        collect x into evens
      else
        collect x into odds
      finally (return (values evens odds)))
```

```
(42 82 24 92 92)
(55 89 59 13 49)
```

Combining multiple clauses in an if body requires special syntax (`and do`, `and count`):

```
 (loop repeat 10
       for x = (random 100)
       if (evenp x)
          collect x into evens
          and do (format t "~a is even!~%" x)
       else
          collect x into odds
          and count t into n-odds
       finally (return (values evens odds n-odds)))
```

```
46 is even!
8 is even!
76 is even!
58 is even!
0 is even!
(46 8 76 58 0)
(7 45 43 15 69)
5
```

iterate

Translating (or even writing!) the above example using iterate is straight-forward:

```
(iter (repeat 10)
   (for x = (random 100))
   (if (evenp x)
       (progn
         (collect x into evens)
         (format t "~a is even!~%" x))
       (progn
         (collect x into odds)
         (count t into n-odds)))
   (finally (return (values evens odds n-odds))))
```

Series

The preceding loop would be done a bit differently in Series. `split` sorts one series into multiple according to provided boolean series.

```
(let* ((number (#M(lambda (n) (random 100))
                  (scan-range :below 10)))
       (parity (#Mevenp number)))
  (iterate ((n number) (p parity))
    (when p (format t "~a is even!~%" n)))
  (multiple-value-bind (evens odds) (split number parity)
    (values (collect evens)
            (collect odds)
            (collect-length odds))))
```

Note that although `iterate` and the three `collect` expressions are written sequentially, only one iteration is performed, the same as the example with loop.

Begin the loop with a clause (initially)

```
(loop initially
      (format t "~a " 'loop-begin)
      for x below 3
      do (format t "~a " x))
;; LOOP-BEGIN 0 1 2
```

`initially` also exists with `iterate`.

Terminate the loop with a test (until, while)

loop

```
(loop for x in '(1 2 3 4 5)
    until (> x 3)
    collect x)
;; (1 2 3)
```

the same, with `while`:

```
(loop for x in '(1 2 3 4 5)
    while (< x 4)
    collect x)
```

Series

We truncate the series with `until-if`, then collect from its result.

```
(collect
  (until-if (lambda (i) (> i 3))
            (scan '(1 2 3 4 5))))
```

Loop, print and return a result

loop

`do` and `collect` can be combined in one expression

```
(loop for x in '(1 2 3 4 5)
    while (< x 4)
        do (format t "x is ~a~&" x)
    collect x)
x is 1
x is 2
x is 3
(1 2 3)
```

Series

By mapping, we can perform a side effect and also collect items

```
(collect
  (mapping ((x (until-if (complement (lambda (x) (< x 4)))
                         (scan '(1 2 3 4 5)))))
    (format t "x is ~a~&" x)
    x))
```

Named loops and early exit

loop

The special `loop named` foo syntax allows you to create a loop that you can exit early from. The exit is performed using `return-from`, and can be used from within nested loops.

```
;; useless example
(loop named loop-1
    for x from 0 to 10 by 2
    do (loop for y from 0 to 100 by (1+ (random 3))
            when (< x y)
            do (return-from loop-1 (values x y))))
0
2
```

Sometimes, you want to return early but execute the `finally` clause anyways. Use `loop-finish`.

```
(loop for x from 0 to 100
  do (print x)
  when (>= x 3)
  return x
  finally (print :done))  ;; <-- not printed
;; 0
;; 1
;; 2
;; 3
;; 3

(loop for x from 0 to 100
  do (print x)
  when (>= x 3)
  do (loop-finish)
  finally (print :done)
     (return x))
;; 0
;; 1
;; 2
;; 3
;; :DONE
;; 3
```

It is most needed when some computation must take place in the `finally` clause.

Loop shorthands for when/return

Several actions provide shorthands for combinations of when/return:

```
* (loop for x in '(foo 2)
      thereis (numberp x))
T

* (loop for x in '(foo 2)
      never (numberp x))
NIL
```

```
* (loop for x in '(foo 2)
      always (numberp x))
NIL
```

They correspond to the functions `some`, `notany` and `every`:

```
(some #'numberp '(foo 2))
(notany #'numberp '(foo 2))
(every #'numberp '(foo 2))
```

Series

A block is manually created and returned from.

```
(block loop-1
  (iterate ((x (scan-range :from 0 :upto 10 :by 2)))
    (iterate ((y (scan-range :from 0 :upto 100 :by (1+ (random 3
      (when (< x y)
        (return-from loop-1 (values x y)))))))
```

Count

loop

```
(loop for i from 1 to 3 count (oddp i))
;; 2
```

Series

```
(collect-length (choose-if #'oddp (scan-range :from 1 :upto 3)))
```

Summation

loop

```
(loop for i from 1 to 3 sum (* i i))
;; 14
```

Summing into a variable:

```
(loop for i from 1 to 3
   sum (* i i) into total
   do (print i)
   finally (print total))
1
2
3
14
```

Series

```
(collect-sum (#M(lambda (i) (* i i))
                (scan-range :from 1 :upto 3)))
```

max, min

loop

```
(loop for i from 1 to 3 maximize (mod i 3))
;; 2
```

and `minimize`.

Series

```
(collect-max (#M(lambda (i) (mod i 3))
                (scan-range :from 1 :upto 3)))
```

and `collect-min`.

Destructuring, aka pattern matching against the list or dotted pairs

loop

```
(loop for (a b) in '((x 1) (y 2) (z 3))
      collect (list b a) )
;; ((1 X) (2 Y) (3 Z))
```

```
(loop for (x . y) in '((1 . a) (2 . b) (3 . c)) collect y)
;; (A B C)
```

Use `nil` to ignore a term:

```
(loop for (a nil) in '((x 1) (y 2) (z 3))
      collect a )
;; (X Y Z)
```

Iterating 2 by 2 over a list

To iterate over a list, 2 items at a time we use a combination of `on`, `by` and destructuring.

We use `on` to loop over the rest (the `cdr`) of the list.

```
(loop for rest on '(a 2 b 2 c 3)
      collect rest)
;; ((A 2 B 2 C 3) (2 B 2 C 3) (B 2 C 3) (2 C 3) (C 3) (3))
```

We use `by` to skip one element at every iteration (`(cddr list)` is equivalent to `(rest (rest list))`)

```
(loop for rest on '(a 2 b 2 c 3) by #'cddr
      collect rest)
;; ((A 2 B 2 C 3) (B 2 C 3) (C 3))
```

Then we add destructuring to bind only the first two items at each iteration:

```
(loop for (key value) on '(a 2 b 2 c 3) by #'cddr
      collect (list key (* 2 value)))
;; ((A 2) (B 4) (C 6))
```

Series

In general, with `destructuring-bind`:

```
(collect
  (mapping ((l (scan '((x 1) (y 2) (z 3)))))
    (destructuring-bind (a b) l
      (list b a))))
```

But for alists, `scan-alist` is provided:

```
(collect
  (mapping (((a b) (scan-alist '((1 . a) (2 . b) (3 . c)))))
    b))
```

Iterate unique features lacking in loop

`iterate` has some other things unique to it.

If you are a newcomer in Lisp, it's perfectly OK to keep this section for later. You could very well spend your career in Lisp without resorting to those features… although they might turn out useful one day.

No rigid order for clauses

`loop` requires that all `for` clauses appear before the loop body, for example before a `while`. It's ok for `iter` to not follow this order:

```
(iter (for x in '(1 2 99)
  (while (< x 10))
  (for y = (print x))
  (collect (list x y)))
```

Accumulating clauses can be nested

`collect`, `appending` and other accumulating clauses can appear anywhere:

```
(iter (for x in '(1 2 3))
  (case x
    (1 (collect :a))
    ;;  ^^ iter keyword, nested in a s-expression.
    (2 (collect :b))))
```

Finders: `finding`

`iterate` has finders.

> A finder is a clause whose value is an expression that meets some condition.

We can use `finding` followed by `maximizing`, `minimizing` or `such-that`.

Here's how to find the longest list in a list of lists:

```
(iter (for elt in '((a) (b c d) (e f)))
      (finding elt maximizing (length elt)))
=> (B C D)
```

The rough equivalent in LOOP would be:

```
(loop with max-elt = nil
      with max-key = 0
      for elt in '((a) (b c d) (e f))
      for key = (length elt)
      do
      (when (> key max-key)
        (setf max-elt elt
              max-key key))
      finally (return max-elt))
=> (B C D)
```

There could be more than one `such-that` clause:

```
 (iter (for i in '(7 -4 2 -3))
       (if (plusp i)
    (finding i such-that (evenp i))
        (finding (- i) such-that (oddp i))))
;; => 2
```

We can also write `such-that #'evenp` and `such-that #'oddp`.

Control flow: `next-iteration`

It is like “continue” and loop doesn’t have it.

> Skips the remainder of the loop body and begins the next iteration of the loop.

`iterate` also has `first-iteration-p` and `(if-first-time then else)`.

See control flow.

Generators

Use `generate` and `next`. A generator is lazy, it goes to the next value when said explicitly.

```
(iter (for i in '(1 2 3 4 5))
      (generate c in-string "black")
      (if (oddp i) (next c))
      (format t "~a " c))
;; b b l l a
;; NIL
```

Variable backtracking (`previous`) VS parallel binding

`iterate` allows us to get the previous value of a variable:

```
(iter (for el in '(a b c d e))
      (for prev-el previous el)
      (collect (list el prev-el)))
;; => ((A NIL) (B A) (C B) (D C) (E D))
```

In this case however we can do it with `loop`’s parallel binding `and`, which is unsupported in `iterate`:

```
(loop for el in '(a b c d e)
      and prev-el = nil then el
      collect (list el prev-el))
```

More clauses

- `in-string` can be used explicitly to iterate character by character over a string. With loop, use `across`.

```
(iter (for c in-string "hello")
      (collect c))
;; => (#\h #\e #\l #\l #\o)
```

- `loop` offers `collecting`, `nconcing`, and `appending`. `iterate` has these and also `adjoining`, `unioning`, `nunioning`, and `accumulating`.

```
(iter (for el in '(a b c a d b))
      (adjoining el))
;; => (A B C D)
```

(`adjoin` is a set operation)

- `loop` has `summing`, `counting`, `maximizing`, and `minimizing`. `iterate` also includes `multiplying` and `reducing`. reducing is the generalized reduction builder:

```
(iter (with dividend = 100)
      (for divisor in '(10 5 2))
      (reducing divisor by #'/ initial-value dividend))
;; => 1
```

Iterate is extensible

```
(defmacro dividing-by (num &keys (initial-value 0))
  `(reducing ,num by #'/ initial-value ,initial-value))

(iter (for i in '(10 5 2))
      (dividing-by i :initial-value 100))
=> 1
```

but there is more to it, see the documentation.

We saw libraries extending `loop`, for example CLSQL, but they are full of feature flag checks (`#+(or allegro clisp-aloop cmu openmcl sbcl scl)`) and they call internal modules (`ansi-loop::add-loop-path`, `sb-loop::add-loop-path` etc).

Custom series scanners

If we often scan the same type of object, we can write our own scanner for it: the iteration itself can be factored out. Taking the example above, of scanning a list of two-element lists, we'll write a scanner that returns a series of the first elements and a series of the second.

```
(defun scan-listlist (listlist)
  (declare (optimizable-series-function 2))
  (map-fn '(values t t)
          (lambda (l)
            (destructuring-bind (a b) l
              (values a b)))
          (scan listlist)))

(collect
  (mapping (((a b) (scan-listlist '((x 1) (y 2) (z 3)))))
    (list b a)))
```

Shorter series expressions

Consider this series expression:

```
(collect-sum (mapping ((i (scan-range :length 5)))
                    (* i 2)))
```

It's a bit longer than it needs to be, the `mapping` form's only purpose is to bind the variable `i`, and `i` is used in only one place. Series has a "hidden feature" that allows us to simplify this expression to the following:

```
(collect-sum (* 2 (scan-range :length 5)))
```

This is called implicit mapping and can be enabled in the call to `series::install`:

```
(series::install :implicit-map t)
```

When using implicit mapping, the `#M` reader macro demonstrated above becomes redundant.

Loop gotchas

- the keyword `it`, often used in functional constructs, can be recognized as a loop keyword. Don't use it inside a loop.

Iterate gotchas

It breaks on the function `count`:

```
(iter (for i from 1 to 10)
      (sum (count i '(1 3 5))))
```

It doesn't recognize the built-in `count` function and instead signals a condition.

It works in loop:

```
(loop for i from 1 to 10
    sum (count i '(1 3 5 99)))
;; 3
```

Appendix: list of loop keywords

Name Clause

```
named
```

Variable Clauses

```
initially finally for as with
```

Main Clauses

```
do collect collecting append
appending nconc nconcing into count
counting sum summing maximize return loop-finish
maximizing minimize minimizing doing
thereis always never if when
unless repeat while until
```

These don't introduce clauses:

```
= and it else end from upfrom
above below to upto downto downfrom
in on then across being each the hash-key
hash-keys of using hash-value hash-values
symbol symbols present-symbol
present-symbols external-symbol
external-symbols fixnum float t nil of-type
```

But note that it's the parsing that determines what is a keyword. For example in:

```
(loop for key in hash-values)
```

Only `for` and `in` are keywords.

©Dan Robertson on Stack Overflow.

Credit and references

Loop

- Tutorial for the Common Lisp Loop Macro by Peter D. Karp
- Common Lisp's Loop Macro Examples for Beginners by Yusuke Shinyama
- Section 6.1 The LOOP Facility, of the draft Common Lisp Standard (X3J13/94-101R) - the (draft) standard provides background information on Loop development, specification and examples. Single PDF file available
- 26. Loop by Jon L White, edited and expanded by Guy L. Steele Jr. - from the book "Common Lisp the Language, 2nd Edition". Strong connection to the draft above, with supplementing comments and examples.

Iterate

- The Iterate Manual -by Jonathan Amsterdam and Luís Oliveira
- iterate - Pseudocodic Iteration - by Shubhamkar Ayare

- Loop v Iterate - SabraOnTheHill
- Comparing loop and iterate - by Stephen Bach (web archive)

Series

- Common Lisp the Language (2nd Edition) - Appendix A. Series
- SERIES for Common Lisp - Richard C. Waters

Others

- See also: more functional constructs (do-repeat, take,…)

Multidimensional arrays

Common Lisp has native support for multidimensional arrays, with some special treatment for 1-D arrays, called `vectors`. Arrays can be *generalised* and contain any type (`element-type t`), or they can be *specialised* to contain specific types such as `single-float` or `integer`. A good place to start is Practical Common Lisp Chapter 11, Collections by Peter Seibel.

A quick reference to some common operations on arrays is given in the section on Arrays and vectors.

Some libraries available on Quicklisp for manipulating arrays:

- array-operations maintained by @Symbolics defines functions `generate`, `permute`, `displace`, `flatten`, `split`, `combine`, `reshape`. It also defines `each`, for element-wise operations. This is a fork of bendudson/array-operations which is a fork of tpapp/array-operations, the original author.
- cmu-infix includes array indexing syntax for multidimensional arrays.
- lla is a library for linear algebra, calling BLAS and LAPACK libraries. It differs from most CL linear algebra packages in using intuitive function names, and can operate on native arrays as well as CLOS objects.

This page covers what can be done with the built-in multidimensional arrays, but there are limitations. In particular:

- Interoperability with foreign language arrays, for example when calling libraries such as BLAS, LAPACK or GSL.
- Extending arithmetic and other mathematical operators to handle arrays, for example so that `(+ a b)` works when `a` and/or `b` are arrays.

Both of these problems can be solved by using CLOS to define an extended array class, with native arrays as a special case. Some libraries available through quicklisp which take this approach are:

- matlisp, some of which is described in sections below.
- MGL-MAT, which has a manual and provides bindings to BLAS and CUDA. This is used in a machine learning library MGL.
- cl-ana, a data analysis package with a manual, which includes operations on arrays.
- Antik, used in GSLL, a binding to the GNU Scientific Library.

A relatively new but actively developed package is MAGICL, which provides wrappers around BLAS and LAPACK libraries. At the time of writing this package is not on Quicklisp, and only works under SBCL and CCL. It seems to be particularly focused on complex arrays, but not exclusively. To install, clone the repository in your quicklisp `local-projects` directory e.g. under Linux/Unix:

```
$ cd ~/quicklisp/local-projects
$ git clone https://github.com/rigetticomputing/magicl.git
```

Instructions for installing dependencies (BLAS, LAPACK and Expokit) are given on the github web pages. Low-level routines wrap foreign functions, so have the Fortran names e.g `magicl.lapack-cffi::%zgetrf`. Higher-level interfaces to some of these functions also exist, see the source directory and documentation.

Taking this further, domain specific languages have been built on Common Lisp, which can be used for numerical calculations with arrays. At the time of writing the most widely used and supported of these are:

- Maxima
- Axiom

CLASP is a project which aims to ease interoperability of Common Lisp with other languages (particularly C++), by using LLVM. One of the main applications of this project is to numerical/scientific computing.

Creating

The function CLHS: make-array can create arrays filled with a single value

```
* (defparameter *my-array* (make-array '(3 3) :initial-element 1
```

```
* (defparameter *my-array* (make-array '(3 2) :initial-element 1
*MY-ARRAY*
* *my-array*
#2A((1.0 1.0) (1.0 1.0) (1.0 1.0))
```

More complicated array values can be generated by first making an array, and then iterating over the elements to fill in the values (see section below on element access).

The [array-operations](#) library provides `generate`, a convenient function for creating arrays which wraps this iteration.

```
* (ql:quickload :array-operations)
To load "array-operations":
  Load 1 ASDF system:
    array-operations
; Loading "array-operations"

(:ARRAY-OPERATIONS)

* (aops:generate #'identity 7 :position)
#(0 1 2 3 4 5 6)
```

Note that the nickname for `array-operations` is `aops`. The `generate` function can also iterate over the array subscripts by passing the key `:subscripts`. See the [Array Operations manual on generate](#) for more examples.

Random numbers

To create an 3x3 array containing random numbers drawn from a uniform distribution, `generate` can be used to call the CL [random](#) function:

```
* (aops:generate (lambda () (random 1.0)) '(3 3))
#2A((0.99292254 0.929777 0.93538976)
    (0.31522608 0.45167792 0.9411855)
    (0.96221936 0.9143338 0.21972346))
```

An array of Gaussian (normal) random numbers with mean of zero and standard deviation of one, using the alexandria package:

```
* (ql:quickload :alexandria)
To load "alexandria":
  Load 1 ASDF system:
    alexandria
; Loading "alexandria"

(:ALEXANDRIA)

* (aops:generate #'alexandria:gaussian-random 4)
#(0.5522547885338768d0 -1.2564808468164517d0 0.9488161476129733c
  -0.10372852118266523d0)
```

Note that this is not particularly efficient: It requires a function call for each element, and although `gaussian-random` returns two random numbers, only one of them is used.

For more efficient implementations, and a wider range of probability distributions, there are packages available on Quicklisp. See CLiki for a list.

Accessing elements

To access the individual elements of an array there are the aref and row-major-aref functions.

The aref function takes the same number of index arguments as the array has dimensions. Indexing is from 0 and row-major as in C, but not Fortran.

```
* (defparameter *a* #(1 2 3 4))
*A*
* (aref *a* 0)
1
* (aref *a* 3)
4
* (defparameter *b* #2A((1 2 3) (4 5 6)))
*B*
* (aref *b* 1 0)
```

```
4
* (aref *b* 0 2)
3
```

The range of these indices can be found using array-dimensions:

```
* (array-dimensions *a*)
(4)
* (array-dimensions *b*)
(2 3)
```

or the rank of the array can be found, and then the size of each dimension queried:

```
* (array-rank *a*)
1
* (array-dimension *a* 0)
4
* (array-rank *b*)
2
* (array-dimension *b* 0)
2
* (array-dimension *b* 1)
3
```

To loop over an array nested loops can be used, such as:

```
* (defparameter a #2A((1 2 3) (4 5 6)))
A
* (destructuring-bind (n m) (array-dimensions a)
    (loop for i from 0 below n do
      (loop for j from 0 below m do
        (format t "a[~a ~a] = ~a~%" i j (aref a i j)))))

a[0 0] = 1
a[0 1] = 2
a[0 2] = 3
a[1 0] = 4
a[1 1] = 5
a[1 2] = 6
NIL
```

A utility macro which does this for multiple dimensions is `nested-loop`:

```
(defmacro nested-loop (syms dimensions &body body)
  "Iterates over a multidimensional range of indices.

   SYMS must be a list of symbols, with the first symbol
   corresponding to the outermost loop.

   DIMENSIONS will be evaluated, and must be a list of
   dimension sizes, of the same length as SYMS.

   Example:
    (nested-loop (i j) '(10 20) (format t '~a ~a~%' i j))

  "
  (unless syms (return-from nested-loop `(progn ,@body))) ; No s

  ;; Generate gensyms for dimension sizes
  (let* ((rank (length syms))
         ;; reverse our symbols list,
         ;; since we start from the innermost.
         (syms-rev (reverse syms))
         ;; innermost dimension first:
         (dims-rev (loop for i from 0 below rank
                         collecting (gensym)))
         ;; start with innermost expression
         (result `(progn ,@body)))
    ;; Wrap previous result inside a loop for each dimension
    (loop for sym in syms-rev for dim in dims-rev do
         (unless (symbolp sym)
           (error "~S is not a symbol. First argument to nested-
         (setf result
               `(loop for ,sym from 0 below ,dim do
                     ,result)))
    ;; Add checking of rank and dimension types,
    ;; and get dimensions into gensym list.
    (let ((dims (gensym)))
      `(let ((,dims ,dimensions))
         (unless (= (length ,dims) ,rank)
           (error "Incorrect number of dimensions: Expected ~a b
         (dolist (dim ,dims)
           (unless (integerp dim)
```

```
          (error "Dimensions must be integers: ~S" dim)))
      ;; dimensions reversed so that innermost is last:
      (destructuring-bind ,(reverse dims-rev) ,dims
        ,result)))))
```

so that the contents of a 2D array can be printed using:

```
* (defparameter a #2A((1 2 3) (4 5 6)))
A
* (nested-loop (i j) (array-dimensions a)
      (format t "a[~a ~a] = ~a~%" i j (aref a i j)))

a[0 0] = 1
a[0 1] = 2
a[0 2] = 3
a[1 0] = 4
a[1 1] = 5
a[1 2] = 6
NIL
```

[Note: This macro is available in this fork of array-operations, but not Quicklisp]

Row major indexing

In some cases, particularly element-wise operations, the number of dimensions does not matter. To write code which is independent of the number of dimensions, array element access can be done using a single flattened index via row-major-aref. The array size is given by array-total-size, with the flattened index starting at 0.

```
* (defparameter a #2A((1 2 3) (4 5 6)))
A
* (array-total-size a)
6
* (loop for i from 0 below (array-total-size a) do
     (setf (row-major-aref a i) (+ 2.0 (row-major-aref a i))))
NIL
* a
#2A((3.0 4.0 5.0) (6.0 7.0 8.0))
```

Infix syntax

The cmu-infix library provides some different syntax which can make mathematical expressions easier to read:

```
* (ql:quickload :cmu-infix)
To load "cmu-infix":
  Load 1 ASDF system:
    cmu-infix
; Loading "cmu-infix"

(:CMU-INFIX)

* (named-readtables:in-readtable cmu-infix:syntax)
(("COMMON-LISP-USER" . #<NAMED-READTABLE CMU-INFIX:SYNTAX {10030
 ...)

* (defparameter arr (make-array '(3 2) :initial-element 1.0))
ARR

* #i(arr[0 1] = 2.0)
2.0

* arr
#2A((1.0 2.0) (1.0 1.0) (1.0 1.0))
```

A matrix-matrix multiply operation can be implemented as:

```
(let ((A #2A((1 2) (3 4)))
      (B #2A((5 6) (7 8)))
      (result (make-array '(2 2) :initial-element 0.0)))

     (loop for i from 0 to 1 do
           (loop for j from 0 to 1 do
                 (loop for k from 0 to 1 do
                       #i(result[i j] += A[i k] * B[k j]))))
      result)
```

See the section below on linear algebra, for alternative matrix-multiply implementations.

Element-wise operations

To multiply two arrays of numbers of the same size, pass a function to `each` in the array-operations library:

```
* (aops:each #'* #(1 2 3) #(2 3 4))
#(2 6 12)
```

For improved efficiency there is the `aops:each*` function, which takes a type as first argument to specialise the result array.

To add a constant to all elements of an array:

```
* (defparameter *a* #(1 2 3 4))
*A*
* (aops:each (lambda (it) (+ 42 it)) *a*)
#(43 44 45 46)
* *a*
#(1 2 3 4)
```

Note that `each` is not destructive, but makes a new array. All arguments to `each` must be arrays of the same size, so `(aops:each #'+ 42 *a*)` is not valid.

Vectorising expressions

An alternative approach to the `each` function above, is to use a macro to iterate over all elements of an array:

```
(defmacro vectorize (variables &body body)
  ;; Check that variables is a list of only symbols
  (dolist (var variables)
    (if (not (symbolp var))
        (error "~S is not a symbol" var)))

    ;; Get the size of the first variable, and create a new arra
    ;; of the same type for the result
```

```
    ;; of the same type for the result
    `(let ((size (array-total-size ,(first variables))) ; Total
           (result (make-array (array-dimensions ,(first variabl
                               :element-type (array-element-type
       ;; Check that all variables have the same sizeo
       ,@(mapcar (lambda (var) `(if (not (equal (array-dimensior
                                               (array-dimensior
                                    (error "~S and ~S have diffe
              (rest variables))

       (dotimes (indx size)
         ;; Locally redefine variables to be scalars at a given
         (let ,(mapcar (lambda (var) (list var `(row-major-aref
           ;; User-supplied function body now evaluated for each
           (setf (row-major-aref result indx) (progn ,@body))))
       result))
```

[Note: Expanded versions of this macro are available in this fork of array-operations, but not Quicklisp]

This can be used as:

```
* (defparameter *a* #(1 2 3 4))
*A*
* (vectorize (*a*) (* 2 *a*))
#(2 4 6 8)
```

Inside the body of the expression (second form in `vectorize` expression) the symbol `*a*` is bound to a single element. This means that the built-in mathematical functions can be used:

```
* (defparameter a #(1 2 3 4))
A
* (defparameter b #(2 3 4 5))
B
* (vectorize (a b) (* a (sin b)))
#(0.9092974 0.28224 -2.2704074 -3.8356972)
```

and combined with `cmu-infix`:

```
* (vectorize (a b) #i(a * sin(b)) )
#(0.9092974 0.28224 -2.2704074 -3.8356972)
```

Calling BLAS

Several packages provide wrappers around BLAS, for fast matrix manipulation.

The lla package in quicklisp includes calls to some functions:

Scale an array

scaling by a constant factor:

```
* (defparameter a #(1 2 3))
* (lla:scal! 2.0 a)
* a
#(2.0d0 4.0d0 6.0d0)
```

AXPY

This calculates `a * x + y` where `a` is a constant, `x` and `y` are arrays. The `lla:axpy!` function is destructive, modifying the last argument (`y`).

```
* (defparameter x #(1 2 3))
A
* (defparameter y #(2 3 4))
B
* (lla:axpy! 0.5 x y)
#(2.5d0 4.0d0 5.5d0)
* x
#(1.0d0 2.0d0 3.0d0)
* y
#(2.5d0 4.0d0 5.5d0)
```

If the `y` array is complex, then this operation calls the complex number versions of these operators:

```
* (defparameter x #(1 2 3))
* (defparameter y (make-array 3 :element-type '(complex double-f
```

```
                                    :initial-element #C(1d0 1d0)))
* y
#(#C(1.0d0 1.0d0) #C(1.0d0 1.0d0) #C(1.0d0 1.0d0))

* (lla:axpy! #C(0.5 0.5) a b)
#(#C(1.5d0 1.5d0) #C(2.0d0 2.0d0) #C(2.5d0 2.5d0))
```

Dot product

The dot product of two vectors:

```
* (defparameter x #(1 2 3))
* (defparameter y #(2 3 4))
* (lla:dot x y)
20.0d0
```

Reductions

The reduce function operates on sequences, including vectors (1D arrays), but not on multidimensional arrays. To get around this, multidimensional arrays can be displaced to create a 1D vector. Displaced arrays share storage with the original array, so this is a fast operation which does not require copying data:

```
* (defparameter a #2A((1 2) (3 4)))
A
* (reduce #'max (make-array (array-total-size a) :displaced-to a
4
```

The `array-operations` package contains `flatten`, which returns a displaced array i.e doesn't copy data:

```
* (reduce #'max (aops:flatten a))
```

An SBCL extension, array-storage-vector provides an efficient but not portable way to achieve the same thing:

```
* (reduce #'max (array-storage-vector a))
4
```

More complex reductions are sometimes needed, for example finding the maximum absolute difference between two arrays. Using the above methods we could do:

```
* (defparameter a #2A((1 2) (3 4)))
A
* (defparameter b #2A((1 3) (5 4)))
B
* (reduce #'max (aops:flatten
                  (aops:each
                    (lambda (a b) (abs (- a b))) a b)))
2
```

This involves allocating an array to hold the intermediate result, which for large arrays could be inefficient. Similarly to `vectorize` defined above, a macro which does not allocate can be defined as:

```
(defmacro vectorize-reduce (fn variables &body body)
  "Performs a reduction using FN over all elements in a vectoriz
   on array VARIABLES.

   VARIABLES must be a list of symbols bound to arrays.
   Each array must have the same dimensions. These are
   checked at compile and run-time respectively.
  "
  ;; Check that variables is a list of only symbols
  (dolist (var variables)
    (if (not (symbolp var))
        (error "~S is not a symbol" var)))

  (let ((size (gensym)) ; Total array size (same for all variabl
        (result (gensym)) ; Returned value
        (indx (gensym)))  ; Index inside loop from 0 to size

    ;; Get the size of the first variable
    `(let ((,size (array-total-size ,(first variables))))
       ;; Check that all variables have the same size
       ,@(mapcar (lambda (var) `(if (not (equal (array-dimensior
                                                 (array-dimensior
```

```
                                          (error "~S and ~S have diffe
      (rest variables))

    ;; Apply FN with the first two elements (or fewer if size
    (let ((,result (apply ,fn (loop for ,indx below (min ,siz
                                            (let ,(map 'list (lambda (
                                              (progn ,@body))))))

      ;; Loop over the remaining indices
      (loop for ,indx from 2 below ,size do
         ;; Locally redefine variables to be scalars at a giv
           (let ,(mapcar (lambda (var) (list var `(row-major-
             ;; User-supplied function body now evaluated for
             (setf ,result (funcall ,fn ,result (progn ,@body
      ,result))))
```

[Note: This macro is available in this fork of array-operations, but not Quicklisp]

Using this macro, the maximum value in an array A (of any shape) is:

```
* (vectorize-reduce #'max (a) a)
```

The maximum absolute difference between two arrays A and B, of any shape as long as they have the same shape, is:

```
* (vectorize-reduce #'max (a b) (abs (- a b)))
```

Linear algebra

Several packages provide bindings to BLAS and LAPACK libraries, including:

- lla
- MAGICL

A longer list of available packages is on CLiki’s linear algebra page.

In the examples below the lla package is loaded:

```
* (ql:quickload :lla)

To load "lla":
  Load 1 ASDF system:
    lla
; Loading "lla"
.
(:LLA)
```

Matrix multiplication

The lla function `mm` performs vector-vector, matrix-vector and matrix-matrix multiplication.

Vector dot product

Note that one vector is treated as a row vector, and the other as column:

```
* (lla:mm #(1 2 3) #(2 3 4))
20
```

Matrix-vector product

```
* (lla:mm #2A((1 1 1) (2 2 2) (3 3 3))  #(2 3 4))
#(9.0d0 18.0d0 27.0d0)
```

which has performed the sum over `j` of `A[i j] * x[j]`

Matrix-matrix multiply

```
* (lla:mm #2A((1 2 3) (1 2 3) (1 2 3))  #2A((2 3 4) (2 3 4) (2 3
#2A((12.0d0 18.0d0 24.0d0) (12.0d0 18.0d0 24.0d0) (12.0d0 18.0d0
```

which summed over `j` in `A[i j] * B[j k]`

Note that the type of the returned arrays are simple arrays, specialised to element type `double-float`

```
* (type-of (lla:mm #2A((1 0 0) (0 1 0) (0 0 1)) #(1 2 3)))
(SIMPLE-ARRAY DOUBLE-FLOAT (3))
```

Outer product

The array-operations package contains a generalised outer product function:

```
* (ql:quickload :array-operations)
To load "array-operations":
  Load 1 ASDF system:
    array-operations
; Loading "array-operations"

(:ARRAY-OPERATIONS)
* (aops:outer #'* #(1 2 3) #(2 3 4))
#2A((2 3 4) (4 6 8) (6 9 12))
```

which has created a new 2D array `A[i j] = B[i] * C[j]`. This `outer` function can take an arbitrary number of inputs, and inputs with multiple dimensions.

Matrix inverse

The direct inverse of a dense matrix can be calculated with `invert`

```
* (lla:invert #2A((1 0 0) (0 1 0) (0 0 1)))
#2A((1.0d0 0.0d0 -0.0d0) (0.0d0 1.0d0 -0.0d0) (0.0d0 0.0d0 1.0d0
```

e.g

```
* (defparameter a #2A((1 2 3) (0 2 1) (1 3 2)))
A
* (defparameter b (lla:invert a))
B
* (lla:mm a b)
#2A((1.0d0 2.220446049250313d-16 0.0d0)
    (0.0d0 1.0d0 0.0d0)
    (0.0d0 1.1102230246251565d-16 0.9999999999999998d0))
```

Calculating the direct inverse is generally not advisable, particularly for large matrices. Instead the LU decomposition can be calculated and used for multiple inversions.

```
* (defparameter a #2A((1 2 3) (0 2 1) (1 3 2)))
A
* (defparameter b (lla:mm a #(1 2 3)))
B
* (lla:solve (lla:lu a) b)
#(1.0d0 2.0d0 3.0d0)
```

Singular value decomposition

The `svd` function calculates the singular value decomposition of a given matrix, returning an object with slots for the three returned matrices:

```
* (defparameter a #2A((1 2 3) (0 2 1) (1 3 2)))
A
* (defparameter a-svd (lla:svd a))
A-SVD
* a-svd
#S(LLA:SVD
   :U #2A((-0.6494608633564334d0 0.7205486773948702d0 0.24292013
          (-0.3744175632000917d0 -0.5810891192666799d0 0.7225973
          (-0.6618248071322363d0 -0.3783451320875919d0 -0.647180
   :D #S(CL-NUM-UTILS.MATRIX:DIAGONAL-MATRIX
         :ELEMENTS #(5.593122609997059d0 1.2364443401235103d0
                     0.4338027931171437 6d0))
   :VT #2A((-0.2344460799312531d0 -0.7211054639318696d0 -0.65195
           (0.2767642134809678d0 -0.6924017945853318d0 0.6663192
           (-0.9318994611765425d0 -0.0242211631144076 4d0 0.36190
```

The diagonal matrix (singular values) and vectors can be accessed with functions:

```
(lla:svd-u a-svd)
#2A((-0.6494608633564334d0 0.7205486773948702d0 0.24292013188045
    (-0.3744175632000917d0 -0.5810891192666799d0 0.72259734557785
    (-0.6618248071322363d0 -0.3783451320875919d0 -0.64718072104З


* (lla:svd-d a-svd)
```

```
* (lla:svd-d a-svd)
#S(CL-NUM-UTILS.MATRIX:DIAGONAL-MATRIX
   :ELEMENTS #(5.593122609997059d0 1.2364443401235103d0 0.433802

* (lla:svd-vt a-svd)
#2A((-0.23444607993125 31d0 -0.7211054639318696d0 -0.651952410450
    (0.2767642134809678d0 -0.6924017945853318d0 0.666319236546O2
    (-0.9318994611765425d0 -0.02422116311440764d0 0.361907073039
```

Matlisp

The Matlisp scientific computation library provides high performance operations on arrays, including wrappers around BLAS and LAPACK functions. It can be loaded using quicklisp:

```
* (ql:quickload :matlisp)
```

The nickname for `matlisp` is `m`. To avoid typing `matlisp:` or `m:` in front of each symbol, you can define your own package which uses matlisp (See the PCL section on packages):

```
* (defpackage :my-new-code
     (:use :common-lisp :matlisp))
#<PACKAGE "MY-NEW-CODE">

* (in-package :my-new-code)
```

and to use the `#i` infix reader (note the same name as for `cmu-infix`), run:

```
* (named-readtables:in-readtable :infix-dispatch-table)
```

Creating tensors

```
* (matlisp:zeros '(2 2))
#<|<BLAS-MIXIN SIMPLE-DENSE-TENSOR: DOUBLE-FLOAT>| #(2 2)
  0.000     0.000
  0.000     0.000
>
```

Note that by default matrix storage types are `double-float`. To create a complex array using `zeros`, `ones` and `eye`, specify the type:

```
* (matlisp:zeros '(2 2) '((complex double-float)))
#<|<BLAS-MIXIN SIMPLE-DENSE-TENSOR: (COMPLEX DOUBLE-FLOAT)>| #(2
  0.000    0.000
  0.000    0.000
>
```

As well as `zeros` and `ones` there is `eye` which creates an identity matrix:

```
* (matlisp:eye '(3 3) '((complex double-float)))
#<|<BLAS-MIXIN SIMPLE-DENSE-TENSOR: (COMPLEX DOUBLE-FLOAT)>| #(3
  1.000    0.000    0.000
  0.000    1.000    0.000
  0.000    0.000    1.000
>
```

Ranges

To generate 1D arrays there are the `range` and `linspace` functions:

```
* (matlisp:range 1 10)
#<|<SIMPLE-DENSE-TENSOR: (INTEGER 0 4611686018427387903)>| #(9)
 1   2   3   4   5   6   7   8   9
>
```

The `range` function rounds down it's final argument to an integer:

```
* (matlisp:range 1 -3.5)
#<|<BLAS-MIXIN SIMPLE-DENSE-TENSOR: SINGLE-FLOAT>| #(5)
 1.000   0.000   -1.000  -2.000  -3.000
>
* (matlisp:range 1 3.3)
#<|<BLAS-MIXIN SIMPLE-DENSE-TENSOR: SINGLE-FLOAT>| #(3)
 1.000   2.000   3.000
>
```

Linspace is a bit more general, and the values returned include the end point.

```
* (matlisp:linspace 1 10)
#<|<SIMPLE-DENSE-TENSOR: (INTEGER 0 4611686018427387903)>| #(10)
 1   2   3   4   5   6   7   8   9   10
>
```

```
* (matlisp:linspace 0 (* 2 pi) 5)
#<|<BLAS-MIXIN SIMPLE-DENSE-TENSOR: DOUBLE-FLOAT>| #(5)
 0.000   1.571   3.142   4.712   6.283
>
```

Currently `linspace` requires real inputs, and doesn't work with complex numbers.

Random numbers

```
* (matlisp:random-uniform '(2 2))
#<|<BLAS-MIXIN SIMPLE-DENSE-TENSOR: DOUBLE-FLOAT>| #(2 2)
  0.7287        0.9480
  2.6703E-2     0.1834
>
```

```
(matlisp:random-normal '(2 2))
#<|<BLAS-MIXIN SIMPLE-DENSE-TENSOR: DOUBLE-FLOAT>| #(2 2)
  0.3536    -1.291
 -0.3877    -1.371
>
```

There are functions for other distributions, including `random-exponential`, `random-beta`, `random-gamma` and `random-pareto`.

Reader macros

The `#d` and `#e` reader macros provide a way to create `double-float` and `single-float` tensors:

```
* #d[1,2,3]
#<|<BLAS-MIXIN SIMPLE-DENSE-TENSOR: DOUBLE-FLOAT>| #(3)
 1.000   2.000   3.000
>
```

```
* #d[[1,2,3],[4,5,6]]
#<|<BLAS-MIXIN SIMPLE-DENSE-TENSOR: DOUBLE-FLOAT>| #(2 3)
  1.000    2.000    3.000
  4.000    5.000    6.000
>
```

Note that the comma separators are needed.

Tensors from arrays

Common lisp arrays can be converted to Matlisp tensors by copying:

```
* (copy #2A((1 2 3)
            (4 5 6))
        '#.(tensor 'double-float))
#<|<BLAS-MIXIN SIMPLE-DENSE-TENSOR: DOUBLE-FLOAT>| #(2 3)
  1.000    2.000    3.000
  4.000    5.000    6.000
>
```

Instances of the `tensor` class can also be created, specifying the dimensions. The internal storage of `tensor` objects is a 1D array (`simple-vector`) in a slot `store`.

For example, to create a `double-float` type tensor:

```
(make-instance (tensor 'double-float)
    :dimensions  (coerce '(2) '(simple-array index-type (*)))
    :store (make-array 2 :element-type 'double-float))
```

Arrays from tensors

The array store can be accessed using slots:

```
* (defparameter vec (m:range 0 5))
* vec
#<|<SIMPLE-DENSE-TENSOR: (INTEGER 0 4611686018427387903)>| #(5)
 0   1   2   3   4
>
* (slot-value vec 'm:store)
#(0 1 2 3 4)
```

Multidimensional tensors are also stored in 1D arrays, and are stored in column-major order rather than the row-major ordering used for common lisp arrays. A displaced array will therefore be transposed.

The contents of a tensor can be copied into an array

```
* (let ((tens (m:ones '(2 3))))
    (m:copy tens 'array))
#2A((1.0d0 1.0d0 1.0d0) (1.0d0 1.0d0 1.0d0))
```

or a list:

```
* (m:copy (m:ones '(2 3)) 'cons)
((1.0d0 1.0d0 1.0d0) (1.0d0 1.0d0 1.0d0))
```

Element access

The `ref` function is the equivalent of `aref` for standard CL arrays, and is also setf-able:

```
* (defparameter a (matlisp:ones '(2 3)))

* (setf (ref a 1 1) 2.0)
2.0d0
* a
#<|<BLAS-MIXIN SIMPLE-DENSE-TENSOR: DOUBLE-FLOAT>| #(2 3)
  1.000    1.000    1.000
  1.000    2.000    1.000
>
```

Element-wise operations

The `matlisp-user` package, loaded when `matlisp` is loaded, contains functions for operating element-wise on tensors.

```
* (matlisp-user:* 2 (ones '(2 3)))
#<|<BLAS-MIXIN SIMPLE-DENSE-TENSOR: DOUBLE-FLOAT>| #(2 3)
  2.000    2.000    2.000
  2.000    2.000    2.000
>
```

This includes arithmetic operators '+', '-', '*','/' and 'expt', but also `sqrt`,`sin`,`cos`,`tan`, hyperbolic functions, and their inverses. The `#i` reader macro recognises many of these, and uses the `matlisp-user` functions:

```
* (let ((a (ones '(2 2)))
        (b (random-normal '(2 2))))
     #i( 2 * a + b ))
#<|<BLAS-MIXIN SIMPLE-DENSE-TENSOR: DOUBLE-FLOAT>| #(2 2)
  0.9684    3.250
  1.593     1.508
>
```

```
* (let ((a (ones '(2 2)))
        (b (random-normal '(2 2))))
     (macroexpand-1 '#i( 2 * a + b )))
(MATLISP-USER:+ (MATLISP-USER:* 2 A) B)
```

Dates and Times

Common Lisp provides two different ways of looking at time: universal time, meaning time in the "real world", and run time, meaning time as seen by your computer's CPU. We will deal with both of them separately.

Built-in time functions

Universal Time

Universal time is represented as the number of seconds that have elapsed since 00:00 of January 1, 1900 in the GMT time zone. The function `get-universal-time` returns the current universal time:

```
CL-USER> (get-universal-time)
3220993326
```

Of course this value is not very readable, so you can use the function `decode-universal-time` to turn it into a "calendar time" representation:

```
CL-USER> (decode-universal-time 3220993326)
6
22
19
25
1
2002
4
NIL
5
```

NB: in the next section we'll use the `local-time` library to get more user-friendly functions, such as `(local-time:universal-to-timestamp (get-universal-time))` which returns `@2021-06-25T09:16:29.000000+02:00`.

This call to `decode-universal-time` returns nine values: `seconds, minutes, hours, day, month, year, day of the week, daylight savings time flag and time zone`. Note that the day of the week is represented as an integer in the range 0..6 with 0 being Monday and 6 being Sunday. Also, the **time zone** is represented as the number of hours you need to add to the current time in order to get GMT time.

So in this example the decoded time would be `19:22:06 of Friday, January 25, 2002`, in the EST time zone, with no daylight savings in effect. This, of course, relies on the computer's own clock, so make sure that it is set correctly (including the time zone you are in and the DST flag). As a shortcut, you can use [`get-decoded-time`]() to get the calendar time representation of the current time directly:

```
CL-USER> (get-decoded-time)
```

is equivalent to

```
CL-USER> (decode-universal-time (get-universal-time))
```

Here is an example of how to use these functions in a program (but frankly, use the `local-time` library instead):

```
CL-USER> (defconstant *day-names*
           '("Monday" "Tuesday" "Wednesday"
             "Thursday" "Friday" "Saturday"
             "Sunday"))
*DAY-NAMES*


CL-USER> (multiple-value-bind
           (second minute hour day month year day-of-week dst-p
           (get-decoded-time)
           (format t "It is now ~2,'0d:~2,'0d:~2,'0d of ~a, ~d/~
             hour
             minute
             second
             (nth day-of-week *day-names*)
             month
             day
             year
             (- tz)))
```

```
It is now 17:07:17 of Saturday, 1/26/2002 (GMT-5)
```

Of course the call to `get-decoded-time` above could be replaced by `(decode-universal-time n)`, where n is any integer number, to print an arbitrary date. You can also go the other way around: the function `encode-universal-time` lets you encode a calendar time into the corresponding universal time. This function takes six mandatory arguments (seconds, minutes, hours, day, month and year) and one optional argument (the time zone) and it returns a universal time:

```
CL-USER> (encode-universal-time 6 22 19 25 1 2002)
3220993326
```

Note that the result is automatically adjusted for daylight savings time if the time zone is not supplied. If it is supplied, than Lisp assumes that the specified time zone already accounts for daylight savings time, and no adjustment is performed.

Since universal times are simply numbers, they are easier and safer to manipulate than calendar times. Dates and times should always be stored as universal times if possible, and only converted to string representations for output purposes. For example, it is straightforward to know which of two dates came before the other, by simply comparing the two corresponding universal times with <.

Internal Time

Internal time is the time as measured by your Lisp environment, using your computer's clock. It differs from universal time in three important respects. First, internal time is not measured starting from a specified point in time: it could be measured from the instant you started your Lisp, from the instant you booted your machine, or from any other arbitrary time point in the past. As we will see shortly, the absolute value of an internal time is almost always meaningless; only differences between internal times are useful. The second difference is that internal time is not measured in seconds, but in a (usually smaller) unit whose value can be deduced from `internal-time-units-per-second`:

```
CL-USER> internal-time-units-per-second
1000
```

This means that in the Lisp environment used in this example, internal time is measured in milliseconds.

Finally, what is being measured by the “internal time” clock? There are actually two different internal time clocks in your Lisp:

- one of them measures the passage of “real” time (the same time that universal time measures, but in different units), and
- the other one measures the passage of CPU time, that is, the time your CPU spends doing actual computation for the current Lisp process.

On most modern computers these two times will be different, since your CPU will never be entirely dedicated to your program (even on single-user machines, the CPU has to devote part of its time to processing interrupts, performing I/O, etc). The two functions used to retrieve internal times are called `get-internal-real-time` and `get-internal-run-time` respectively. Using them, we can solve the above problem about measuring a function’s run time, which is what the `time` built-in macro does.

```
CL-USER> (time (sleep 1))
Evaluation took:
  1.000 seconds of real time
  0.000049 seconds of total run time (0.000044 user, 0.000005 sy
  0.00% CPU
  2,594,553,447 processor cycles
  0 bytes consed
```

The `local-time` library

The local-time library (GitHub) is a very handy extension to the somewhat limited functionalities as defined by the standard.

In particular, it can

- print timestamps in various standard or custom formats (e.g. RFC1123 or RFC3339)
- parse timestrings,
- perform time arithmetic,
- convert Unix times, timestamps, and universal times to and from.

We present below what we find the most useful functions. See its manual for the full details.

It is available in Quicklisp:

```
CL-USER> (ql:quickload "local-time")
```

Create timestamps (encode-timestamp, universal-to-timestamp)

Create a timestamp with `encode-timestamp`, giving it its number of nanoseconds, seconds, minutes, days, months and years:

```
(local-time:encode-timestamp 0 0 0 0 1 1 1984)
@1984-01-01T00:00:00.000000+01:00
```

The complete signature is:

encode-timestamp nsec sec minute hour day month year &key timezone offset into

The offset is the number of seconds offset from UTC of the locale. If offset is not specified, the offset will be guessed from the timezone. If a timestamp is passed as the into argument, its value will be set and that timestamp will be returned. Otherwise, a new timestamp is created.

Create a timestamp from a universal time with `universal-to-timestamp`:

```
(get-universal-time)
3833588757
(local-time:universal-to-timestamp (get-universal-time))
@2021-06-25T07:45:59.000000+02:00
```

You can also parse a human-readable time string:

```
(local-time:parse-timestring "1984-01-01")
@1984-01-01T01:00:00.000000+01:00
```

But see the section on parsing timestrings for more.

Get today’s date (now, today)

Use `now` or `today`:

```
(local-time:now)
@2019-11-13T20:02:13.529541+01:00

(local-time:today)
@2019-11-13T01:00:00.000000+01:00
```

“today” is the midnight of the current day in the UTC zone.

To compute “yesterday” and “tomorrow”, see below.

Add or substract times (timestamp+, timestamp-)

Use `timestamp+` and `timestamp-`. Each takes 3 arguments: a date, a number and a unit (and optionally a timezone and an offset):

```
(local-time:now)
@2021-06-25T07:19:39.836973+02:00

(local-time:timestamp+ (local-time:now) 1 :day)
@2021-06-26T07:16:58.086226+02:00

(local-time:timestamp- (local-time:now) 1 :day)
@2021-06-24T07:17:02.861763+02:00
```

The available units are `:sec :minute :hour :day :year`.

This operation is also possible with `adjust-timestamp`, which can do a bit more as we’ll see right in the next section (it can do many operations at once).

```
(local-time:timestamp+ (today) 3 :day)
@2021-06-28T02:00:00.000000+02:00

(local-time:adjust-timestamp (today) (offset :day 3))
@2021-06-28T02:00:00.000000+02:00
```

Here's `yesterday` and `tomorrow` defined from `today`:

```
(defun yesterday ()
  "Returns a timestamp representing the day before today."
  (timestamp- (today) 1 :day))

(defun tomorrow ()
  "Returns a timestamp representing the day after today."
  (timestamp+ (today) 1 :day))
```

Modify timestamps with any offset (adjust-timestamp)

`adjust-timestamp`'s first argument is the timestamp we operate on, and then it accepts a full `&body changes` where a "change" is in the form `(offset :part value)`:

Please point to the previous Monday:

```
(local-time:adjust-timestamp (today) (offset :day-of-week :monda
@2021-06-21T02:00:00.000000+02:00
```

We can apply many changes at once. Travel in time:

```
(local-time:adjust-timestamp (today)
  (offset :day 3)
  (offset :year 110)
  (offset :month -1))
@2131-05-28T02:00:00.000000+01:00
```

There is a destructive version, `adjust-timestamp!`.

Compare timestamps (timestamp<, timestamp<, timestamp= …)

These should be self-explanatory.

```
timestamp< time-a time-b
timestamp<= time-a time-b
timestamp> time-a time-b
timestamp>= time-a time-b
timestamp= time-a time-b
timestamp/= time-a time-b
```

Find the minimum or maximum timestamp

Use `timestamp-minimum` and `timestamp-maximum`. They accept any number of arguments.

```
(local-time:timestamp-minimum (local-time:today)
                              (local-time:timestamp- (local-time
@1921-06-25T02:00:00.000000+01:00
```

If you have a list of timestamps, use `(apply #'timestamp-minimum <your list of timestamps>)`.

Maximize or minimize a timestamp according to a time unit (timestamp-maximize-part, timestamp-minimize-part)

We can answer quite a number of questions with this handy function.

Here’s an example: please give me the last day of this month:

```
(let ((in-february (local-time:parse-timestring "1984-02-01")))
  (local-time:timestamp-maximize-part in-february :day))

@1984-02-29T23:59:59.999999+01:00
```

Querying timestamp objects (get the day, the day of week, the days in month…)

Use:

```
timestamp-[year, month, day, hour, minute, second, millisecond
```

```
timestamp [year, month, day, hour, minute, second, millisecond,
           day-of-week (starts at 0 for sunday),
           millenium, century, decade]
```

Get all the values at once with decode-timestamp.

Bind a variable to a value of your choice with this convenient macro:

```
(local-time:with-decoded-timestamp (:hour h)
     (now)
   (print h))

8
8
```

You can of course bind each time unit (:sec :minute :day) to its variable, in any order.

See also (days-in-month <month> <year>).

Formatting time strings (format, format-timestring, +iso-8601-format+)

local-time's date representation starts with @. We can format them as usual, with the aesthetic directive for instance, to get a usual date representation.

```
(local-time:now)
@2019-11-13T18:07:57.425654+01:00

(format nil "~a" (local-time:now))
"2019-11-13T18:08:23.312664+01:00"
```

We can use format-timestring, which can be used like format (thus it takes a stream as first argument):

```
(local-time:format-timestring nil (local-time:now))
"2019-11-13T18:09:06.313650+01:00"
```

Here nil returns a new string. t would print to *standard-output*.

But `format-timestring` also accepts a `:format` argument. We can use predefined date formats as well as give our own in s-expression friendly way (see next section).

Its default value is `+iso-8601-format+`, with the output shown above. The `+rfc3339-format+` format defaults to it.

With `+rfc-1123-format+`:

```
(local-time:format-timestring nil (local-time:now) :format local
"Wed, 13 Nov 2019 18:11:38 +0100"
```

With `+asctime-format+`:

```
(local-time:format-timestring nil (local-time:now) :format local
"Wed Nov 13 18:13:15 2019"
```

With `+iso-week-date-format+`:

```
(local-time:format-timestring nil (local-time:now) :format local
"2019-W46-3"
```

Putting all this together, here is a function that returns Unix times as a human readable string:

```
(defun unix-time-to-human-string (unix-time)
  (local-time:format-timestring
   nil
   (local-time:unix-to-timestamp unix-time)
   :format local-time:+asctime-format+))

(unix-time-to-human-string (get-universal-time))
"Mon Jun 25 06:46:49 2091"
```

Defining format strings (format-timestring (:year "-" :month "-" :day))

We can pass a custom `:format` argument to `format-timestring`.

The syntax consists of a list made of symbols with special meanings (`:year`, `:day`…), strings and characters:

```
(local-time:format-timestring nil (local-time:now) :format '(:ye
"2019-11-13"
```

The list of symbols is available in the documentation: https://common-lisp.net/project/local-time/manual.html#Parsing-and-Formatting

There are `:year :month :day :weekday :hour :hour12 :min :sec :msec`, long and short notations (`:long-weekday` for "Monday", `:short-weekday` for "Mon.", `:minimal-weekday` for "Mo." as well as `:long-month` for "January" and `:short-month` for "Jan."), gmt offset, timezone markers, `:ampm`, `:ordinal-day` (1st, 23rd), iso numbers and more.

The `+rfc-1123-format+` itself is defined like this:

```
(defparameter +rfc-1123-format+
  ;; Sun, 06 Nov 1994 08:49:37 GMT
  '(:short-weekday ", " (:day 2) #\space :short-month #\space (:
    (:hour 2) #\: (:min 2) #\: (:sec 2) #\space :gmt-offset-hhmm
  "See the RFC 1123 for the details about the possible values of
```

We see the form `(:day 2)`: the 2 is for **padding**, to ensure that the day is printed with two digits (not only `1`, but `01`). There could be an optional third argument, the character with which to fill the padding (by default, `#\0`).

Parsing time strings

Use `parse-timestring` to parse timestrings, in the form `2019-11-13T18:09:06.313650+01:00`. It works in a variety of formats by default, and we can change parameters to adapt it to our needs.

To parse more formats such as "Thu Jul 23 19:42:23 2013" (asctime), we'll use the cl-date-time-parser library.

The `parse-timestring` docstring is:

> Parses a timestring and returns the corresponding timestamp. Parsing begins at start and stops at the end position. If there are invalid characters within timestring and fail-on-error is T, then an invalid-timestring error is signaled, otherwise NIL is returned.
>
> If there is no timezone specified in timestring then offset is used as the default timezone offset (in seconds).

Examples:

```
(local-time:parse-timestring "2019-11-13T18:09:06.313650+01:00")
;; @2019-11-13T18:09:06.313650+01:00
```

```
(local-time:parse-timestring "2019-11-13")
;; @2019-11-13T01:00:00.000000+01:00
```

This custom format fails by default: "2019/11/13", but we can set the `:date-separator` to "/":

```
(local-time:parse-timestring "2019/11/13" :date-separator #\/)
;; @2019-11-13T19:42:32.394092+01:00
```

There is also a `:time-separator` (defaulting to `#\:`) and `:date-time-separator` (`#\T`).

Other options include:

- the start and end positions
- `fail-on-error` (defaults to `t`)
- `(allow-missing-elements t)`
- `(allow-missing-date-part allow-missing-elements)`
- `(allow-missing-time-part allow-missing-elements)`
- `(allow-missing-timezone-part allow-missing-elements)`
- `(offset 0)`

Now a format like ""Wed Nov 13 18:13:15 2019" will fail. We'll use the `cl-date-time-parser` library:

```lisp
(cl-date-time-parser:parse-date-time "Wed Nov 13 18:13:15 2019")
;; 3782657595
;; 0
```

It returns the universal time which, in turn, we can ingest with the local-time library:

```lisp
(local-time:universal-to-timestamp *)
;; @2019-11-13T19:13:15.000000+01:00
```

Misc

To find out if it’s Alice anniversary, use `timestamp-whole-year-difference time-a time-b`.

Pattern Matching

The ANSI Common Lisp standard does not include facilities for pattern matching, but libraries existed for this task and Trivia became a community standard.

For an introduction to the concepts of pattern matching, see Trivia's wiki.

Trivia matches against *a lot* of lisp objects and is extensible.

The library is in Quicklisp:

```
(ql:quickload "trivia")
```

For the following examples, let's `use` the library:

```
(use-package :trivia)
```

Common destructuring patterns

`cons`

```
(match '(1 2 3)
  ((cons x y)
  ; ^^ pattern
   (print x)
   (print y)))
;; |-> 1
;; |-> (2 3)
```

`list`, `list*`

`list` is a strict pattern, it expects the length of the matched object to be the same length as its subpatterns.

```
(match '(something 2 3)
  ((list a b _)
   (values a b)))
SOMETHING
2
```

Without the _ placeholder, it would not match:

```
(match '(something 2 3)
  ((list a b)
   (values a b)))
NIL
```

The `list*` pattern is flexible on the object's length:

```
(match '(something 2 3)
  ((list* a b)
   (values a b)))
SOMETHING
(2 3)
```

```
(match '(1 2 . 3)
  ((list* _ _ x)
   x))
3
```

However pay attention that if `list*` receives only one object, that object is returned, regardless of whether or not it is a list:

```
(match #(0 1 2)
  ((list* a)
   a))
#(0 1 2)
```

This is related to the definition of `list*` in the HyperSpec: http://clhs.lisp.se/Body/f_list_.htm.

vector, vector*

`vector` checks if the object is a vector, if the lengths are the same, and if the contents matches against each subpatterns.

`vector*` is similar, but called a soft-match variant that allows if the length is larger-than-equal to the length of subpatterns.

```
(match #(1 2 3)
  ((vector _ x _)
   x))
;; -> 2

(match #(1 2 3 4)
  ((vector _ x _)
   x))
;; -> NIL : does not match

(match #(1 2 3 4)
  ((vector* _ x _)
   x))
;; -> 2 : soft match.
```

```
<vector-pattern> : vector      | simple-vector
                   bit-vector  | simple-bit-vector
                   string      | simple-string
                   base-string | simple-base-string | sequence
(<vector-pattern> &rest subpatterns)
```

Class and structure pattern

There are three styles that are equivalent:

```
(defstruct foo bar baz)
(defvar *x* (make-foo :bar 0 :baz 1)

(match *x*
  ;; make-instance style
  ((foo :bar a :baz b)
   (values a b))
  ;; with-slots style
  ((foo (bar a) (baz b))
   (values a b))
  ;; slot name style
  ((foo bar baz)
   (values bar baz)))
```

`type`, `satisfies`

The `type` pattern matches if the object is of type. `satisfies` matches if the predicate returns true for the object. A lambda form is acceptable.

`assoc`, `property`, `alist`, `plist`

All these patterns first check if the pattern is a list. If that is satisfied, then they obtain the contents, and the value is matched against the subpattern.

Array, simple-array, row-major-array patterns

See https://github.com/guicho271828/trivia/wiki/Type-Based-Destructuring-Patterns#array-simple-array-row-major-array-pattern !

Logic based patterns

We can combine any pattern with some logic.

`and`, `or`

The following:

```
(match x
  ((or (list 1 a)
       (cons a 3))
   a))
```

matches against both `(1 2)` and `(4 . 3)` and returns 2 and 4, respectively.

`not`

It does not match when subpattern matches. The variables used in the subpattern are not visible in the body.

Guards

Guards allow us to use patterns *and* to verify them against a predicate.

The syntax is `guard` + `subpattern` + `a test form`, and the body.

```
(match (list 2 5)
  ((guard (list x y)     ; subpattern1
          (= 10 (* x y))) ; test-form
   :ok))
```

If the subpattern is true, the test form is evaluated, and if it is true it is matched against subpattern1.

Nesting patterns

Patterns can be nested:

```
(match '(:a (3 4) 5)
  ((list :a (list _ c) _)
   c))
```

returns `4`.

See more

See special patterns: `place`, `bind` and `access`.

Regular Expressions

The ANSI Common Lisp standard does not include facilities for regular expressions, but a couple of libraries exist for this task, for instance: cl-ppcre.

See also the respective Cliki: regexp page for more links.

Note that some CL implementations include regexp facilities, notably CLISP and ALLEGRO CL. If in doubt, check your manual or ask your vendor.

The description provided below is far from complete, so don't forget to check the reference manual that comes along with the CL-PPCRE library.

PPCRE

CL-PPCRE (abbreviation for Portable Perl-compatible regular expressions) is a portable regular expression library for Common Lisp with a broad set of features and good performance. It has been ported to a number of Common Lisp implementations and can be easily installed (or added as a dependency) via Quicklisp:

```
(ql:quickload "cl-ppcre")
```

Basic operations with the CL-PPCRE library functions are described below.

Looking for matching patterns: scan, create-scanner

The `scan` function tries to match the given pattern and on success returns four multiple-values values - the start of the match, the end of the match, and two arrays denoting the beginnings and ends of register matches. On failure returns `NIL`.

A regular expression pattern can be compiled with the `create-scanner` function call. A “scanner” will be created that can be used by other functions.

For example:

```
(let ((ptrn (ppcre:create-scanner "(a)*b")))
  (ppcre:scan ptrn "xaaabd"))
```

will yield the same results as:

```
(ppcre:scan "(a)*b" "xaaabd")
```

but will require less time for repeated `scan` calls as parsing the expression and compiling it is done only once.

Extracting information

CL-PPCRE provides several ways to extract matching fragments.

all-matches, all-matches-as-strings

The function `all-matches-as-strings` is very handy: it returns a list of matches:

```
(ppcre:all-matches-as-strings "\\d+" "numbers: 1 10 42")
;; => ("1" "10" "42")
```

The function `all-matches` is similar, but it returns a list of positions:

```
(ppcre:all-matches "\\d+" "numbers: 1 10 42")
;; => (9 10 11 13 14 16)
```

Look carefully: it actually return a list containing the start and end positions of all matches: 9 and 10 are the start and end for the first number (1), and so on.

If you wanted to extract integers from this example string, simply map `parse-integer` to the result:

```
CL-USER> (ppcre:all-matches-as-strings "\\d+" "numbers: 1 10 42'
;; ("1" "10" "42")
CL-USER> (mapcar #'parse-integer *)
(1 10 42)
```

The two functions accept the usual `:start` and `:end` key arguments. Additionnaly, `all-matches-as-strings` accepts a `:sharedp` argument:

> If SHAREDP is true, the substrings may share structure with TARGET-STRING.

scan-to-strings, register-groups-bind

The `scan-to-strings` function is similar to `scan` but returns substrings of target-string instead of positions. This function returns two values on success: the whole match as a string plus an array of substrings (or NILs) corresponding to the matched registers.

The `register-groups-bind` function tries to match the given pattern against the target string and binds matching fragments with the given variables.

```
(ppcre:register-groups-bind (first second third fourth)
      ("((a)|(b)|(c))+" "abababc" :sharedp t)
    (list first second third fourth))
;; => ("c" "a" "b" "c")
```

CL-PPCRE also provides a shortcut for calling a function before assigning the matching fragment to the variable:

```
(ppcre:register-groups-bind
  (fname lname (#'parse-integer date month year))
      ("(\\w+)\\s+(\\w+)\\s+(\\d{1,2})\\.(\\d{1,2})\\.(\\d{4})"
       "Frank Zappa 21.12.1940")
    (list fname lname date month year))
;; => ("Frank" "Zappa" 21 12 1940)
```

Replacing text: regex-replace, regex-replace-all

```lisp
(ppcre:regex-replace "a" "abc" "A") ;; => "Abc"
;; or
(let ((pat (ppcre:create-scanner "a")))
  (ppcre:regex-replace pat "abc" "A"))
```

Syntactic sugar

You might like to use CL-PPCRE with the **cl-interpol** library. cl-interpol is a library for Common Lisp which modifies the reader in a way that introduces interpolation within strings similar to Perl, Scala, or Unix Shell scripts.

In addition to loading the CL-INTERPOL library, initialization call must be made to properly configure the Lisp reader. This is accomplished by either calling the `enable-interpol-syntax` function from the REPL or placing that call in the source file before using any of its features:

```lisp
(interpol:enable-interpol-syntax)
```

In this mode you can write regular expressions in-between `#?/` and `/`.

See more

- cl-ppcre on common-lisp-libraries.readthedocs.io and read on: `do-matches`, `do-matches-as-strings`, `do-register-groups`, `do-scans`, `parse-string`, `regex-apropos`, `quote-meta-chars`, `split`…

Input/Output

Redirecting the Standard Output of your Program

You do it like this:

```
(let ((*standard-output* <some form generating a stream>))
  ...)
```

Because `*STANDARD-OUTPUT*` is a dynamic variable, all references to it during execution of the body of the `LET` form refer to the stream that you bound it to. After exiting the `LET` form, the old value of `*STANDARD-OUTPUT*` is restored, no matter if the exit was by normal execution, a `RETURN-FROM` leaving the whole function, an exception, or what-have-you. (This is, incidentally, why global variables lose much of their brokenness in Common Lisp compared to other languages: since they can be bound for the execution of a specific form without the risk of losing their former value after the form has finished, their use is quite safe; they act much like additional parameters that are passed to every function.)

If the output of the program should go to a file, you can do the following:

```
(with-open-file (*standard-output* "somefile.dat"
                                   :direction :output
                                   :if-exists :supersede)
  ...)
```

`WITH-OPEN-FILE` opens the file - creating it if necessary - binds `*STANDARD-OUTPUT*`, executes its body, closes the file, and restores `*STANDARD-OUTPUT*` to its former value. It doesn't get more comfortable than this!

Faithful Output with Character Streams

By *faithful output* I mean that characters with codes between 0 and 255 will be written out as is. It means, that I can `(PRINC (CODE-CHAR 0..255) s)` to a stream and expect 8-bit bytes to be written out, which is not obvious in the times of Unicode and 16 or 32 bit character representations. It does *not* require that the characters ä, ß, or þ must have their `CHAR-CODE` in the range 0..255 - the implementation is free to use any code. But it does require that no `#\Newline` to CRLF translation takes place, among others.

Common Lisp has a long tradition of distinguishing character from byte (binary) I/O, e.g. `READ-BYTE` and `READ-CHAR` are in the standard. Some implementations let both functions be called interchangeably. Others allow either one or the other. (The simple stream proposal defines the notion of a *bivalent stream* where both are possible.)

Varying element-types are useful as some protocols rely on the ability to send 8-Bit output on a channel. E.g. with HTTP, the header is normally ASCII and ought to use CRLF as line terminators, whereas the body can have the MIME type application/octet-stream, where CRLF translation would destroy the data. (This is how the Netscape browser on MS-Windows destroys data sent by incorrectly configured Webservers which declare unknown files as having MIME type text/plain - the default in most Apache configurations).

What follows is a list of implementation dependent choices and behaviours and some code to experiment.

CLISP

On CLISP, faithful output is possible using

```
:external-format
(ext:make-encoding :charset 'charset:iso-8859-1
                   :line-terminator :unix)
```

You can also use `(SETF (STREAM-ELEMENT-TYPE F) '(UNSIGNED-BYTE 8))`, where the ability to `SETF` is a CLISP-specific extension. Using `:EXTERNAL-FORMAT :UNIX` will cause portability problems, since the default character set on MS-Windows is `CHARSET:CP1252`. `CHARSET:CP1252` doesn't allow output of e.g. `(CODE-CHAR #x81)`:

```
;*** - Character #\u0080 cannot be represented in the character
set CHARSET:CP1252
```

Characters with code > 127 cannot be represented in ASCII:

```
;*** - Character #\u0080 cannot be represented in the character
set CHARSET:ASCII
```

AllegroCL

`#+(AND ALLEGRO UNIX) :DEFAULT` (untested) - seems enough on UNIX, but would not work on the MS-Windows port of AllegroCL.

LispWorks

`:EXTERNAL-FORMAT '(:LATIN-1 :EOL-STYLE :LF)` (confirmed by Marc Battyani)

Example

Here's some sample code to play with:

```
(defvar *unicode-test-file* "faithtest-out.txt")

(defun generate-256 (&key (filename *unicode-test-file*)
              #+CLISP (charset 'charset:iso-8859-1)
                          external-format)
  (let ((e (or external-format
           #+CLISP (ext:make-encoding :charset charset
                          :line-terminator :unix))))
    (describe e)
    (with-open-file (f filename :direction :output

             :external-format e)
      (write-sequence
        (loop with s = (make-string 256)
          for i from 0 to 255
          do (setf (char s i) (code-char i))
```

```lisp
          do (setf (char s i) (code-char i))
          finally (return s))
       f)
      (file-position f))))

;(generate-256 :external-format :default)
;#+CLISP (generate-256 :external-format :unix)
;#+CLISP (generate-256 :external-format 'charset:ascii)
;(generate-256)

(defun check-256 (&optional (filename *unicode-test-file*))
  (with-open-file (f filename :direction :input
                  :element-type '(unsigned-byte 8))
    (loop for i from 0
      for c = (read-byte f nil nil)
      while c
      unless (= c i)
      do (format t "~&Position ~D found ~D(#x~X)." i c c)
      when (and (= i 33) (= c 32))
      do (let ((c (read-byte f)))
           (format t "~&Resync back 1 byte ~D(#x~X) - cause CRLF
    (file-length f)))

#| CLISP
(check-256 *unicode-test-file*)
(progn (generate-256 :external-format :unix) (check-256))
; uses UTF-8 -> 385 bytes

(progn (generate-256 :charset 'charset:iso-8859-1) (check-256))

(progn (generate-256 :external-format :default) (check-256))
; uses UTF-8 + CRLF(on MS-Windows) -> 387 bytes

(progn (generate-256 :external-format
  (ext:make-encoding :charset 'charset:iso-8859-1 :line-terminat
(progn (generate-256 :external-format


  (ext:make-encoding :charset 'charset:iso-8859-1 :line-terminat
|#
```

Fast Bulk I/O

If you need to copy a lot of data and the source and destination are both streams (of the same element type), it's very fast to use `READ-SEQUENCE` and `WRITE-SEQUENCE`:

```
(let ((buf (make-array 4096 :element-type (stream-element-type i
  (loop for pos = (read-sequence buf input-stream)
        while (plusp pos)
        do (write-sequence buf output-stream :end pos)))
```

Files and Directories

We'll see here a handful of functions and libraries to operate on files and directories.

In this chapter, we use mainly namestrings to specify filenames. In a recipe or two we also use pathnames.

Many functions will come from UIOP, so we suggest you have a look directly at it:

- UIOP/filesystem
- UIOP/pathname

Of course, do not miss:

- Files and File I/O in Practical Common Lisp

Getting the components of a pathname

File name (sans directory)

Use `file-namestring` to get a file name from a pathname:

```
(file-namestring #p"/path/to/file.lisp") ;; => "file.lisp"
```

File extension

The file extension is called "pathname type" in Lisp parlance:

```
(pathname-type "~/foo.org")  ;; => "org"
```

File basename

The basename is called the "pathname name" -

```
(pathname-name "~/foo.org")  ;; => "foo"
(pathname-name "~/foo")      ;; => "foo"
```

If a directory pathname has a trailing slash, `pathname-name` may return `nil`; use `pathname-directory` instead -

```
(pathname-name "~/foo/")     ;; => NIL
(first (last (pathname-directory #P"~/foo/"))) ;; => "foo"
```

Parent directory

```
(uiop:pathname-parent-directory-pathname #P"/foo/bar/quux/")
;; => #P"/foo/bar/"
```

Testing whether a file exists

Use the function [`probe-file`] which will return a [generalized boolean] - either `nil` if the file doesn't exists, or its [truename] (which might be different from the argument you supplied).

For more portability, use `uiop:probe-file*` or `uiop:file-exists-p` which will return the file pathname (if it exists).

```
$ ln -s /etc/passwd foo

* (probe-file "/etc/passwd")
#p"/etc/passwd"

* (probe-file "foo")
#p"/etc/passwd"

* (probe-file "bar")
NIL
```

Expanding a file or a directory name with a tilde (~)

For portability, use `uiop:native-namestring`:

```
(uiop:native-namestring "~/.emacs.d/")
"/home/me/.emacs.d/"
```

It also expand the tilde with files and directories that don’t exist:

```
(uiop:native-namestring "~/foo987.txt")
:: "/home/me/foo987.txt"
```

On several implementations (CCL, ABCL, ECL, CLISP, LispWorks), `namestring` works similarly. On SBCL, if the file or directory doesn’t exist, `namestring` doesn’t expand the path but returns the argument, with the tilde.

With files that exist, you can also use `truename`. But, at least on SBCL, it returns an error if the path doesn’t exist.

Turning a pathname into a string with Windows’ directory separator

Use again `uiop:native-namestring`:

```
CL-USER> (uiop:native-namestring #p"~/foo/")
"C:\\Users\\You\\foo\\"
```

See also `uiop:parse-native-namestring` for the inverse operation.

Creating directories

The function [ensure-directories-exist](#) creates the directories if they do not exist:

```
(ensure-directories-exist "foo/bar/baz/")
```

This may create `foo`, `bar` and `baz`. Don’t forget the trailing slash.

Deleting directories

Use `uiop:delete-directory-tree` with a pathname (`#p`), a trailing slash and the `:validate` key:

```
;; mkdir dirtest
(uiop:delete-directory-tree #p"dirtest/" :validate t)
```

You can use `pathname` around a string that designates a directory:

```
(defun rmdir (path)
  (uiop:delete-directory-tree (pathname path) :validate t))
```

UIOP also has `delete-empty-directory`

cl-fad has `(fad:delete-directory-and-files "dirtest")`.

Merging files and directories

Use `merge-pathnames`, with one thing to note: if you want to append directories, the second argument must have a trailing `/`.

As always, look at UIOP functions. We have a `uiop:merge-pathnames*` equivalent which fixes corner cases.

So, here's how to append a directory to another one:

```
(merge-pathnames "otherpath" "/home/vince/projects/")
;; important:                                          ^^
;; a trailing / denotes a directory.
;; => #P"/home/vince/projects/otherpath"
```

Look at the difference: if you don't include a trailing slash to either paths, `otherpath` and `projects` are seen as files, so `otherpath` is appended to the base directory containing `projects`:

```
(merge-pathnames "otherpath" "/home/vince/projects")
;; #P"/home/vince/otherpath"
;;               ^^ no "projects", because it was seen as a file
```

or again, with `otherpath/` (a trailing `/`) but `projects` seen as a file:

```
(merge-pathnames "otherpath/" "/home/vince/projects")
;; #P"/home/vince/otherpath/projects"
;;                ^^ inserted here
```

Get the current working directory (CWD)

Use `uiop/os:getcwd`:

```
(uiop/os:getcwd)
;; #P"/home/vince/projects/cl-cookbook/"
;;                                    ^ with a trailing slash, u
```

Get the current directory relative to a Lisp project

Use `asdf:system-relative-pathname system path`.

Say you are working inside `mysystem`. It has an ASDF system declaration, the system is loaded in your Lisp image. This ASDF file is somewhere on your filesystem and you want the path to `src/web/`. Do this:

```
(asdf:system-relative-pathname "mysystem" "src/web/")
;; => #P"/home/vince/projects/mysystem/src/web/"
```

This will work on another user's machine, where the system sources are located in another location.

Setting the current working directory

Use [`uiop:chdir`] *`path`*:

```
(uiop:chdir "/bin/")
0
```

The trailing slash in *path* is optional.

Or, to set for the current directory for the next operation only, use `uiop:with-current-directory`:

```
(let ((dir "/path/to/another/directory/"))
  (uiop:with-current-directory (dir)
      (directory-files "./")))
```

Opening a file

Common Lisp has `open` and `close` functions which resemble the functions of the same denominator from other programming languages you're probably familiar with. However, it is almost always recommendable to use the macro `with-open-file` instead. Not only will this macro open the file for you and close it when you're done, it'll also take care of it if your code leaves the body abnormally (such as by a use of throw). A typical use of `with-open-file` looks like this:

```
(with-open-file (str <_file-spec_>
    :direction <_direction_>
    :if-exists <_if-exists_>
    :if-does-not-exist <_if-does-not-exist_>)
  (your code here))
```

- `str` is a variable which'll be bound to the stream which is created by opening the file.
- `<_file-spec_>` will be a truename or a pathname.
- `<_direction_>` is usually `:input` (meaning you want to read from the file), `:output` (meaning you want to write to the file) or `:io` (which is for reading *and* writing at the same time) - the default is `:input`.
- `<_if-exists_>` specifies what to do if you want to open a file for writing and a file with that name already exists - this option is ignored if you just want to read from the file. The default is `:error` which means that an error is signalled. Other useful options are `:supersede` (meaning that the new file will replace the old one), `:append` (content is added to the file), `nil` (the stream variable will be bound to `nil`), and `:rename` (i.e. the old file is renamed).
- `<_if-does-not-exist_>` specifies what to do if the file you want to open does not exist. It is one of `:error` for signalling an error, `:create` for creating an empty file, or `nil` for binding the stream variable to `nil`. The default is, to be brief, to do the right thing depending on the other options you provided. See the CLHS for details.

Note that there are a lot more options to `with-open-file`. See [the CLHS entry for `open`] for all the details. You'll find some examples on how to use `with-open-file` below. Also note that you usually don't need to provide any keyword arguments if you just want to open an existing file for reading.

Reading files

Reading a file into a string or a list of lines

It's quite common to need to access the contents of a file in string form, or to get a list of lines.

uiop is included in ASDF (there is no extra library to install or system to load) and has the following functions:

```
(uiop:read-file-string "file.txt")
```

and

```
(uiop:read-file-lines "file.txt")
```

Otherwise, this can be achieved by using `read-line` or `read-char` functions, that probably won't be the best solution. The file might not be divided into multiple lines or reading one character at a time might bring significant performance problems. To solve this problems, you can read files using buckets of specific sizes.

```
(with-output-to-string (out)
  (with-open-file (in "/path/to/big/file")
    (loop with buffer = (make-array 8192 :element-type 'characte
          for n-characters = (read-sequence buffer in)
          while (< 0 n-characters)
          do (write-sequence buffer out :start 0 :end n-characte
```

Furthermore, you're free to change the format of the read/written data, instead of using elements of type character every time. For instance, you can set `:element-type` type argument of `with-output-to-string`, `with-`

`open-file` and `make-array` functions to `'(unsigned-byte 8)` to read data in octets.

Reading with an utf-8 encoding

To avoid an `ASCII stream decoding error` you might want to specify an UTF-8 encoding:

```
(with-open-file (in "/path/to/big/file"
                     :external-format :utf-8)
                 ...
```

Set SBCL's default encoding format to utf-8

Sometimes you don't control the internals of a library, so you'd better set the default encoding to utf-8. Add this line to your `~/.sbclrc`:

(setf sb-impl::*default-external-format* :utf-8)

and optionally

(setf sb-alien::*default-c-string-external-format* :utf-8)

Reading a file one line at a time

`read-line` will read one line from a stream (which defaults to *standard input*) the end of which is determined by either a newline character or the end of the file. It will return this line as a string *without* the trailing newline character. (Note that `read-line` has a second return value which is true if there was no trailing newline, i.e. if the line was terminated by the end of the file.) `read-line` will by default signal an error if the end of the file is reached. You can inhibit this by supplying NIL as the second argument. If you do this, `read-line` will return `nil` if it reaches the end of the file.

```
(with-open-file (stream "/etc/passwd")
  (do ((line (read-line stream nil)
       (read-line stream nil)))
```

```
       ((null line))
       (print line)))
```

You can also supply a third argument which will be used instead of `nil` to signal the end of the file:

```
(with-open-file (stream "/etc/passwd")
  (loop for line = (read-line stream nil 'foo)
   until (eq line 'foo)
   do (print line)))
```

Reading a file one character at a time

read-char is similar to `read-line`, but it only reads one character as opposed to one line. Of course, newline characters aren't treated differently from other characters by this function.

```
(with-open-file (stream "/etc/passwd")
  (do ((char (read-char stream nil)
       (read-char stream nil)))
       ((null char))
       (print char)))
```

Looking one character ahead

You can 'look at' the next character of a stream without actually removing it from there - this is what the function peek-char is for. It can be used for three different purposes depending on its first (optional) argument (the second one being the stream it reads from): If the first argument is `nil`, `peek-char` will just return the next character that's waiting on the stream:

```
CL-USER> (with-input-from-string (stream "I'm not amused")
           (print (read-char stream))
           (print (peek-char nil stream))
           (print (read-char stream))
           (values))

#\I
```

```
#\'
#\'
```

If the first argument is `T`, `peek-char` will skip whitespace characters, i.e. it will return the next non-whitespace character that's waiting on the stream. The whitespace characters will vanish from the stream as if they had been read by `read-char`:

```
CL-USER> (with-input-from-string (stream "I'm not amused")
           (print (read-char stream))
           (print (read-char stream))
           (print (read-char stream))
           (print (peek-char t stream))
           (print (read-char stream))
           (print (read-char stream))
           (values))

#\I
#\'
#\m
#\n
#\n
#\o
```

If the first argument to `peek-char` is a character, the function will skip all characters until that particular character is found:

```
CL-USER> (with-input-from-string (stream "I'm not amused")
           (print (read-char stream))
           (print (peek-char #\a stream))
           (print (read-char stream))
           (print (read-char stream))
           (values))

#\I
#\a
#\a
#\m
```

Note that `peek-char` has further optional arguments to control its behaviour on end-of-file similar to those for `read-line` and `read-char` (and it will signal an error by default):

```
CL-USER> (with-input-from-string (stream "I'm not amused")
           (print (read-char stream))
           (print (peek-char #\d stream))
           (print (read-char stream))
           (print (peek-char nil stream nil 'the-end))
           (values))

#\I
#\d
#\d
THE-END
```

You can also put one character back onto the stream with the function [unread-char](). You can use it as if, *after* you have read a character, you decide that you'd better used `peek-char` instead of `read-char`:

```
CL-USER> (with-input-from-string (stream "I'm not amused")
           (let ((c (read-char stream)))
             (print c)
             (unread-char c stream)
             (print (read-char stream))
             (values)))

#\I
#\I
```

Note that the front of a stream doesn't behave like a stack: You can only put back exactly *one* character onto the stream. Also, you *must* put back the same character that has been read previously, and you can't unread a character if none has been read before.

Random access to a File

Use the function [file-position]() for random access to a file. If this function is used with one argument (a stream), it will return the current position within the stream. If it's used with two arguments (see below), it will actually change the [file position]() in the stream.

```
CL-USER> (with-input-from-string (stream "I'm not amused")
           (print (file-position stream))
           (print (read-char stream))
           (print (file-position stream))
           (file-position stream 4)
           (print (file-position stream))
           (print (read-char stream))
           (print (file-position stream))
           (values))

0
#\I
1
4
#\n
5
```

Writing content to a file

With `with-open-file`, specify `:direction :output` and use `write-sequence` inside:

```
(with-open-file (f <pathname> :direction :output
                              :if-exists :supersede
                              :if-does-not-exist :create)
    (write-sequence s f))
```

If the file exists, you can also `:append` content to it.

If it doesn’t exist, you can `:error` out. See the standard for more details.

Using libraries

The library Alexandria has a function called write-string-into-file

```
(alexandria:write-string-into-file content "file.txt")
```

Alternatively, the library str has the `to-file` function.

```
(str:to-file "file.txt" content) ;; with optional options
```

Both `alexandria:write-string-into-file` and `str:to-file` take the same keyword arguments as `cl:open` that controls file creation: `:if-exists` and `if-does-not-exists`.

Getting file attributes (size, access time,…)

[Osicat](https://www.common-lisp.net/project/osicat/) is a lightweight operating system interface for Common Lisp on POSIX-like systems, including Windows. With Osicat we can get and set **environment variables** (now doable with `uiop:getenv`), manipulate **files and directories**, **pathnames** and a bit more.

[file-attributes](https://github.com/Shinmera/file-attributes/) is a newer and lighter OS portability library specifically for getting file attributes, using system calls (cffi).

SBCL with its `sb-posix` contrib can be used too.

File attributes (Osicat)

Once Osicat is installed, it also defines the `osicat-posix` system, which permits us to get file attributes.

```
(ql:quickload "osicat")

(let ((stat (osicat-posix:stat #P"./files.md")))
  (osicat-posix:stat-size stat))  ;; => 10629
```

We can get the other attributes with the following methods:

```
osicat-posix:stat-dev
osicat-posix:stat-gid
osicat-posix:stat-ino
osicat-posix:stat-uid
osicat-posix:stat-mode
osicat-posix:stat-rdev
osicat-posix:stat-size
osicat-posix:stat-atime
osicat-posix:stat-ctime
osicat-posix:stat-mtime
osicat-posix:stat-nlink
```

```
osicat-posix:stat-blocks
osicat-posix:stat-blksize
```

File attributes (file-attributes)

Install the library with

(ql:quickload "file-attributes")

Its package is `org.shirakumo.file-attributes`. You can use a package-local nickname for a shorter access to its functions, for example:

```
(uiop:add-package-local-nickname :file-attributes :org.shirakumo
```

Then simply use the functions:

- `access-time`, `modification-time`, `creation-time`. You can `setf` them.
- `owner`, `group`, and `attributes`. The values used are OS specific for these functions. The attributes flag can be decoded and encoded via a standardised form with `decode-attributes` and `encode-attributes`.

```
CL-USER> (file-attributes:decode-attributes
           (file-attributes:attributes #p"test.txt"))
(:READ-ONLY NIL :HIDDEN NIL :SYSTEM-FILE NIL :DIRECTORY NIL :ARC
 NIL :NORMAL NIL :TEMPORARY NIL :SPARSE NIL :LINK NIL :COMPRESSE
 NIL :NOT-INDEXED NIL :ENCRYPTED NIL :INTEGRITY NIL :VIRTUAL NIL
 :RECALL NIL)
```

See [its documentation](#).

File attributes (sb-posix)

This contrib is loaded by default on POSIX systems.

First get a stat object for a file, then get the stat you want:

```
CL-USER> (sb-posix:stat "test.txt")
#<SB-POSIX:STAT {10053FCBE3}>

CL-USER> (sb-posix:stat-mtime *)
1686671405
```

Listing files and directories

Some functions below return pathnames, so you might need the following:

```
(namestring #p"/foo/bar/baz.txt")           ==> "/foo/bar/baz.tx
(directory-namestring #p"/foo/bar/baz.txt") ==> "/foo/bar/"
(file-namestring #p"/foo/bar/baz.txt")      ==> "baz.txt"
```

Listing files in a directory

```
(uiop:directory-files "./")
```

Returns a list of pathnames:

```
(#P"/home/vince/projects/cl-cookbook/.emacs"
 #P"/home/vince/projects/cl-cookbook/.gitignore"
 #P"/home/vince/projects/cl-cookbook/AppendixA.jpg"
 #P"/home/vince/projects/cl-cookbook/AppendixB.jpg"
 #P"/home/vince/projects/cl-cookbook/AppendixC.jpg"
 #P"/home/vince/projects/cl-cookbook/CHANGELOG"
 #P"/home/vince/projects/cl-cookbook/CONTRIBUTING.md"
 […]
```

Listing sub-directories

```
(uiop:subdirectories "./")

(#P"/home/vince/projects/cl-cookbook/.git/"
 #P"/home/vince/projects/cl-cookbook/.sass-cache/"
 #P"/home/vince/projects/cl-cookbook/_includes/"
 #P"/home/vince/projects/cl-cookbook/_layouts/"
 #P"/home/vince/projects/cl-cookbook/_site/"
 #P"/home/vince/projects/cl-cookbook/assets/")
```

Traversing (walking) directories recursively

See `uiop/filesystem:collect-sub*directories`. It takes as arguments:

- a `directory`
- a `collectp` function
- a `recursep` function
- a `collector` function

Given a directory, when `collectp` returns true with the directory, call the `collector` function on the directory, and recurse each of its subdirectories on which `recursep` returns true.

This function will thus let you traverse a filesystem hierarchy, superseding the functionality of `cl-fad:walk-directory`.

The behavior in presence of symlinks is not portable. Use IOlib to handle such situations.

Examples:

- this collects only subdirectories:

```
(defparameter *dirs* nil "All recursive directories.")

(uiop:collect-sub*directories "~/cl-cookbook"
    (constantly t)
    (constantly t)
    (lambda (it) (push it *dirs*)))
```

- this collects files and subdirectories:

```
(let ((results))
    (uiop:collect-sub*directories
     "./"
     (constantly t)
     (constantly t)
     (lambda (subdir)
       (setf results
             (nconc results
```

```
                    ;; A detail: we return strings, not pathname
                    (loop for path in (append (uiop:subdirectori
                                              (uiop:directory-fi
                          collect (namestring path))))))
    results)
```

- we can do the same with the `cl-fad` library:

```
(cl-fad:walk-directory "./"
  (lambda (name)
     (format t "~A~%" name))
   :directories t)
```

- and of course, we can use an external tool: the good ol' unix `find`, or the newer `fd` (`fdfind` on Debian) that has a simpler syntax and filters out a set of common files and directories by default (node_modules, .git…):

```
(str:lines (uiop:run-program (list "find" ".") :output :string))
;; or
(str:lines (uiop:run-program (list "fdfind") :output :string))
```

Here with the help of the `str` library.

Finding files matching a pattern

Below we simply list files of a directory and check that their name contains a given string.

```
(remove-if-not (lambda (it)
                   (search "App" (namestring it)))
               (uiop:directory-files "./"))
```

```
(#P"/home/vince/projects/cl-cookbook/AppendixA.jpg"
 #P"/home/vince/projects/cl-cookbook/AppendixB.jpg"
 #P"/home/vince/projects/cl-cookbook/AppendixC.jpg")
```

We used `namestring` to convert a `pathname` to a string, thus a sequence that `search` can deal with.

Finding files with a wildcard

We can not transpose unix wildcards to portable Common Lisp.

In pathname strings we can use `*` and `**` as wildcards. This works in absolute and relative pathnames.

```
(directory #P"*.jpg")
```

```
(directory #P"**/*.png")
```

Change the default pathname

The concept of `.` denoting the current directory does not exist in portable Common Lisp. This may exist in specific filesystems and specific implementations.

Also `~` to denote the home directory does not exist. They may be recognized by some implementations as non-portable extensions.

`*default-pathname-defaults*`provides a default for some pathname operations.

```
(let ((*default-pathname-defaults* (pathname "/bin/")))
          (directory "*sh"))
(#P"/bin/zsh" #P"/bin/tcsh" #P"/bin/sh" #P"/bin/ksh" #P"/bin/csh
```

See also `(user-homedir-pathname)`.

Error and exception handling

Common Lisp has mechanisms for error and condition handling as found in other languages, and can do more.

What is a condition ?

> Just like in languages that support exception handling (Java, C++, Python, etc.), a condition represents, for the most part, an "exceptional" situation. However, even more so than those languages, *a condition in Common Lisp can represent a general situation where some branching in program logic needs to take place*, not necessarily due to some error condition. Due to the highly interactive nature of Lisp development (the Lisp image in conjunction with the REPL), this makes perfect sense in a language like Lisp rather than say, a language like Java or even Python, which has a very primitive REPL. In most cases, however, we may not need (or even allow) the interactivity that this system offers us. Thankfully, the same system works just as well even in non-interactive mode.
>
> z0ltan

Let's dive into it step by step. More resources are given afterwards.

Ignoring all errors, returning nil

Sometimes you know that a function can fail and you just want to ignore it: use ignore-errors:

```
(ignore-errors
  (/ 3 0))
; in: IGNORE-ERRORS (/ 3 0)
;     (/ 3 0)
;
; caught STYLE-WARNING:
;   Lisp error during constant folding:
```

```
;   arithmetic error DIVISION-BY-ZERO signalled
;   Operation was (/ 3 0).
;
; compilation unit finished
;   caught 1 STYLE-WARNING condition
NIL
#<DIVISION-BY-ZERO {1008FF5F13}>
```

We get a welcome `division-by-zero` warning but the code runs well and it returns two things: `nil` and the condition that was signaled. We could not choose what to return.

Remember that we can `inspect` the condition with a right click in Slime.

Catching any condition (handler-case)

`ignore-errors` is built from handler-case. We can write the previous example by catching the general `error` but now we can return whatever we want:

```
(handler-case (/ 3 0)
  (error (c)
    (format t "We caught a condition.~&")
    (values 0 c)))
; in: HANDLER-CASE (/ 3 0)
;     (/ 3 0)
;
; caught STYLE-WARNING:
;   Lisp error during constant folding:
;   Condition DIVISION-BY-ZERO was signalled.
;
; compilation unit finished
;   caught 1 STYLE-WARNING condition
We caught a condition.
0
#<DIVISION-BY-ZERO {1004846AE3}>
```

We also returned two values, 0 and the signaled condition.

The general form of `handler-case` is

```
(handler-case (code that errors out)
   (condition-type (the-condition) ;; <-- optional argument
      (code))
   (another-condition (the-condition)
       ...))
```

Catching a specific condition

We can specify what condition to handle:

```
(handler-case (/ 3 0)
  (division-by-zero (c)
    (format t "Caught division by zero: ~a~%" c)))
;; …
;; Caught division by zero: arithmetic error DIVISION-BY-ZERO si
;; Operation was (/ 3 0).
;; NIL
```

This workflow is similar to a try/catch as found in other languages, but we can do more.

handler-case VS handler-bind

`handler-case` is similar to the `try/catch` forms that we find in other languages.

handler-bind (see the next examples), is what to use when we need absolute control over what happens when a signal is raised. It allows us to use the debugger and restarts, either interactively or programmatically.

If some library doesn't catch all conditions and lets some bubble out to us, we can see the restarts (established by `restart-case`) anywhere deep in the stack, including restarts established by other libraries that this library called. And *we can see the stack trace*, with every frame that was called and, in some lisps, even see local variables and such. Once we `handler-case`, we “forget” about this, everything is unwound. `handler-bind` does *not* rewind the stack.

Before we properly see `handler-bind`, let’s study conditions and restarts.

Defining and making conditions

We define conditions with [define-condition]() and we make (initialize) them with [make-condition]().

```
(define-condition my-division-by-zero (error)
  ())

(make-condition 'my-division-by-zero)
;; #<MY-DIVISION-BY-ZERO {1005A5FE43}>
```

It’s better if we give more information to it when we create a condition, so let’s use slots:

```
(define-condition my-division-by-zero (error)
  ((dividend :initarg :dividend
             :initform nil
             :reader dividend)) ;; <-- we'll get the dividend wi
  (:documentation "Custom error when we encounter a division by
```

Now when we’ll “signal” or “throw” the condition in our code we’ll be able to populate it with information to be consumed later:

```
(make-condition 'my-division-by-zero :dividend 3)
;; #<MY-DIVISION-BY-ZERO {1005C18653}>
```

Note: here’s a quick reminder on classes, if you are not fully operational on the [Common Lisp Object System]().

```
(make-condition 'my-division-by-zero :dividend 3)
;;                                   ^^ this is the ":initarg"
```

and `:reader dividend` created a *generic function* that is a “getter” for the dividend of a `my-division-by-zero` object:

```
(make-condition 'my-division-by-zero :dividend 3)
;; #<MY-DIVISION-BY-ZERO {1005C18653}>
```

```
(dividend *)
;; 3
```

an ":accessor" would be both a getter and a setter.

So, the general form of `define-condition` looks and feels like a regular class definition, but despite the similarities, conditions are not standard objects.

A difference is that we can't use `slot-value` on slots.

Signaling (throwing) conditions: error, warn, signal

We can use error in two ways:

- `(error "some text")`: signals a condition of type simple-error, and opens-up the interactive debugger.
- `(error 'my-error :message "We did this and that and it didn't work.")`: creates and throws a custom condition with its slot "message" and opens-up the interactive debugger.

With our own condition we can do:

```
(error 'my-division-by-zero :dividend 3)
;; which is a shortcut for
(error (make-condition 'my-division-by-zero :dividend 3))
```

Throwing these conditions will enter the interactive debugger, where the user may select a restart.

`warn` will not enter the debugger (create warning conditions by subclassing simple-warning).

Use signal if you do not want to enter the debugger, but you still want to signal to the upper levels that something *exceptional* happened.

And that can be anything. For example, it can be used to track progress during an operation. You would create a condition with a `percent` slot, signal one when progress is made, and the higher level code would handle it and display it to the user. See the resources below for more.

Conditions hierarchy

The class precedence list of `simple-error` is `simple-error, simple-condition, error, serious-condition, condition, t`.

The class precedence list of `simple-warning` is `simple-warning, simple-condition, warning, condition, t`.

Custom error messages (:report)

So far, when throwing our error, we saw this default text in the debugger:

```
Condition COMMON-LISP-USER::MY-DIVISION-BY-ZERO was signalled.
   [Condition of type MY-DIVISION-BY-ZERO]
```

We can do better by giving a `:report` function in our condition declaration:

```
(define-condition my-division-by-zero (error)
  ((dividend :initarg :dividend
             :initform nil
             :accessor dividend))
  ;; the :report is the message into the debugger:
  (:report (lambda (condition stream)
     (format stream
             "You were going to divide ~a by zero.~&"
             (dividend condition)))))
```

Now:

```
(error 'my-division-by-zero :dividend 3)
;; Debugger:
;;
;; You were going to divide 3 by zero.
;;    [Condition of type MY-DIVISION-BY-ZERO]
```

Inspecting the stacktrace

That's another quick reminder, not a Slime tutorial. In the debugger, you can inspect the stacktrace, the arguments to the function calls, go to the erroneous source line (with `v` in Slime), execute code in the context (`e`), etc.

Often, you can edit a buggy function, compile it (with the `C-c C-c` shortcut in Slime), choose the "RETRY" restart and see your code pass.

All this depends on compiler options, wether it is optimized for debugging, speed or security.

See our debugging section.

Restarts, interactive choices in the debugger

Restarts are the choices we get in the debugger, which always has the `RETRY` and `ABORT` ones.

By *handling* restarts we can start over the operation as if the error didn't occur (as seen in the stack).

Using assert's optional restart

In its simple form `assert` does what we know:

```
(assert (realp 3))
;; NIL = passed
```

When the assertion fails, we are prompted into the debugger:

```
(defun divide (x y)
  (assert (not (zerop y)))
  (/ x y))

(divide 3 0)
;; The assertion (NOT #1=(ZEROP Y)) failed with #1# = T.
;;    [Condition of type SIMPLE-ERROR]
```

```
;;
;; Restarts:
;;  0: [CONTINUE] Retry assertion.
;;  1: [RETRY] Retry SLIME REPL evaluation request.
;;  …
```

It also accepts an optional parameter to offer to change values:

```
(defun divide (x y)
  (assert (not (zerop y))
          (y)   ;; list of values that we can change.
          "Y can not be zero. Please change it") ;; custom error
  (/ x y))
```

Now we get a new restart that offers to change the value of Y:

```
(divide 3 0)
;; Y can not be zero. Please change it
;;    [Condition of type SIMPLE-ERROR]
;;
;; Restarts:
;;  0: [CONTINUE] Retry assertion with new value for Y.  <--- ne
;;  1: [RETRY] Retry SLIME REPL evaluation request.
;;  …
```

and when we choose it, we are prompted for a new value in the REPL:

```
The old value of Y is 0.
Do you want to supply a new value?  (y or n) y

Type a form to be evaluated:
2
3/2  ;; and our result.
```

Defining restarts (restart-case)

All this is good but we might want more custom choices. We can add restarts on the top of the list by wrapping our function call inside restart-case.

```lisp
(defun divide-with-restarts (x y)
  (restart-case (/ x y)
    (return-zero ()  ;; <-- creates a new restart called "RETURN
      0)
    (divide-by-one ()
      (/ x 1))))
(divide-with-restarts 3 0)
```

In case of *any error* (we’ll improve on that with `handler-bind`), we’ll get those two new choices at the top of the debugger:

```
arithmetic error DIVISION-BY-ZERO signalled
Operation was (/ 3 0).
   [Condition of type DIVISION-BY-ZERO]

Restarts:
 0: [RETURN-ZERO] RETURN-ZERO
 1: [DIVIDE-BY-ONE] DIVIDE-BY-ONE
 2: [RETRY] Retry SLIME REPL evaluation request.
 3: [*ABORT] Return to SLIME's top level.
 4: [ABORT] abort thread (#<THREAD "repl-thread" RUNNING {1003A6FFA3}>)

Backtrace:
  0: (SB-KERNEL::INTEGER-/-INTEGER 3 0)
  1: (/ 3 0)
  2: (DIVISION-RESTARTER)
  3: (SB-INT:SIMPLE-EVAL-IN-LEXENV (DIVISION-RESTARTER) #<NULL-LEXENV>)
  4: (EVAL (DIVISION-RESTARTER))
 --more--
```

That’s allright but let’s just write more human-friendy “reports”:

```lisp
(defun divide-with-restarts (x y)
  (restart-case (/ x y)
    (return-zero ()
      :report "Return 0"  ;; <-- added
      0)
    (divide-by-one ()
      :report "Divide by 1"
      (/ x 1))))
(divide-with-restarts 3 0)
;; Nicer restarts:
;;  0: [RETURN-ZERO] Return 0
;;  1: [DIVIDE-BY-ONE] Divide by 1
```

That’s better, but we lack the ability to change an operand, as we did with the `assert` example above.

Changing a variable with restarts

The two restarts we defined didn’t ask for a new value. To do this, we add an `:interactive` lambda function to the restart, that asks for the user a new value with the input method of its choice. Here, we’ll use the regular `read`.

```
(defun divide-with-restarts (x y)
  (restart-case (/ x y)
    (return-zero ()
      :report "Return 0"
      0)
    (divide-by-one ()
      :report "Divide by 1"
      (/ x 1))
    (set-new-divisor (value)
      :report "Enter a new divisor"
      ;;
      ;; Ask the user for a new value:
      :interactive (lambda () (prompt-new-value "Please enter a
      ;;
      ;; and call the divide function with the new value…
      ;; … possibly catching bad input again!
      (divide-with-restarts x value))))

(defun prompt-new-value (prompt)
  (format *query-io* prompt) ;; *query-io*: the special stream t
  (force-output *query-io*)  ;; Ensure the user sees what he typ
  (list (read *query-io*)))  ;; We must return a list.

(divide-with-restarts 3 0)
```

When calling it, we are offered a new restart, we enter a new value, and we get our result:

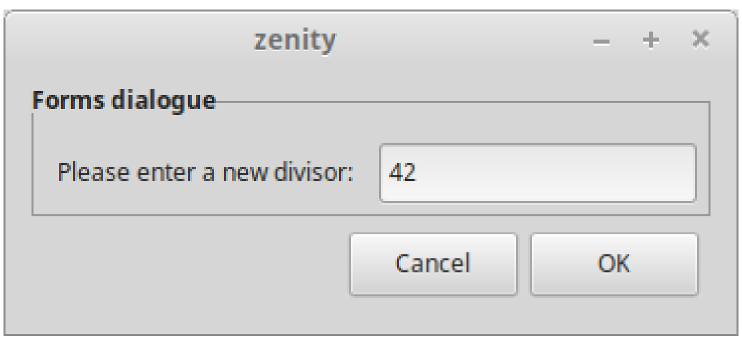
```
(divide-with-restarts 3 0)
;; Debugger:
;;
;; 2: [SET-NEW-DIVISOR] Enter a new divisor
```

```
;;
;; Please enter a new divisor: 10
;;
;; 3/10
```

Oh, you prefer a graphical user interface? We can use the `zenity` command line interface on GNU/Linux.

```
(defun prompt-new-value (prompt)
  (list
   (let ((input
          ;; We capture the program's output to a string.
          (with-output-to-string (s)
            (let* ((*standard-output* s))
              (uiop:run-program `("zenity"
                                  "--forms"
                                  ,(format nil "--add-entry=~a"
                                :output s)))))
     ;; We get a string and we want a number.
     ;; We could also use parse-integer, the parse-number librar
     (read-from-string input))))
```

Now try again and you should get a little window asking for a new number:



That’s fun, but that’s not all. Choosing restarts manually is not always (or often?) satisfactory. And by *handling* restarts we can start over the operation as if the error didn’t occur, as seen in the stack.

Calling restarts programmatically (handler-bind, invoke-restart)

We have a piece of code that we know can throw conditions. Here, `divide-with-restarts` can signal an error about a division by zero. What we want to do, is our higher-level code to automatically handle it and call the appropriate restart.

We can do this with `handler-bind` and invoke-restart:

```
(defun divide-and-handle-error (x y)
  (handler-bind
      ((division-by-zero (lambda (c)
                 (format t "Got error: ~a~%" c) ;; error-message
                 (format t "and will divide by 1~&")
                 (invoke-restart 'divide-by-one))))
    (divide-with-restarts x y)))

(divide-and-handle-error 3 0)
;; Got error: arithmetic error DIVISION-BY-ZERO signalled
;; Operation was (/ 3 0).
;; and will divide by 1
;; 3
```

Using other restarts (find-restart)

Use find-restart.

`find-restart 'name-of-restart` will return the most recent bound restart with the given name, or `nil`.

Hiding and showing restarts

Restarts can be hidden. In `restart-case`, in addition to `:report` and `:interactive`, they also accept a `:test` key:

```
(restart-case
   (return-zero ()
     :test (lambda ()
             (some-test))
    ...
```

Handling conditions (handler-bind)

We just saw a use for handler-bind.

Its general form is:

```
(handler-bind ((a-condition #'function-to-handle-it)
               (another-one #'another-function))
    (code that can...)
    (...error out))
```

If the handler returns normally (it declines to handle the condition), the condition continues to bubble up, searching for another handler, and it will find the interactive debugger (when it's an error, not when it's a simple condition).

We can study a real example with the `unix-opts` library, that parses command line arguments. It defined some conditions: `unknown-option`, `missing-arg` and `arg-parser-failed`, and it is up to us to write what to do in these cases.

```
(handler-bind ((opts:unknown-option #'unknown-option)
               (opts:missing-arg #'missing-arg)
               (opts:arg-parser-failed #'arg-parser-failed))
  (opts:get-opts))
```

Our `unknown-option` function is simple and looks like this:

```
(defun unknown-option (condition)
  (format t "~s option is unknown.~%" (opts:option condition))
  (opts:describe)
  (exit)) ;; <-- we return to the command line, no debugger.
```

it takes the condition as parameter, so we can read information from it if needed. Here we get the name of the erroneous option with the condition's reader `(opts:option condition)`.

Running some code, condition or not ("finally") (unwind-protect)

The “finally” part of others `try/catch/finally` forms is done with unwind-protect.

It is the construct used in “with-” macros, like `with-open-file`, which always closes the file after it.

With this example:

```
(unwind-protect (/ 3 0)
  (format t "This place is safe.~&"))
```

We *do* get the interactive debugger (we didn’t use handler-bind or anything), but our message is printed afterwards anyway.

Conclusion

You’re now more than ready to write some code and to dive into other resources!

Resources

- Practical Common Lisp: “Beyond Exception Handling: Conditions and Restarts” - the go-to tutorial, more explanations and primitives.
- Common Lisp Recipes, chap. 12, by E. Weitz
- language reference
- Video tutorial: introduction on conditions and restarts, by Patrick Stein.
- Condition Handling in the Lisp family of languages
- z0ltan.wordpress.com (the article this recipe is heavily based upon)

See also

- Algebraic effects - You can touch this ! - how to use conditions and restarts to implement progress reporting and aborting of a long-running calculation, possibly in an interactive or GUI context.
- A tutorial on conditions and restarts, based around computing the roots of a real function. It was presented by the author at a Bay Area Julia meetup on may 2019 (talk slides here).

- lisper.in - example with parsing a csv file and using restarts with success, in a flight travel company.
- https://github.com/svetlyak40wt/python-cl-conditions - implementation of the CL conditions system in Python.

Packages

See: The Complete Idiot's Guide to Common Lisp Packages

Creating a package

Here's an example package definition. It takes a name, and you probably want to `:use` the Common Lisp symbols and functions.

```
(defpackage :my-package
  (:use :cl))
```

To start writing code for this package, go inside it:

```
(in-package :my-package)
```

This `in-package` macro puts you "inside" a package:

- any new variable or function will be created in this package, aka in the "namespace" of this package.
- you can call all this package's symbols directly, without using the package prefix.

Just try!

We can also use `in-package` to try packages on the REPL. Note that on a new Lisp REPL session, we are "inside" the CL-USER package. It is a regular package.

Let's show you an example. We open a new .lisp file and we create a new package with a function inside our package:

```
;; in test-package.lisp
(defpackage :my-package
  (:use :cl))
```

```lisp
(in-package :my-package)

(defun hello ()
  (print "Hello from my package."))
```

This “hello” function lives inside “my-package”. It is not exported yet.

Continue below to see how to call it.

Accessing symbols from a package

As soon as you have defined a package or loaded one (with Quicklisp, or if it was defined as a dependency in your `.asd` system definition), you can access its symbols with `package:a-symbol`, using a colon as delimiter.

For example:

```lisp
(str:concat …)
```

When the symbol is not exported (it is “private”), use a double colon:

```lisp
(package::non-exported-symbol)
(my-package::hello)
```

Continuing our example: in the REPL, be sure to be in `my-package` and not in `CL-USER`. There you can call “hello” directly:

```lisp
CL-USER> (in-package :my-package)
#<PACKAGE "MY-PACKAGE">
;; ^^^ this creates a package object.
MY-PACKAGE> (hello)
;; ^^^^ the REPL shows you the current package.
"Hello from my package."
```

But now, come back to the CL-USER package and try to call “hello”: we get an error.

```lisp
MY-PACKAGE> (in-package :cl-user)
#<PACKAGE "COMMON-LISP-USER">
CL-USER> (hello)
```

```
=> you get the interactive debugger that says:

The function COMMON-LISP-USER::HELLO is undefined.

(quit)
```

We have to “namespace” our hello function with its package name:

```
CL-USER> (my-package::hello)
"Hello from my package."
```

Let’s export the function.

Exporting symbols

Augment our `defpackage` declaration to export our “hello” function like so:

```
(defpackage :my-package
  (:use :cl)
  (:export
   #:hello))
```

Compile this (`C-c C-c` in Slime), and now you can call

```
CL-USER> (my-package:hello)
```

with a single colon.

You can also use the `export` function:

```
(in-package :my-package)
(export #:hello)
```

Observation:

- exporting `:hello` without the sharpsign (`#:hello`) works too, but it will always create a new symbol. The `#:` notation does not create a new symbol. More precisely: it doesn’t *intern* a new symbol in our current package. It is a detail and at this point, a personal preference to

> use it or not. It can be helpful to not clutter our symbols namespace, specially when we import and re-export symbols from other libraries. That way, our editor's symbols completion only shows relevant results. It is not useful for us at this point, don't worry.

Now we might want to import individual symbols in order to access them right away, without the package prefix.

Importing symbols from another package

You can import exactly the symbols you need with `:import-from`:

```
(defpackage :my-package
  (:import-from :ppcre #:regex-replace)
  (:use :cl))
```

Now you can call `regex-replace` from inside `my-package`, without the `ppcre` package prefix. `regex-replace` is a new symbol inside your package. It is not exported.

Sometimes, we see `(:import-from :ppcre)`, without an explicit import. This helps people using ASDF's *package inferred system*.

You can also use the `import` function from outside a package definition:

```
CL-USER> (import 'ppcre:regex-replace)
CL-USER> (regex-replace …)
```

Importing all symbols

It is a better practice to carefully choose what symbols you import from another package (read below), but we can also import all symbols at once with `:use`:

```
(defpackage :my-package
  (:use :cl :ppcre))
```

Now you can access all variables, functions and macros of `cl-ppcre` from your `my-package` package.

You can also use the `use-package` function:

```
CL-USER> (use-package 'cl-ppcre)
```

About “use”-ing packages being a bad practice

`:use` is a well spread idiom. You could do:

```
(defpackage :my-package
  (:use :cl :ppcre))
```

and now, **all** symbols that are exported by `cl-ppcre` (aka `ppcre`) are available to use directly in your package. However, this should be considered bad practice, unless you `use` another package of your project that you control. Indeed, if the external package adds a symbol, it could conflict with one of yours, or you could add one which will hide the external symbol and you might not see a warning.

To quote this thorough explanation (a recommended read):

> USE is a bad idea in contemporary code except for internal packages that you fully control, where it is a decent idea until you forget that you mutate the symbol of some other package while making that brand new shiny DEFUN. USE is the reason why Alexandria cannot nowadays even add a new symbol to itself, because it might cause name collisions with other packages that already have a symbol with the same name from some external source.

List all Symbols in a Package (do-external-symbols)

Common Lisp provides some macros to iterate through the symbols of a package. The two most interesting are: `DO-SYMBOLS` and `DO-EXTERNAL-SYMBOLS`. `DO-SYMBOLS` iterates over the symbols accessible in the package and `DO-EXTERNAL-SYMBOLS` only iterates over the external symbols (you can see them as the real package API).

To print all exported symbols of a package named “PACKAGE”, you can write:

```
(do-external-symbols (s (find-package "PACKAGE"))
  (print s))
```

You can also collect all these symbols in a list by writing:

```
(let (symbols)
  (do-external-symbols (s (find-package "PACKAGE"))
    (push s symbols))
  symbols)
```

Or you can do it with `LOOP`.

```
(loop for s being the external-symbols of (find-package "PACKAGE
  collect s)
```

Package nickname

Package Local Nicknames (PLN)

Sometimes it is handy to give a local name to an imported package to save some typing, especially when the imported package does not provide nice global nicknames.

Many implementations (SBCL, CCL, ECL, Clasp, ABCL, ACL, LispWorks >= 7.2…) support Package Local Nicknames (PLN).

To use a PLN you can simply do the following, for example, if you’d like to try out a local nickname in an ad-hoc fashion:

```
(uiop:add-package-local-nickname :a :alexandria)
(a:iota 12) ; (0 1 2 3 4 5 6 7 8 9 10 11)
```

You can also set up a PLN in a `defpackage` form. The effect of PLN is totally within `mypackage` i.e. the `nickname` won’t work in other packages unless defined there too. So, you don't have to worry about unintended package name clash in other libraries.

```
(defpackage :mypackage
  (:use :cl)
  (:local-nicknames (:nickname :original-package-name)
                    (:alex :alexandria)
                    (:re :cl-ppcre)))

(in-package :mypackage)

;; You can use :nickname instead of :original-package-name
(nickname:some-function "a" "b")
```

Another facility exists for adding nicknames to packages. The function `RENAME-PACKAGE` can be used to replace the name and nicknames of a package. But it's use would mean that other libraries may not be able to access the package using the original name or nicknames. There is rarely any situation to use this. Use Package Local Nicknames instead.

Nickname Provided by Packages

When defining a package, it is trivial to give it a nickname for better user experience. But this mechanism is *global*, a nickname defined here is visible by all other packages everywhere. If you were thinking in giving a short name to a package you use often, you can get a conflict with another package. That's why *package-local* nicknames appeared. You should use them instead.

Here's an example anyways, from the `prove` package:

```
(defpackage prove
  (:nicknames :cl-test-more :test-more)
  (:export #:run
           #:is
           #:ok)
```

Afterwards, a user may use a nickname instead of the package name to refer to this package. For example:

```
(prove:run)
(cl-test-more:is)
(test-more:ok)
```

Please note that although Common Lisp allows defining multiple nicknames for one package, too many nicknames may bring maintenance complexity to the users. Thus the nicknames shall be meaningful and straightforward. For example:

```
(defpackage #:iterate
  (:nicknames #:iter))

(defpackage :cl-ppcre
  (:nicknames :ppcre)
```

Package locks

The package `common-lisp` and SBCL internal implementation packages are locked by default, including `sb-ext`.

In addition, any user-defined package can be declared to be locked so that it cannot be modified by the user. Attempts to change its symbol table or redefine functions which its symbols name result in an error.

More detailed information can be obtained from documents of SBCL and CLisp.

For example, if you try the following code:

```
(asdf:load-system :alexandria)
(rename-package :alexandria :alex)
```

You will get the following error (on SBCL):

```
Lock on package ALEXANDRIA violated when renaming as ALEX while
in package COMMON-LISP-USER.
   [Condition of type PACKAGE-LOCKED-ERROR]
See also:
  SBCL Manual, Package Locks [:node]

Restarts:
```

```
 0: [CONTINUE] Ignore the package lock.
 1: [IGNORE-ALL] Ignore all package locks in the context of
this operation.
 2: [UNLOCK-PACKAGE] Unlock the package.
 3: [RETRY] Retry SLIME REPL evaluation request.
 4: [*ABORT] Return to SLIME's top level.
 5: [ABORT] abort thread (#<THREAD "repl-thread" RUNNING
{10047A8433}>)

...
```

If a modification is required anyway, a package named cl-package-lock can be used to ignore package locks. For example:

```
(cl-package-locks:without-package-locks
  (rename-package :alexandria :alex))
```

See also

- Package Local Nicknames in Common Lisp article.

Macros

The word *macro* is used generally in computer science to mean a syntactic extension to a programming language. (Note: The name comes from the word “macro-instruction,” which was a useful feature of many second-generation assembly languages. A macro-instruction looked like a single instruction, but expanded into a sequence of actual instructions. The basic idea has since been used many times, notably in the C preprocessor. The name “macro” is perhaps not ideal, since it connotes nothing relevant to what it names, but we’re stuck with it.) Although many languages have a macro facility, none of them are as powerful as Lisp’s. The basic mechanism of Lisp macros is simple, but has subtle complexities, so learning your way around it takes a bit of practice.

How Macros Work

A macro is an ordinary piece of Lisp code that operates on *another piece of putative Lisp code*, translating it into (a version closer to) executable Lisp. That may sound a bit complicated, so let’s give a simple example. Suppose you want a version of `setq` that sets two variables to the same value. So if you write

```
(setq2 x y (+ z 3))
```

when `z`=8 then both `x` and `y` are set to 11. (I can’t think of any use for this, but it’s just an example.)

It should be obvious that we can’t define `setq2` as a function. If `x`=50 and `y`=-5, this function would receive the values 50, -5, and 11; it would have no knowledge of what variables were supposed to be set. What we really want to say is, When you (the Lisp system) see:

```
(setq2 v1 v2 e)
```

then treat it as equivalent to:

```
(progn
  (setq v1 e)
  (setq v2 e))
```

Actually, this isn't quite right, but it will do for now. A macro allows us to do precisely this, by specifying a program for transforming the input pattern `(setq2` v_1 v_2 e`)` into the output pattern `(progn ...)`.

Quote

Here's how we could define the `setq2` macro:

```
(defmacro setq2 (v1 v2 e)
  (list 'progn (list 'setq v1 e) (list 'setq v2 e)))
```

It takes as parameters two variables and one expression.

Then it returns a piece of code. In Lisp, because code is represented as lists, we can simply return a list that represents code.

We also use the *quote*, a *special operator* (not a function nor a macro, but one of a few special operators forming the core of Lisp).

Each *quoted* object evaluates to itself, aka it is returned as is:

- `(+ 1 2)` evaluates to `3` but `(quote (+ 1 2))` evaluates to `(+ 1 2)`
- `(quote (foo bar baz))` evaluates to `(foo bar baz)`
- `'` is a shortcut for `quote`: `(quote foo)` and `'foo` are equvalent - both evaluate to `foo`.

So, our macro returns the following bits:

- the symbol `progn`,
- a second list, that contains
  - the symbol `setq`
  - the variable `v1`: note that the variable is not evaluated inside the macro!
  - the expression `e`: it is not evaluated either!
- a second list, with `v2`.

We can use it like this:

```
(defparameter v1 1)
(defparameter v2 2)
(setq2 v1 v2 3)
;; 3
```

We can check, `v1` and `v2` were set to `3`.

Macroexpand

We must start writing a macro when we know what code we want to generate. Once we've begun writing one, it becomes very useful to check effectively what code does the macro generate. The function for that is `macroexpand`. It is a function, and we give it some code, as a list (so, we quote the code snippet we give it):

```
(macroexpand '(setq2 v1 v2 3))
;; (PROGN (SETQ V1 3) (SETQ V2 3))
;; T
```

Yay, our macro expands to the code we wanted!

More interestingly:

```
(macroexpand '(setq2 v1 v2 (+ z 3)))
;; (PROGN (SETQ V1 (+ z 3)) (SETQ V2 (+ z 3)))
;; T
```

We can confirm that our expression `e`, here `(+ z 3)`, was not evaluated. We will see how to control the evaluation of arguments with the comma: `,`.

Note: Slime tips

With Slime, you can call macroexpand by putting the cursor at the left of the parenthesis of the s-expr to expand and call the function`M-x slime-macroexpand-[1,all]`, or `C-c M-m`:

```
[|](setq2 v1 v2 3)
;^ cursor
; C-c M-m
; =>
; (PROGN (SETQ V1 3) (SETQ V2 3))
```

Another tip: on a macro name, type `C-c C-w m` (or `M-x slime-who-macroexpands`) to get a new buffer with all the places where the macro was expanded. Then type the usual `C-c C-k` (`slime-compile-and-load-file`) to recompile all of them.

Macros VS functions

Our macro is very close to the following function definition:

```
(defun setq2-function (v1 v2 e)
  (list 'progn (list 'setq v1 e) (list 'setq v2 e)))
```

If we evaluated `(setq2-function 'x 'y '(+ z 3))` (note that each argument is *quoted*, so it isn't evaluated when we call the function), we would get

```
(progn (setq x (+ z 3)) (setq y (+ z 3)))
```

This is a perfectly ordinary Lisp computation, whose sole point of interest is that its output is a piece of executable Lisp code. What `defmacro` does is create this function implicitly and make sure that whenever an expression of the form `(setq2 x y (+ z 3))` is seen, `setq2-function` is called with the pieces of the form as arguments, namely `x`, `y`, and `(+ z 3)`. The resulting piece of code then replaces the call to `setq2`, and execution resumes as if the new piece of code had occurred in the first place. The macro form is said to *expand* into the new piece of code.

Evaluation context

This is all there is to it, except, of course, for the myriad subtle consequences. The main consequence is that *run time for the `setq2` macro* is *compile time for its context.* That is, suppose the Lisp system is compiling a function, and midway through it finds the expression `(setq2 x y (+ z`

`3))`. The job of the compiler is, of course, to translate source code into something executable, such as machine language or perhaps byte code. Hence it doesn't execute the source code, but operates on it in various mysterious ways. However, once the compiler sees the `setq2` expression, it must suddenly switch to executing the body of the `setq2` macro. As I said, this is an ordinary piece of Lisp code, which can in principle do anything any other piece of Lisp code can do. That means that when the compiler is running, the entire Lisp (run-time) system must be present.

We'll stress this once more: at compile-time, you have the full language at your disposal.

Novices often make the following sort of mistake. Suppose that the `setq2` macro needs to do some complex transformation on its `e` argument before plugging it into the result. Suppose this transformation can be written as a Lisp procedure `some-computation`. The novice will often write:

```
(defmacro setq2 (v1 v2 e)
  (let ((e1 (some-computation e)))
    (list 'progn (list 'setq v1 e1) (list 'setq v2 e1))))

(defmacro some-computation (exp) ...) ;; _Wrong!_
```

The mistake is to suppose that once a macro is called, the Lisp system enters a "macro world," so naturally everything in that world must be defined using `defmacro`. This is the wrong picture. The right picture is that `defmacro` enables a step into the *ordinary Lisp world*, but in which the principal object of manipulation is Lisp code. Once that step is taken, one uses ordinary Lisp function definitions:

```
(defmacro setq2 (v1 v2 e)
  (let ((e1 (some-computation e)))
    (list 'progn (list 'setq v1 e1) (list 'setq v2 e1))))

(defun some-computation (exp) ...) ;; _Right!_
```

One possible explanation for this mistake may be that in other languages, such as C, invoking a preprocessor macro *does* get you into a different world; you can't run an arbitrary C program. It might be worth pausing to think about what it might mean to be able to.

Another subtle consequence is that we must spell out how the arguments to the macro get distributed to the hypothetical behind-the-scenes function (called `setq2-function` in my example). In most cases, it is easy to do so: In defining a macro, we use all the usual `lambda`-list syntax, such as `&optional`, `&rest`, `&key`, but what gets bound to the formal parameters are pieces of the macro form, not their values (which are mostly unknown, this being compile time for the macro form). So if we defined a macro thus:

```
(defmacro foo (x &optional y &key (cxt 'null)) ...)
```

then

- if we call it with `(foo a)`, the parameters' values are: `x=a`, `y=nil`, `cxt=null`.
- calling `(foo (+ a 1) (- y 1))` gives: `x=(+ a 1)`, `y=(- y 1)`, `cxt=null`.
- and `(foo a b :cxt (zap zip))` gives: `x=a`, `y=b`, `cxt=(zap zip)`.

Note that the values of the variables are the actual expressions `(+ a 1)` and `(zap zip)`. There is no requirement that these expressions' values be known, or even that they have values. The macro can do anything it likes with them. For instance, here's an even more useless variant of `setq`: `(setq-reversible` e_1 e_2 d`)` behaves like `(setq` e_1 e_2`)` if d=`:normal`, and behaves like `(setq` e_2 e_1`)` if d=`:backward`. It could be defined thus:

```
(defmacro setq-reversible (e1 e2 direction)
  (case direction
    (:normal (list 'setq e1 e2))
    (:backward (list 'setq e2 e1))
    (t (error "Unknown direction: ~a" direction))))
```

Here's how it expands:

```
(macroexpand '(setq-reversible x y :normal))
(SETQ X Y)
T
```

```
(macroexpand '(setq-reversible x y :backward))
(SETQ Y X)
T
```

And with a wrong direction:

```
(macroexpand '(setq-reversible x y :other-way-around))
```

We get an error and are prompted into the debugger!

We'll see the backquote and comma mechanism in the next section, but here's a fix:

```
(defmacro setq-reversible (v1 v2 direction)
  (case direction
    (:normal (list 'setq v1 v2))
    (:backward (list 'setq v2 v1))
    (t `(error "Unknown direction: ~a" ,direction))))
    ;; ^^ backquote                    ^^ comma: get the value i

(macroexpand '(SETQ-REVERSIBLE v1 v2 :other-way-around))

;; (ERROR "Unknown direction: ~a" :OTHER-WAY-AROUND)
;; T
```

Now when we call `(setq-reversible v1 v2 :other-way-around)` we still get the error and the debugger, but at least not when using `macroexpand`.

Backquote and comma

Before taking another step, we need to introduce a piece of Lisp notation that is indispensable to defining macros, even though technically it is quite independent of macros. This is the *backquote facility*. As we saw above, the main job of a macro, when all is said and done, is to define a piece of Lisp code, and that means evaluating expressions such as `(list 'prog (list 'setq ...) ...)`. As these expressions grow in complexity, it becomes hard to read them and write them. What we find ourselves wanting is a notation that provides the skeleton of an expression, with some of the pieces filled in with new expressions. That's what backquote provides. Instead of the `list` expression given above, one writes

```
  `(progn (setq ,v1 ,e) (setq ,v2 ,e))
;;^ backquote   ^   ^         ^   ^ commas
```

The backquote (`) character signals that in the expression that follows, every subexpression *not* preceded by a comma is to be quoted, and every subexpression preceded by a comma is to be evaluated.

You can think of it, and use it, as data interpolation:

```
`(v1 = ,v1) ;; => (V1 = 3)
```

That's mostly all there is to backquote. There are just two extra items to point out.

Comma-splice ,@

First, if you write "`,@e`" instead of "`,e`" then the value of *e* is *spliced* (or "joined", "combined", "interleaved") into the result. So if `v` equals `(oh boy)`, then

```
`(zap ,@v ,v)
```

evaluates to

```
(zap oh boy (oh boy))
;;   ^^^^^ elements of v (two elements), spliced.
;;          ^^ v itself (a list)
```

The second occurrence of `v` is replaced by its value. The first is replaced by the elements of its value. If `v` had had value `()`, it would have disappeared entirely: the value of `(zap ,@v ,v)` would have been `(zap ())`, which is the same as `(zap nil)`.

Quote-comma ',

When we are inside a backquote context and we want to print an expression literally, we have no choice but to use the combination of quote and comma:

```
(defmacro explain-exp (exp)
  `(format t "~S = ~S" ',exp ,exp))
  ;;                   ^^

(explain-exp (+ 2 3))
;; (+ 2 3) = 5
```

See by yourself:

```
;; Defmacro with no quote at all:
(defmacro explain-exp (exp)
  (format t "~a = ~a" exp exp))
(explain-exp v1)
;; V1 = V1

;; OK, with a backquote and a comma to get the value of exp:
(defmacro explain-exp (exp)
  ;; WRONG example
  `(format t "~a = ~a" exp ,exp))
(explain-exp v1)
;; => error: The variable EXP is unbound.

;; We then must use quote-comma:
(defmacro explain-exp (exp)
  `(format t "~a = ~a" ',exp ,exp))
(explain-exp (+ 1 2))
;; (+ 1 2) = 3
```

Nested backquotes

Second, one might wonder what happens if a backquote expression occurs inside another backquote. The answer is that the backquote becomes essentially unreadable and unwriteable; using nested backquote is usually a tedious debugging exercise. The reason, in my not-so-humble opinion, is that backquote is defined wrong. A comma pairs up with the innermost backquote when the default should be that it pairs up with the outermost.

But this is not the place for a rant; consult your favorite Lisp reference for the exact behavior of nested backquote plus some examples.

Building lists with backquote

One problem with backquote is that once you learn it you tend to use for every list-building occasion. For instance, you might write

```
(mapcan (lambda (x)
          (cond ((symbolp x) `((,x)))
                ((> x 10) `(,x ,x))
                (t '())))
        some-list)
```

which yields `((a) 15 15)` when `some-list` = `(a 6 15)`. The problem is that [`mapcan`] destructively alters the results returned by the [`lambda`]-expression. Can we be sure that the lists returned by that expression are "[fresh]," that is, they are different (in the [`eq`] sense) from the structures returned on other calls of that `lambda` expression? In the present case, close analysis will show that they must be fresh, but in general backquote is not obligated to return a fresh list every time (whether it does or not is implementation-dependent). If the example above got changed to

```
(mapcan (lambda (x)
          (cond ((symbolp x) `((,x)))
                ((> x 10) `(,x ,x))
                ((>= x 0) `(low))
                (t '())))
        some-list)
```

then backquote may well treat `(low)` as if it were `'(low)`; the list will be allocated at load time, and every time the `lambda` is evaluated, that same chunk of storage will be returned. So if we evaluate the expression with `some-list` = `(a 6 15)`, we will get `((a) low 15 15)`, but as a side effect the constant `(low)` will get clobbered to become `(low 15 15)`. If we then evaluate the expression with, say, `some-list` = `(8 oops)`, the result will be `(low 15 15 (oops))`, and now the "constant" that started off as `'(low)` will be `(low 15 15 (oops))`. (Note: The bug exemplified here takes other forms, and has often bit newbies - as well as experienced programmers - in the ass. The general form is that a constant list is produced as the value of something that is later destructively altered. The first line of defense against this bug is never to destructively alter any list. For newbies, this is also the last line of defense. For those of us who imagine we're more sophisticated, the next line of defense is to think very carefully any time you use `nconc` or `mapcan`).

To fix the bug, you can write `(map 'list ...)` instead of `mapcan`. However, if you are determined to use `mapcan`, write the expression this way:

```
(mapcan (lambda (x)
          (cond ((symbolp x) (list `(,x)))
                ((> x 10) (list x x))
                ((>= x 0) (list 'low))
                (t '())))
        some-list)
```

My personal preference is to use backquote *only* to build S-expressions, that is, hierarchical expressions that consist of symbols, numbers, and strings, and that are not conceptualized as changing in length. For instance, I would never write

```
(setq sk `(,x ,@sk))
```

If `sk` is being used as a stack, that is, it's going to be `pop`ped in the normal course of things, I would write `(push x sk)`. If not, I would write `(setq sk (cons x sk))`.

Getting Macros Right

I said in the first section that my definition of `setq2` wasn't quite right, and now it's time to fix it.

Suppose we write `(setq2 x y (+ x 2))`, when `x`=*8*. Then according to the definition given above, this form will expand into

```
(progn
  (setq x (+ x 2))
```

```
  (setq y (+ x 2)))
```

so that `x` will have value 10 and `y` will have value 12. Indeed, here's its macroexpansion:

```
(macroexpand '(setq2 x y (+ x 2)))
;;(PROGN (SETQ X (+ X 2)) (SETQ Y (+ X 2)))
```

Chances are that isn't what the macro is expected to do (although you never know). Another problematic case is `(setq2 x y (pop l))`, which causes `l` to be popped twice; again, probably not right.

The solution is to evaluate the expression `e` just once, save it in a temporary variable, and then set `v1` and `v2` to it.

Gensym

To make temporary variables, we use the `gensym` function, which returns a fresh variable guaranteed to appear nowhere else. Here is what the macro should look like:

```
(defmacro setq2 (v1 v2 e)
  (let ((tempvar (gensym)))
    `(let ((,tempvar ,e))
       (progn (setq ,v1 ,tempvar)
              (setq ,v2 ,tempvar)))))
```

Now `(setq2 x y (+ x 2))` expands to

```
(let ((#:g2003 (+ x 2)))
  (progn (setq x #:g2003) (setq y #:g2003)))
```

Here `gensym` has returned the symbol `#:g2003`, which prints in this funny way because it won't be recognized by the reader. (Nor is there any need for the reader to recognize it, since it exists only long enough for the code that contains it to be compiled.)

Exercise: Verify that this new version works correctly for the case `(setq2 x y (pop l1))`.

Exercise: Try writing the new version of the macro without using backquote. If you can't do it, you have done the exercise correctly, and learned what backquote is for!

The moral of this section is to think carefully about which expressions in a macro get evaluated and when. Be on the lookout for situations where the same expression gets plugged into the output twice (as `e` was in my original macro design). For complex macros, watch out for cases where the order that expressions are evaluated differs from the order in which they are written. This is sure to trip up some user of the macro - even if you are the only user.

What Macros are For

Macros are for making syntactic extensions to Lisp. One often hears it said that macros are a bad idea, that users can't be trusted with them, and so forth. Balderdash. It is just as reasonable to extend a language syntactically as to extend it by defining your own procedures. It may be true that the casual reader of your code can't understand the code without seeing the macro definitions, but then the casual reader can't understand it without seeing function definitions either. Having `defmethod`s strewn around several files contributes far more to unclarity than macros ever have, but that's a different diatribe.

Before surveying what sorts of syntactic extensions I have found useful, let me point out what sorts of syntactic extensions are generally *not* useful, or best accomplished using means other than macros. Some novices think macros are useful for open-coding functions. So, instead of defining

```
(defun sqone (x)
  (let ((y (+ x 1))) (* y y)))
```

they might define

```
(defmacro sqone (x)
  `(let ((y (+ ,x 1))) (* y y)))
```

So that `(sqone (* z 13))` might expand into

```
(let ((y (+ (* z 13) 1)))
  (* y y))
```

This is correct, but a waste of effort. For one thing, the amount of time saved is almost certainly negligible. If it's really important that `sqone` be expanded inline, one can put `(declaim (inline sqone))` before `sqone` is defined (although the compiler is not obligated to honor this declaration). For another, once `sqone` is defined as a macro, it becomes impossible to write `(mapcar #'sqone ll)`, or to do anything else with it except call it.

But macros have a thousand and one legitimate uses. Why write `(lambda (x) ...)` when you can write `(\\ (x) ...)`? Just define `\\` as a macro: `(defmacro \ (&rest list) `(lambda ,@list))`.

Many people find `mapcar` and `mapcan` a bit too obscure, especially when used with large `lambda` expressions. Rather than write something like

```
(mapcar (lambda (x)
          (let ((y (hairy-fun1 x))
                (z (hairy-fun2 x)))
            (dolist (y1 y)
              (dolist (z1 z)
                _... and further meaningless_
                _space-filling nonsense..._
                ))))
        list)
```

we might prefer to write

```
(for (x :in list)
     (let ((y (hairy-fun1 x))
           (z (hairy-fun2 x)))
       (dolist (y1 y)
         (dolist (z1 z)
           _... and further meaningless_
           _space-filling nonsense..._
           ))))
```

This macro might be defined thus:

```
(defmacro for (listspec exp)
  ;;           ^^ listspec = (x :in list), a list of length 3.
  ;;                    ^^ exp = the rest of the code.
  (cond
    ((and (= (length listspec) 3)
          (symbolp (first listspec))
          (eq (second listspec) ':in))
     `(mapcar (lambda (,(first listspec))
                ,exp)
              ,(third listspec)))
    (t (error "Ill-formed for spec: ~A" listspec)))))
```

(This is a simplified version of a macro by Chris Riesbeck.)

It's worth stopping for a second to discuss the role the keyword `:in` plays in this macro. It serves as a sort of "local syntax marker," in that it has no meaning as far as Lisp is concerned, but does serve as a syntactic guidepost for the macro itself. I will refer to these markers as *guide symbols*. (Here its job may seem trivial, but if we generalized the `for` macro to allow multiple list arguments and an implicit `progn` in the body the `:in`s would be crucial in telling us where the arguments stopped and the body began.)

It is not strictly necessary for the guide symbols of a macro to be in the [keyword package](), but it is a good idea, for two reasons. First, they highlight to the reader that something idiosyncratic is going on. A form like `(for ((x in (foobar a b 'oof))) (something-hairy x (list x)))` looks a bit wrong already, because of the double parentheses before the `x`. But using "`:in`" makes it more obvious.

Second, notice that I wrote `(eq (second listspec) ':in)` in the macro definition to check for the presence of the guide symbol. If I had used `in` instead, I would have had to think about which package *my* `in` lives in and which package the macro user's `in` lives in. One way to avoid trouble would be to write

```
(and (symbolp (second listspec))
     (eq (intern (symbol-name (second listspec))
                 :keyword)
         ':in))
```

Another would be to write

```
(and (symbolp (second listspec))
     (string= (symbol-name (second listspec)) (symbol-name 'in))
```

which neither of which is particularly clear or aesthetic. The keyword package is there to provide a home for symbols whose home is not per se relevant to anything; you might as well use it. (Note: In ANSI Lisp, I could have written `"IN"` instead of `(symbol-name 'in)`, but there are Lisp implementations that do not convert symbols' names to uppercase. Since I think the whole uppercase conversion idea is an embarrassing relic, I try to write code that is portable to those implementations.)

Let's look at another example, both to illustrate a nice macro, and to provide an auxiliary function for some of the discussion below. One often wants to create new symbols in Lisp, and `gensym` is not always adequate for building them. Here is a description of an alternative facility called `build-symbol`:

> `(build-symbol [(:package` *`p`*`)] -`*`pieces`*`-)` builds a symbol by concatenating the given *pieces* and interns it as specified by *p*. For each element of *pieces*, if it is a …
>
> - … string: The string is added to the new symbol's name.
> - … symbol: The name of the symbol is added to the new symbol's name.
> - … expression of the form `(:<` *`e`*`)`: *e* should evaluate to a string, symbol, or number; the characters of the value of *e* (as printed by `princ`) are concatenated into the new symbol's name.
> - … expression of the form `(:++` *`p`*`)`: *p* should be a place expression (i.e., appropriate as the first argument to `setf`), whose value is an integer; the value is incremented by 1, and the new value is concatenated into the new symbol's name.
>
> If the `:package` specification is omitted, it defaults to the value of `*package*`. If *p* is `nil`, the symbol is interned nowhere. Otherwise, it

should evaluate to a package designator (usually, a keyword whose name is the same of a package).

For example, `(build-symbol (:< x) "-" (:++ *x-num*))`, when `x` = `foo` and `*x-num*` = 8, sets `*x-num*` to 9 and evaluates to `FOO-9`. If evaluated again, the result will be `FOO-10`, and so forth.

Obviously, `build-symbol` can't be implemented as a function; it has to be a macro. Here is an implementation:

```
(defmacro build-symbol (&rest list)
  (let ((p (find-if (lambda (x)
                      (and (consp x)
                           (eq (car x) ':package)))
                    list)))
    (when p
      (setq list (remove p list)))
    (let ((pkg (cond ((eq (second p) 'nil)
                      nil)
                     (t `(find-package ',(second p))))))
      (cond (p
             (cond (pkg
                    `(values (intern ,(symstuff list) ,pkg)))
                   (t
                    `(make-symbol ,(symstuff list)))))
            (t
             `(values (intern ,(symstuff list))))))))

(defun symstuff (list)
  `(concatenate 'string
     ,@(for (x :in list)
            (cond ((stringp x)
                   `',x)
                  ((atom x)
                   `',(format nil "~a" x))
                  ((eq (car x) ':<)
                   `(format nil "~a" ,(second x)))
                  ((eq (car x) ':++)
                   `(format nil "~a" (incf ,(second x))))
                  (t
                   `(format nil "~a" ,x))))))
```

(Another approach would be have `symstuff` return a single call of the form `(format nil` *`format-string -forms-`*`)`, where the *forms* are derived from the *pieces*, and the *format-string* consists of interleaved ~a's and strings.)

Sometimes a macro is needed only temporarily, as a sort of syntactic scaffolding. Suppose you need to define 12 functions, but they fall into 3 stereotyped groups of 4:

```
(defun make-a-zip (y z)
  (vector 2 'zip y z))
(defun test-whether-zip (x)
  (and (vectorp x) (eq (aref x 1) 'zip)))
(defun zip-copy (x) ...)
(defun zip-deactivate (x) ...)

(defun make-a-zap (u v w)
  (vector 3 'zap u v w))
(defun test-whether-zap (x) ...)
(defun zap-copy (x) ...)
(defun zap-deactivate (x) ...)

(defun make-a-zep ()
  (vector 0 'zep))
(defun test-whether-zep (x) ...)
(defun zep-copy (x) ...)
(defun zep-deactivate (x) ...)
```

Where the omitted pieces are the same in all similarly named functions. (That is, the "…" in `zep-deactivate` is the same code as the "…" in `zip-deactivate`, and so forth.) Here, for the sake of concreteness, if not plausibility, `zip`, `zap`, and `zep` are behaving like odd little data structures. The functions could be rather large, and it would get tedious keeping them all in sync as they are debugged. An alternative would be to use a macro:

```
(defmacro odd-define (name buildargs)
  `(progn (defun ,(build-symbol make-a- (:< name))
                                ,buildargs
            (vector ,(length buildargs) ',name ,@buildargs))
          (defun ,(build-symbol test-whether- (:< name)) (x)
            (and (vectorp x) (eq (aref x 1) ',name))
```

```
          (defun ,(build-symbol (:< name) -copy) (x)
            ...)
          (defun ,(build-symbol (:< name) -deactivate) (x)
            ...))))

(odd-define zip (y z))
(odd-define zap (u v w))
(odd-define zep ())
```

If all the uses of this macro are collected in this one place, it might be clearer to make it a local macro using macrolet:

```
(macrolet ((odd-define (name buildargs)
             `(progn
                (defun ,(build-symbol make-a- (:< name))
                                        ,buildargs
                    (vector ,(length buildargs)
                            ',name
                            ,@buildargs))
                  (defun ,(build-symbol test-whether- (:< name))
                         (x)
                    (and (vectorp x) (eq (aref x 1) ',name))
                  (defun ,(build-symbol (:< name) -copy) (x)
                    ...)
                  (defun ,(build-symbol (:< name) -deactivate) (
                    ...)))))
(odd-define zip (y z))
(odd-define zap (u v w))
(odd-define zep ()))
```

Finally, macros are essential for defining “command languages.” A *command* is a function with a short name for use by users in interacting with Lisp’s read-eval-print loop. A short name is useful and possible because we want it to be easy to type and we don’t care much whether the name clashes some other command; if two command names clash, we can change one of them.

As an example, let’s define a little command language for debugging macros. (You may actually find this useful.) There are just two commands, `ex` and `fi`. They keep track of a "current form," the thing to be macro-expanded or the result of such an expansion:

1. `(ex [`*`form`*`])`: Apply `macroexpand-1` to *form* (if supplied) or the current form, and make the result the current form. Then pretty-print the current form.
2. `(fi `*`s`*` [`*`k`*`])`: Find the *k*'th subexpression of the current form whose `car` is *s*. (*k* defaults to 0.) Make that subexpression the current form and pretty-print it.

Suppose you're trying to debug a macro `hair-squared` that expands into something complex containing a subform that is itself a macro form beginning with the symbol `odd-define`. You suspect there is a bug in the subform. You might issue the following commands:

```
(ex (hair-squared ...))
(PROGN (DEFUN ...)
         (ODD-DEFINE ZIP (U V W))
         ...)

(fi odd-define)
(ODD-DEFINE ZIP (U V W))

(ex)
(PROGN (DEFUN MAKE-A-ZIP (U V W) ...)
   ...)
```

Once again, it is clear that `ex` and `fi` cannot be functions, although they could easily be made into functions if we were willing to type a quote before their arguments. But using "quote" often seems inappropriate in commands. For one thing, having to type it is a nuisance in a context where we are trying to save keystrokes, especially if the argument in question is always quoted. For another, in many cases it just seems inappropriate. If we had a command that took a symbol as one of its arguments and set it to a value, it would just be strange to write `(`*`command`*` 'x …)` instead of `(`*`command`*` x …)`, because we want to think of the command as a variant of `setq`.

Here is how `ex` and `fi` might be defined:

```
(defvar *current-form*)

(defmacro ex (&optional (form nil form-supplied))
  `(progn
     (pprint (setq *current-form*
                   (macroexpand-1
                    ,(cond (form-supplied
                            `',form)
                           (t '*current-form*)))))
     (values)))

(defmacro fi (s &optional (k 0))
  `(progn
     (pprint (setq *current-form*
                   (find-nth-occurrence ',s *current-form* ,k)))
     (values)))
```

The `ex` macro expands to a form containing a call to `macroexpand-1`, a built-in function that does one step of macro expansion to a form whose `car` is the name of a macro. (If given some other form, it returns the form unchanged.) `pprint` is a built-in function that pretty-prints its argument. Because we are using `ex` and `fi` at a read-eval-print loop, any value returned by their expansions will be printed. Here the expansion is executed for side effect, so we arrange to return no values at all by having the expansion return `(values)`.

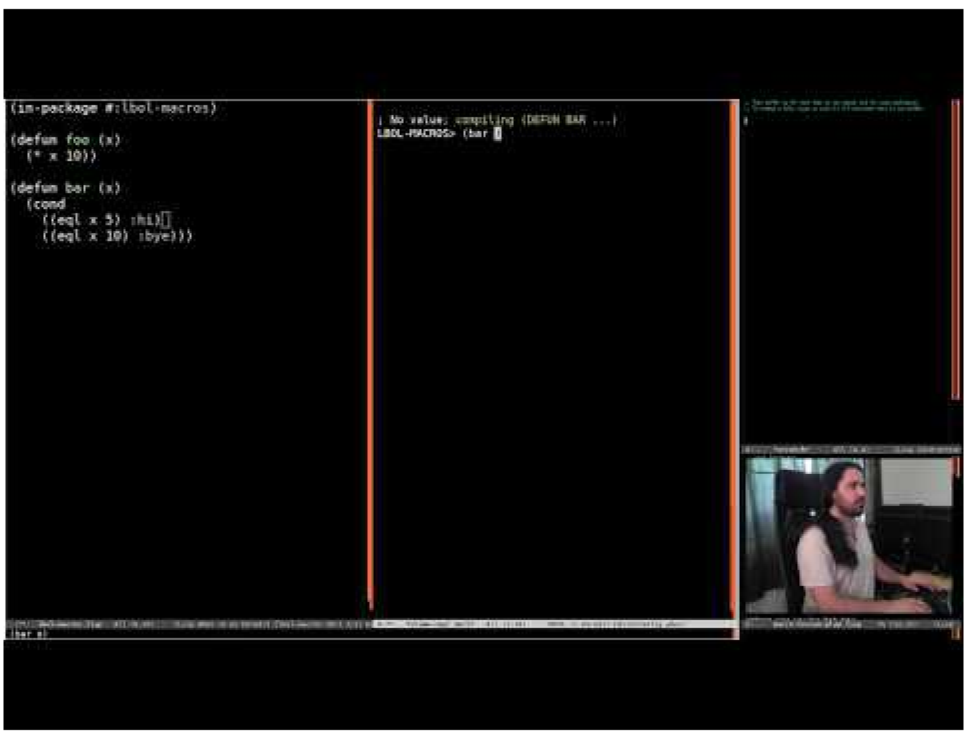
In some Lisp implementations, read-eval-print loops routinely print results using `pprint`. In those implementations we could simplify `ex` and `fi` by having them print nothing, but just return the value of `*current-form*`, which the read-eval-print loop will then print prettily. Use your judgment.

I leave the definition of `find-nth-occurrence` as an exercise. You might also want to define a command that just sets and prints the current form: `(cf` *`e`*`)`.

One caution: In general, command languages will consist of a mixture of macros and functions, with convenience for their definer (and usually sole user) being the main consideration. If a command seems to "want" to evaluate some of its arguments sometimes, you have to decide whether to define two (or more) versions of it, or just one, a function whose arguments must be quoted to prevent their being evaluated. For the `cf` command mentioned in the previous paragraph, some users might prefer `cf` to be a function, some a macro.

See also

- [A gentle introduction to Compile-Time Computing — Part 1]()
- [Safely dealing with scientific units of variables at compile time (a gentle introduction to Compile-Time Computing — part 3)]()
- The following video, from the series [“Little bits of Lisp”]() by [cbaggers](), is a two hours long talk on macros, showing simple to advanced concepts such as compiler macros: [https://www.youtube.com/watch?v=ygKXeLKhiTI]() It also shows how to manipulate macros (and their expansion) in Emacs.



- the article “Reader macros in Common Lisp”: https://lisper.in/reader-macros

Fundamentals of CLOS

CLOS is the "Common Lisp Object System", arguably one of the most powerful object systems available in any language.

Some of its features include:

- it is **dynamic**, making it a joy to work with in a Lisp REPL. For example, changing a class definition will update the existing objects, given certain rules which we have control upon.
- it supports **multiple dispatch** and **multiple inheritance**,
- it is different from most object systems in that class and method definitions are not tied together,
- it has excellent **introspection** capabilities,
- it is provided by a **meta-object protocol**, which provides a standard interface to the CLOS, and can be used to create new object systems.

The functionality belonging to this name was added to the Common Lisp language between the publication of Steele's first edition of "Common Lisp, the Language" in 1984 and the formalization of the language as an ANSI standard ten years later.

This page aims to give a good understanding of how to use CLOS, but only a brief introduction to the MOP.

To learn the subjects in depth, you will need two books:

- Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS, by Sonya Keene,
- the Art of the Metaobject Protocol, by Gregor Kiczales, Jim des Rivières et al.

But see also

- the introduction in Practical Common Lisp (online), by Peter Seibel.
- Common Lisp, the Language

- and for reference, the complete CLOS-MOP specifications.

Classes and instances

Diving in

Let’s dive in with an example showing class definition, creation of objects, slot access, methods specialized for a given class, and inheritance.

```
(defclass person ()
  ((name
    :initarg :name
    :accessor name)
   (lisper
    :initform nil
    :accessor lisper)))

;; => #<STANDARD-CLASS PERSON>

(defvar p1 (make-instance 'person :name "me" ))
;;                                  ^^^^ initarg
;; => #<PERSON {1006234593}>

(name p1)
;;^^^ accessor
;; => "me"

(lisper p1)
;; => nil
;;    ^^ initform (slot unbound by default)

(setf (lisper p1) t)


(defclass child (person)
  ())

(defclass child (person)
  ((can-walk-p
```

```
    :accessor can-walk-p
    :initform t)))
;; #<STANDARD-CLASS CHILD>

(can-walk-p (make-instance 'child))
;; T
```

Defining classes (defclass)

The macro used for defining new data types in CLOS is `defclass`.

We used it like this:

```
(defclass person ()
  ((name
    :initarg :name
    :accessor name)
   (lisper
    :initform nil
    :accessor lisper)))
```

This gives us a CLOS type (or class) called `person` and two slots, named `name` and `lisper`.

```
(class-of p1)
#<STANDARD-CLASS PERSON>

(type-of p1)
PERSON
```

The general form of `defclass` is:

```
(defclass <class-name> (list of super classes)
  ((slot-1
     :slot-option slot-argument)
   (slot-2, etc))
  (:optional-class-option
   :another-optional-class-option))
```

So, our `person` class doesn't explicitly inherit from another class (it gets the empty parentheses `()`). However it still inherits by default from the class `t` and from `standard-object`. See below under “inheritance”.

We could write a minimal class definition without slot options like this:

```
(defclass point ()
  (x y z))
```

or even without slot specifiers: `(defclass point () ())`.

Creating objects (make-instance)

We create instances of a class with `make-instance`:

```
(defvar p1 (make-instance 'person :name "me" ))
```

It is generally good practice to define a constructor:

```
(defun make-person (name &key lisper)
  (make-instance 'person :name name :lisper lisper))
```

This has the direct advantage that you can control the required arguments. You should now export the constructor from your package and not the class itself.

Slots

A function that always works (slot-value)

The function to access any slot anytime is `(slot-value <object> <slot-name>)`.

Given our `point` class above, which didn’t define any slot accessors:

```
(defvar pt (make-instance 'point))

(inspect pt)
The object is a STANDARD-OBJECT of type POINT.
0. X: "unbound"
```

```
1. Y: "unbound"
2. Z: "unbound"
```

We got an object of type `POINT`, but **slots are unbound by default**: trying to access them will raise an `UNBOUND-SLOT` condition:

```
(slot-value pt 'x) ;; => condition: the slot is unbound
```

`slot-value` is `setf`-able:

```
(setf (slot-value pt 'x) 1)
(slot-value pt 'x) ;; => 1
```

Initial and default values (initarg, initform)

- `:initarg :foo` is the keyword we can pass to `make-instance` to give a value to this slot:

```
(make-instance 'person :name "me")
```

(again: slots are unbound by default)

- `:initform <val>` is the *default value* in case we didn't specify an initarg. This form is evaluated each time it's needed, in the lexical environment of the `defclass`.

Sometimes we see the following trick to clearly require a slot:

```
(defclass foo ()
    ((a
      :initarg :a
      :initform (error "you didn't supply an initial value for s
;; #<STANDARD-CLASS FOO>

(make-instance 'foo) ;; => enters the debugger.
```

Getters and setters (accessor, reader, writer)

- `:accessor foo`: an accessor is both a **getter** and a **setter**. Its argument is a name that will become a **generic function**.

```
(name p1) ;; => "me"

(type-of #'name)
STANDARD-GENERIC-FUNCTION
```

- `:reader` and `:writer` do what you expect. Only the `:writer` is `setf`-able.

If you don't specify any of these, you can still use `slot-value`.

You can give a slot more than one `:accessor`, `:reader` or `:initarg`.

We introduce two macros to make the access to slots shorter in some situations:

1- `with-slots` allows to abbreviate several calls to slot-value. The first argument is a list of slot names. The second argument evaluates to a CLOS instance. This is followed by optional declarations and an implicit `progn`. Lexically during the evaluation of the body, an access to any of these names as a variable is equivalent to accessing the corresponding slot of the instance with `slot-value`.

```
(with-slots (name lisper)
    c1
  (format t "got ~a, ~a~&" name lisper))
```

or

```
(with-slots ((n name)
             (l lisper))
    c1
  (format t "got ~a, ~a~&" n l))
```

2- `with-accessors` is equivalent, but instead of a list of slots it takes a list of accessor functions. Any reference to the variable inside the macro is equivalent to a call to the accessor function.

```
(with-accessors ((name        name)
                  ^^variable  ^^accessor
                 (lisper lisper))
            p1
          (format t "name: ~a, lisper: ~a" name lisper))
```

Class VS instance slots

`:allocation` specifies whether this slot is *local* or *shared*.

- a slot is *local* by default, that means it can be different for each instance of the class. In that case `:allocation` equals `:instance`.
- a *shared* slot will always be equal for all instances of the class. We set it with `:allocation :class`.

In the following example, note how changing the value of the class slot `species` of `p2` affects all instances of the class (whether or not those instances exist yet).

```
(defclass person ()
  ((name :initarg :name :accessor name)
   (species
      :initform 'homo-sapiens
      :accessor species

      :allocation :class)))

;; Note that the slot "lisper" was removed in existing instances
(inspect p1)
;; The object is a STANDARD-OBJECT of type PERSON.
;; 0. NAME: "me"
;; 1. SPECIES: HOMO-SAPIENS
;; > q

(defvar p2 (make-instance 'person))

(species p1)
(species p2)
;; HOMO-SAPIENS
```

```
(setf (species p2) 'homo-numericus)
;; HOMO-NUMERICUS

(species p1)
;; HOMO-NUMERICUS

(species (make-instance 'person))
;; HOMO-NUMERICUS

(let ((temp (make-instance 'person)))
    (setf (species temp) 'homo-lisper))
;; HOMO-LISPER
(species (make-instance 'person))
;; HOMO-LISPER
```

Slot documentation

Each slot accepts one `:documentation` option. To obtain its documentation via `documentation`, you need to obtain the slot object. This can be done compatibly using a library such as closer-mop. For instance:

```
(closer-mop:class-direct-slots (find-class 'my-class))
;; => list of slots (objects)
(find 'my-slot * :key #'closer-mop:slot-definition-name)
;; => find desired slot by name
(documentation * t) ; obtain its documentation
```

Note however that generally it may be better to document slot accessors instead, as a popular viewpoint is that slots are implementation details and not part of the public interface.

Slot type

The `:type` slot option may not do the job you expect it does. If you are new to the CLOS, we suggest you skip this section and use your own constructors to manually check slot types.

Indeed, whether slot types are being checked or not is undefined. See the Hyperspec.

Few implementations will do it. Clozure CL does it, SBCL does it since its version 1.5.9 (November, 2019) or when safety is high (`(declaim (optimise safety))`).

To do it otherwise, see this Stack-Overflow answer, and see also quid-pro-quo, a contract programming library.

find-class, class-name, class-of

```
(find-class 'point)
;; #<STANDARD-CLASS POINT 275B78DC>

(class-name (find-class 'point))
;; POINT

(class-of my-point)
;; #<STANDARD-CLASS POINT 275B78DC>

(typep my-point (class-of my-point))
;; T
```

CLOS classes are also instances of a CLOS class, and we can find out what that class is, as in the example below:

```
(class-of (class-of my-point))
;; #<STANDARD-CLASS STANDARD-CLASS 20306534>
```

Note: this is your first introduction to the MOP. You don't need that to get started !

The object `my-point` is an instance of the class named `point`, and the class named `point` is itself an instance of the class named `standard-class`. We say that the class named `standard-class` is the *metaclass* (i.e. the class of the class) of `my-point`. We can make good uses of metaclasses, as we'll see later.

Subclasses and inheritance

As illustrated above, `child` is a subclass of `person`.

All objects inherit from the class `standard-object` and `t`.

Every child instance is also an instance of `person`.

```
(type-of c1)
;; CHILD

(subtypep (type-of c1) 'person)
;; T

(ql:quickload "closer-mop")
;; ...

(closer-mop:subclassp (class-of c1) 'person)
;; T
```

The closer-mop library is *the* portable way to do CLOS/MOP operations.

A subclass inherits all of its parents' slots, and it can override any of their slot options. Common Lisp makes this process dynamic, great for REPL session, and we can even control parts of it (like, do something when a given slot is removed/updated/added, etc).

The **class precedence list** of a `child` is thus:

child <- person <– standard-object <- t

Which we can get with:

```
(closer-mop:class-precedence-list (class-of c1))
;; (#<standard-class child>
;;  #<standard-class person>
;;  #<standard-class standard-object>
;;  #<sb-pcl::slot-class sb-pcl::slot-object>
;;  #<sb-pcl:system-class t>)
```

However, the **direct superclass** of a `child` is only:

```
(closer-mop:class-direct-superclasses (class-of c1))
;; (#<standard-class person>)
```

We can further inspect our classes with `class-direct-[subclasses, slots, default-initargs]` and many more functions.

How slots are combined follows some rules:

- `:accessor` and `:reader` are combined by the **union** of accessors and readers from all the inherited slots.
- `:initarg`: the **union** of initialization arguments from all the inherited slots.
- `:initform`: we get **the most specific** default initial value form, i.e. the first `:initform` for that slot in the precedence list.
- `:allocation` is not inherited. It is controlled solely by the class being defined and defaults to `:instance`.

Last but not least, be warned that inheritance is fairly easy to misuse, and multiple inheritance is multiply so, so please take a little care. Ask yourself whether `foo` really wants to inherit from `bar`, or whether instances of `foo` want a slot containing a `bar`. A good general guide is that if `foo` and `bar` are "same sort of thing" then it's correct to mix them together by inheritance, but if they're really separate concepts then you should use slots to keep them apart.

Multiple inheritance

CLOS supports multiple inheritance.

```
(defclass baby (child person)
  ())
```

The first class on the list of parent classes is the most specific one, `child`'s slots will take precedence over the `person`'s. Note that both `child` and `person` have to be defined prior to defining `baby` in this example.

Redefining and changing a class

This section briefly covers two topics:

- redefinition of an existing class, which you might already have done by following our code snippets, and what we do naturally during development, and
- changing an instance of one class into an instance of another, a powerful feature of CLOS that you'll probably won't use very often.

We'll gloss over the details. Suffice it to say that everything's configurable by implementing methods exposed by the MOP.

To redefine a class, simply evaluate a new `defclass` form. This then takes the place of the old definition, the existing class object is updated, and **all instances of the class** (and, recursively, its subclasses) **are lazily updated to reflect the new definition**. You don't have to recompile anything other than the new `defclass`, nor to invalidate any of your objects. Think about it for a second: this is awesome !

For example, with our `person` class:

```
(defclass person ()
  ((name
    :initarg :name
    :accessor name)
   (lisper
    :initform nil
    :accessor lisper)))

(setf p1 (make-instance 'person :name "me" ))
```

Changing, adding, removing slots,…

```
(lisper p1)
;; NIL

(defclass person ()
```

```
(defclass person ()
  ((name
    :initarg :name
    :accessor name)
   (lisper
    :initform t        ;; <-- from nil to t

    :accessor lisper)))

(lisper p1)
;; NIL (of course!)

(lisper (make-instance 'person :name "You"))
;; T

(defclass person ()
  ((name
    :initarg :name
    :accessor name)
   (lisper
    :initform nil
    :accessor lisper)
   (age                ;; <-- new slot
    :initarg :arg
    :initform 18       ;; <-- default value
    :accessor age)))

(age p1)
;; => 18. Correct. This is the default initform for this new slo

(slot-value p1 'bwarf)
;; => "the slot bwarf is missing from the object #<person…>"

(setf (age p1) 30)
(age p1) ;; => 30

(defclass person ()
  ((name
    :initarg :name
    :accessor name)))

(slot-value p1 'lisper) ;; => slot lisper is missing.
(lisper p1) ;; => there is no applicable method for the generic
```

```
(lisper p1) ;; => there is no applicable method for the generic
```

To change the class of an instance, use `change-class`:

```
(change-class p1 'child)
;; we can also set slots of the new class:
(change-class p1 'child :can-walk-p nil)

(class-of p1)
;; #<STANDARD-CLASS CHILD>

(can-walk-p p1)
;; T
```

In the above example, I became a `child`, and I inherited the `can-walk-p` slot, which is true by default.

Pretty printing

Every time we printed an object so far we got an output like

#<PERSON {1006234593}>

which doesn't say much.

What if we want to show more information ? Something like

#

Pretty printing is done by specializing the generic `print-object` method for this class:

```
(defmethod print-object ((obj person) stream)
      (print-unreadable-object (obj stream :type t)
        (with-accessors ((name name)
                         (lisper lisper))
            obj
          (format stream "~a, lisper: ~a" name lisper))))
```

It gives:

```
p1
;; #<PERSON me, lisper: T>
```

`print-unreadable-object` prints the `#<...>`, that says to the reader that this object can not be read back in. Its `:type t` argument asks to print the object-type prefix, that is, `PERSON`. Without it, we get `#<me, lisper: T>`.

We used the `with-accessors` macro, but of course for simple cases this is enough:

```
(defmethod print-object ((obj person) stream)
  (print-unreadable-object (obj stream :type t)
    (format stream "~a, lisper: ~a" (name obj) (lisper obj))))
```

Caution: trying to access a slot that is not bound by default will lead to an error. Use `slot-boundp`.

For reference, the following reproduces the default behaviour:

```
(defmethod print-object ((obj person) stream)
  (print-unreadable-object (obj stream :type t :identity t)))
```

Here, `:identity` to `t` prints the `{1006234593}` address.

Classes of traditional lisp types

Where we approach that we don't need CLOS objects to use CLOS.

Generously, the functions introduced in the last section also work on lisp objects which are <u>not</u> CLOS instances:

```
(find-class 'symbol)
;; #<BUILT-IN-CLASS SYMBOL>
(class-name *)
;; SYMBOL
(eq ** (class-of 'symbol))
;; T
(class-of ***)
;; #<STANDARD-CLASS BUILT-IN-CLASS>
```

We see here that symbols are instances of the system class `symbol`. This is one of 75 cases in which the language requires a class to exist with the same name as the corresponding lisp type. Many of these cases are concerned with CLOS itself (for example, the correspondence between the type `standard-class` and the CLOS class of that name) or with the condition system (which might or might not be built using CLOS classes in any given implementation). However, 33 correspondences remain relating to "traditional" lisp types:

array	hash-table	readtable
bit-vector	integer	real
broadcast-stream	list	sequence
character	logical-pathname	stream
complex	null	string
concatenated-stream	number	string-stream
cons	package	symbol
echo-stream	pathname	synonym-stream
file-stream	random-state	t
float	ratio	two-way-stream
function	rational	vector

Note that not all "traditional" lisp types are included in this list. (Consider: `atom`, `fixnum`, `short-float`, and any type not denoted by a symbol.)

The presence of `t` is interesting. Just as every lisp object is of type `t`, every lisp object is also a member of the class named `t`. This is a simple example of membership of more then one class at a time, and it brings into question the issue of *inheritance*, which we will consider in some detail later.

```
(find-class t)
;; #<BUILT-IN-CLASS T 20305AEC>
```

In addition to classes corresponding to lisp types, there is also a CLOS class for every structure type you define:

```lisp
(defstruct foo)
FOO

(class-of (make-foo))
;; #<STRUCTURE-CLASS FOO 21DE8714>
```

The metaclass of a `structure-object` is the class `structure-class`. It is implementation-dependent whether the metaclass of a "traditional" lisp object is `standard-class`, `structure-class`, or `built-in-class`. Restrictions:

`built-in-class`: May not use `make-instance`, may not use `slot-value`, may not use `defclass` to modify, may not create subclasses.

`structure-class`: May not use `make-instance`, might work with `slot-value` (implementation-dependent). Use `defstruct` to subclass application structure types. Consequences of modifying an existing `structure-class` are undefined: full recompilation may be necessary.

`standard-class`: None of these restrictions.

Introspection

We already saw some introspection functions.

Your best option is to discover the closer-mop library and to keep the CLOS & MOP specifications at hand.

More functions:

```lisp
closer-mop:class-default-initargs
closer-mop:class-direct-default-initargs
closer-mop:class-direct-slots
closer-mop:class-direct-subclasses
closer-mop:class-direct-superclasses
closer-mop:class-precedence-list
closer-mop:class-slots
```

```
closer-mop:classp
closer-mop:extract-lambda-list
closer-mop:extract-specializer-names
closer-mop:generic-function-argument-precedence-order
closer-mop:generic-function-declarations
closer-mop:generic-function-lambda-list
closer-mop:generic-function-method-class
closer-mop:generic-function-method-combination
closer-mop:generic-function-methods
closer-mop:generic-function-name
closer-mop:method-combination
closer-mop:method-function
closer-mop:method-generic-function
closer-mop:method-lambda-list
closer-mop:method-specializers
closer-mop:slot-definition
closer-mop:slot-definition-allocation
closer-mop:slot-definition-initargs
closer-mop:slot-definition-initform
closer-mop:slot-definition-initfunction
closer-mop:slot-definition-location
closer-mop:slot-definition-name
closer-mop:slot-definition-readers
closer-mop:slot-definition-type
closer-mop:slot-definition-writers
closer-mop:specializer-direct-generic-functions
closer-mop:specializer-direct-methods
closer-mop:standard-accessor-method
```

See also

defclass/std: write shorter classes

The library defclass/std provides a macro to write shorter `defclass` forms.

By default, it adds an accessor, an initarg and an initform to `nil` to your slots definition:

This:

```
(defclass/std example ()
  ((slot1 slot2 slot3)))
```

expands to:

```lisp
(defclass example ()
  ((slot1
    :accessor slot1
    :initarg :slot1
    :initform nil)
   (slot2
     :accessor slot2
     :initarg :slot2
     :initform nil)
   (slot3
     :accessor slot3
     :initarg :slot3
     :initform nil)))
```

It does much more and it is very flexible, however it is seldom used by the Common Lisp community: use at your own risk©.

Methods

Diving in

Recalling our `person` and `child` classes from the beginning:

```lisp
(defclass person ()
  ((name
    :initarg :name
    :accessor name)))
;; => #<STANDARD-CLASS PERSON>

(defclass child (person)
  ())
;; #<STANDARD-CLASS CHILD>

(setf p1 (make-instance 'person :name "me"))
(setf c1 (make-instance 'child :name "Alice"))
```

Below we create methods, we specialize them, we use method combination (before, after, around), and qualifiers.

```
(defmethod greet (obj)
  (format t "Are you a person ? You are a ~a.~&" (type-of obj)))
;; style-warning: Implicitly creating new generic function commo
;; #<STANDARD-METHOD GREET (t) {1008EE4603}>

(greet :anything)
;; Are you a person ? You are a KEYWORD.
;; NIL
(greet p1)
;; Are you a person ? You are a PERSON.

(defgeneric greet (obj)
  (:documentation "say hello"))
;; STYLE-WARNING: redefining COMMON-LISP-USER::GREET in DEFGENER
;; #<STANDARD-GENERIC-FUNCTION GREET (2)>

(defmethod greet ((obj person))
  (format t "Hello ~a !~&" (name obj)))
;; #<STANDARD-METHOD GREET (PERSON) {1007C26743}>

(greet p1) ;; => "Hello me !"

(greet c1) ;; => "Hello Alice !"

(defmethod greet ((obj child))
  (format t "ur so cute~&"))
;; #<STANDARD-METHOD GREET (CHILD) {1008F3C1C3}>

(greet p1) ;; => "Hello me !"
(greet c1) ;; => "ur so cute"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Method combination: before, after, around.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod greet :before ((obj person))
  (format t "-- before person~&"))
#<STANDARD-METHOD GREET :BEFORE (PERSON) {100C94A013}>

(greet p1)
;; -- before person
;; Hello me
```

```lisp
(defmethod greet :before ((obj child))
  (format t "-- before child~&"))
;; #<STANDARD-METHOD GREET :BEFORE (CHILD) {100AD32A43}>
(greet c1)
;; -- before child
;; -- before person
;; ur so cute

(defmethod greet :after ((obj person))
  (format t "-- after person~&"))
;; #<STANDARD-METHOD GREET :AFTER (PERSON) {100CA2E1A3}>
(greet p1)
;; -- before person
;; Hello me
;; -- after person

(defmethod greet :after ((obj child))
  (format t "-- after child~&"))
;; #<STANDARD-METHOD GREET :AFTER (CHILD) {10075B71F3}>
(greet c1)

;; -- before child
;; -- before person
;; ur so cute
;; -- after person
;; -- after child

(defmethod greet :around ((obj child))
  (format t "Hello my dear~&"))
;; #<STANDARD-METHOD GREET :AROUND (CHILD) {10076658E3}>
(greet c1) ;; Hello my dear



;; call-next-method

(defmethod greet :around ((obj child))
  (format t "Hello my dear~&")
  (when (next-method-p)
    (call-next-method)))
;; #<standard-method greet :around (child) {100AF76863}>
```

```
(greet c1)
;; Hello my dear
;; -- before child
;; -- before person
;; ur so cute
;; -- after person
;; -- after child

;;;;;;;;;;;;;;;;;
;; Adding in &key
;;;;;;;;;;;;;;;;;

;; In order to add "&key" to our generic method, we need to remo
(fmakunbound 'greet)  ;; with Slime: C-c C-u (slime-undefine-fur
(defmethod greet ((obj person) &key talkative)
  (format t "Hello ~a~&" (name obj))
  (when talkative
    (format t "blah")))

(defgeneric greet (obj &key &allow-other-keys)
  (:documentation "say hi"))


(defmethod greet (obj &key &allow-other-keys)
  (format t "Are you a person ? You are a ~a.~&" (type-of obj)))

(defmethod greet ((obj person) &key talkative &allow-other-keys)
  (format t "Hello ~a !~&" (name obj))
  (when talkative
    (format t "blah")))

(greet p1 :talkative t) ;; ok
(greet p1 :foo t) ;; still ok



;;;;;;;;;;;;;;;;;;;;;;;

(defgeneric greet (obj)
  (:documentation "say hello")
  (:method (obj)
    (format t "Are you a person ? You are a ~a~&." (type-of obj)
  (:method ((obj person))
```

```
  (:method ((obj person))
    (format t "Hello ~a !~&" (name obj)))
  (:method ((obj child))
    (format t "ur so cute~&")))

;;;;;;;;;;;;;;;;
;;; Specializers
;;;;;;;;;;;;;;;;

(defgeneric feed (obj meal-type)
  (:method (obj meal-type)
    (declare (ignorable meal-type))
    (format t "eating~&")))

(defmethod feed (obj (meal-type (eql :dessert)))
    (declare (ignorable meal-type))
    (format t "mmh, dessert !~&"))

(feed c1 :dessert)
;; mmh, dessert !


(defmethod feed ((obj child) (meal-type (eql :soup)))
    (declare (ignorable meal-type))
    (format t "bwark~&"))

(feed p1 :soup)
;; eating
(feed c1 :soup)
;; bwark
```

Generic functions (defgeneric, defmethod)

A `generic function` is a lisp function which is associated with a set of methods and dispatches them when it's invoked. All the methods with the same function name belong to the same generic function.

The `defmethod` form is similar to a `defun`. It associates a body of code with a function name, but that body may only be executed if the types of the arguments match the pattern declared by the lambda list.

They can have optional, keyword and `&rest` arguments.

The `defgeneric` form defines the generic function. If we write a `defmethod` without a corresponding `defgeneric`, a generic function is automatically created (see examples).

It is generally a good idea to write the `defgenerics`. We can add a default implementation and even some documentation.

```
(defgeneric greet (obj)
  (:documentation "says hi")
  (:method (obj)
    (format t "Hi")))
```

The required parameters in the method's lambda list may take one of the following three forms:

1- a simple variable:

```
(defmethod greet (foo)
  ...)
```

This method can take any argument, it is always applicable.

The variable `foo` is bound to the corresponding argument value, as usual.

2- a variable and a **specializer**, as in:

```
(defmethod greet ((foo person))
  ...)
```

In this case, the variable `foo` is bound to the corresponding argument only if that argument is of specializer class `person` *or a subclass*, like `child` (indeed, a "child" is also a "person").

If any argument fails to match its specializer then the method is not *applicable* and it cannot be executed with those arguments.We'll get an error message like "there is no applicable method for the generic function xxx when called with arguments yyy".

Only required parameters can be specialized. We can't specialize on optional `&key` arguments.

3- a variable and an **eql specializer**

```
(defmethod feed ((obj child) (meal-type (eql :soup)))
    (declare (ignorable meal-type))
    (format t "bwark~&"))

(feed c1 :soup)
;; "bwark"
```

In place of a simple symbol (`:soup`), the eql specializer can be any lisp form. It is evaluated at the same time of the defmethod.

You can define any number of methods with the same function name but with different specializers, as long as the form of the lambda list is *congruent* with the shape of the generic function. The system chooses the most *specific* applicable method and executes its body. The most specific method is the one whose specializers are nearest to the head of the `class-precedence-list` of the argument (classes on the left of the lambda list are more specific). A method with specializers is more specific to one without any.

Notes:

- It is an error to define a method with the same function name as an ordinary function. If you really want to do that, use the shadowing mechanism.

- To add or remove `keys` or `rest` arguments to an existing generic method's lambda list, you will need to delete its declaration with `fmakunbound` (or `C-c C-u` (slime-undefine-function) with the cursor on the function in Slime) and start again. Otherwise, you'll see:

```
attempt to add the method
  #<STANDARD-METHOD NIL (#<STANDARD-CLASS CHILD>) {1009504233}>
to the generic function
  #<STANDARD-GENERIC-FUNCTION GREET (2)>;
but the method and generic function differ in whether they
```

`accept`
`&REST or &KEY arguments.`

- Methods can be redefined (exactly as for ordinary functions).
- The order in which methods are defined is irrelevant, although any classes on which they specialize must already exist.
- An unspecialized argument is more or less equivalent to being specialized on the class `t`. The only difference is that all specialized arguments are implicitly taken to be “referred to” (in the sense of `declare ignore`.)
- Each `defmethod` form generates (and returns) a CLOS instance, of class `standard-method`.
- An `eql` specializer won’t work as is with strings. Indeed, strings need `equal` or `equalp` to be compared. But, we can assign our string to a variable and use the variable both in the `eql` specializer and for the function call.
- All the methods with the same function name belong to the same generic function.
- All slot accessors and readers defined by `defclass` are methods. They can override or be overridden by other methods on the same generic function.

See more about defmethod on the CLHS.

Multimethods

Multimethods explicitly specialize more than one of the generic function’s required parameters.

They don’t belong to a particular class. Meaning, we don’t have to decide on the class that would be best to host this method, as we might have to in other languages.

```lisp
(defgeneric hug (a b)
   (:documentation "Hug between two persons."))
;; #<STANDARD-GENERIC-FUNCTION HUG (0)>

(defmethod hug ((a person) (b person))
  :person-person-hug)

(defmethod hug ((a person) (b child))
  :person-child-hug)
```

Read more on [Practical Common Lisp](#).

Controlling setters (setf-ing methods)

In Lisp, we can define `setf` counterparts of functions or methods. We might want this to have more control on how to update an object.

```lisp
(defmethod (setf name) (new-val (obj person))
  (if (equalp new-val "james bond")
    (format t "Dude that's not possible.~&")
    (setf (slot-value obj 'name) new-val)))

(setf (name p1) "james bond") ;; -> no rename
```

If you know Python, this behaviour is provided by the `@property` decorator.

Dispatch mechanism and next methods

When a generic function is invoked, the application cannot directly invoke a method. The dispatch mechanism proceeds as follows:

1. compute the list of applicable methods
2. if no method is applicable then signal an error
3. sort the applicable methods in order of specificity
4. invoke the most specific method.

Our `greet` generic function has three applicable methods:

```
(closer-mop:generic-function-methods #'greet)
(#<STANDARD-METHOD GREET (CHILD) {10098406A3}>
 #<STANDARD-METHOD GREET (PERSON) {1009008EC3}>
 #<STANDARD-METHOD GREET (T) {1008E6EBB3}>)
```

During the execution of a method, the remaining applicable methods are still accessible, via the *local function* `call-next-method`. This function has lexical scope within the body of a method but indefinite extent. It invokes the next most specific method, and returns whatever value that method returned. It can be called with either:

- no arguments, in which case the *next method* will receive exactly the same arguments as this method did, or
- explicit arguments, in which case it is required that the sorted set of methods applicable to the new arguments must be the same as that computed when the generic function was first called.

For example:

```
(defmethod greet ((obj child))
  (format t "ur so cute~&")
  (when (next-method-p)
    (call-next-method)))
;; STYLE-WARNING: REDEFINING GREET (#<STANDARD-CLASS CHILD>) in
;; #<STANDARD-METHOD GREET (child) {1003D3DB43}>

(greet c1)
;; ur so cute
;; Hello Alice !
```

Calling `call-next-method` when there is no next method signals an error. You can find out whether a next method exists by calling the local function `next-method-p` (which also has has lexical scope and indefinite extent).

Note finally that the body of every method establishes a block with the same name as the method's generic function. If you `return-from` that name you are exiting the current method, not the call to the enclosing generic function.

Method qualifiers (before, after, around)

In our "Diving in" examples, we saw some use of the `:before`, `:after` and `:around` *qualifiers*:

- `(defmethod foo :before (obj) (...))`
- `(defmethod foo :after (obj) (...))`
- `(defmethod foo :around (obj) (...))`

By default, in the *standard method combination* framework provided by CLOS, we can only use one of those three qualifiers, and the flow of control is as follows:

- a **before-method** is called, well, before the applicable primary method. If they are many before-methods, **all** are called. The most specific before-method is called first (child before person).
- the most specific applicable **primary method** (a method without qualifiers) is called (only one).
- all applicable **after-methods** are called. The most specific one is called *last* (after-method of person, then after-method of child).

The generic function returns the value of the primary method. Any values of the before or after methods are ignored. They are used for their side effects.

And then we have **around-methods**. They are wrappers around the core mechanism we just described. They can be useful to catch return values or to set up an environment around the primary method (set up a catch, a lock, timing an execution,…).

If the dispatch mechanism finds an around-method, it calls it and returns its result. If the around-method has a `call-next-method`, it calls the next most applicable around-method. It is only when we reach the primary method that we start calling the before and after-methods.

Thus, the full dispatch mechanism for generic functions is as follows:

1. compute the applicable methods, and partition them into separate lists according to their qualifier;
2. if there is no applicable primary method then signal an error;
3. sort each of the lists into order of specificity;
4. execute the most specific `:around` method and return whatever that returns;
5. if an `:around` method invokes `call-next-method`, execute the next most specific `:around` method;

6. if there were no `:around` methods in the first place, or if an `:around` method invokes `call-next-method` but there are no further `:around` methods to call, then proceed as follows:

    a. run all the `:before` methods, in order, ignoring any return values and not permitting calls to `call-next-method` or `next-method-p`;

    b. execute the most specific primary method and return whatever that returns;

    c. if a primary method invokes `call-next-method`, execute the next most specific primary method;

    d. if a primary method invokes `call-next-method` but there are no further primary methods to call then signal an error;

    e. after the primary method(s) have completed, run all the `:after` methods, in **reverse** order, ignoring any return values and not permitting calls to `call-next-method` or `next-method-p`.

Think of it as an onion, with all the `:around` methods in the outermost layer, `:before` and `:after` methods in the middle layer, and primary methods on the inside.

Other method combinations

The default method combination type we just saw is named `standard`, but other method combination types are available, and no need to say that you can define your own.

The built-in types are:

progn + list nconc and max or append min

You notice that these types are named after a lisp operator. Indeed, what they do is they define a framework that combines the applicable primary methods inside a call to the lisp operator of that name. For example, using the `progn` combination type is equivalent to calling **all** the primary methods one after the other:

```
(progn
  (method-1 args)
  (method-2 args)
  (method-3 args))
```

Here, unlike the standard mechanism, all the primary methods applicable for a given object are called, the most specific first.

To change the combination type, we set the `:method-combination` option of `defgeneric` and we use it as the methods' qualifier:

```
(defgeneric foo (obj)
  (:method-combination progn))

(defmethod foo progn ((obj obj))
   (...))
```

An example with **progn**:

```
(defgeneric dishes (obj)
   (:method-combination progn)
   (:method progn (obj)
     (format t "- clean and dry.~&"))
   (:method progn ((obj person))
     (format t "- bring a person's dishes~&"))
   (:method progn ((obj child))
     (format t "- bring the baby dishes~&")))
;; #<STANDARD-GENERIC-FUNCTION DISHES (3)>

(dishes c1)
```

```
;; - bring the baby dishes
;; - bring a person's dishes
;; - clean and dry.

(greet c1)
;; ur so cute  --> only the most applicable method was called.
```

Similarly, using the `list` type is equivalent to returning the list of the values of the methods.

```
(list
  (method-1 args)
  (method-2 args)
  (method-3 args))

(defgeneric tidy (obj)
  (:method-combination list)
  (:method list (obj)
    :foo)
  (:method list ((obj person))
    :books)
  (:method list ((obj child))
    :toys))
;; #<STANDARD-GENERIC-FUNCTION TIDY (3)>

(tidy c1)
;; (:toys :books :foo)
```

Around methods are accepted:

```
(defmethod tidy :around (obj)
   (let ((res (call-next-method)))
     (format t "I'm going to clean up ~a~&" res)
     (when (> (length res)
              1)
       (format t "that's too much !~&"))))

(tidy c1)
;; I'm going to clean up (toys book foo)
;; that's too much !
```

Note that these operators don't support `before`, `after` and `around` methods (indeed, there is no room for them anymore). They do support around methods, where `call-next-method` is allowed, but they don't support calling `call-next-method` in the primary methods (it would indeed be redundant since all primary methods are called, or clunky to *not* call one).

CLOS allows us to define a new operator as a method combination type, be it a lisp function, macro or special form. We'll let you refer to the books if you feel the need.

Debugging: tracing method combination

It is possible to [trace](#) the method combination, but this is implementation dependent.

In SBCL, we can use `(trace foo :methods t)`. See [this post by an SBCL core developer](#).

For example, given a generic:

```
(defgeneric foo (x)
  (:method (x) 3))
(defmethod foo :around ((x fixnum))
  (1+ (call-next-method)))
(defmethod foo ((x integer))
  (* 2 (call-next-method)))
(defmethod foo ((x float))
  (* 3 (call-next-method)))
(defmethod foo :before ((x single-float))
  'single)
(defmethod foo :after ((x double-float))
 'double)
```

Let's trace it:

```
(trace foo :methods t)

(foo 2.0d0)
  0: (FOO 2.0d0)
    1: ((SB-PCL::COMBINED-METHOD FOO) 2.0d0)
```

```
      2: ((METHOD FOO (FLOAT)) 2.0d0)
        3: ((METHOD FOO (T)) 2.0d0)
        3: (METHOD FOO (T)) returned 3
      2: (METHOD FOO (FLOAT)) returned 9
      2: ((METHOD FOO :AFTER (DOUBLE-FLOAT)) 2.0d0)
      2: (METHOD FOO :AFTER (DOUBLE-FLOAT)) returned DOUBLE
    1: (SB-PCL::COMBINED-METHOD FOO) returned 9
  0: FOO returned 9
9
```

Difference between defgeneric and defmethod: redefinition

There is a difference between declaring methods inside a `defgeneric` body or by writing multiple `defmethod`s: the two methods handle re-definition of methods differently. `defgeneric` will delete methods that are not in its body anymore.

Below we define a new generic function, using two `defmethod` that specialize on `person` and `child`:

```
(defmethod goodbye ((p person))
  (format t "goodbye ~a.~&" (name p)))

(defmethod goodbye ((c child))
  (format t "love you lil' one <3~&"))
```

You can try them with `(goodbye (make-instance 'person :name "you"))`.

Now, later in your work session, you decide that you don't need the one specializing on `child` any more. You delete its source code. But **the method still exists in the image**. You have to programmatically remove the method, see below.

Had you used `defgeneric`, all the methods would have been updated, added or deleted. We have defined the `tidy` generic function already with three methods:

```
(defgeneric tidy (obj)
  (:method-combination list)
```

```
  (:method list (obj)
    :foo)
  (:method list ((obj person))
    :books)
  (:method list ((obj child))
    :toys))
```

It works for any object type, a person or a child. Try it on a string: `(tidy "tidy what?")`, it works.

Now remove this declaration from the `defgeneric`:

```
(defgeneric tidy (obj)
  (:method-combination list)
  ;;(:method list (obj)  ;; <--- commented out
  ;;  :foo)
  (:method list ((obj person))
    :books)
  (:method list ((obj child))
    :toys))
```

Try to call it again: you get a “no applicable method” error:

```
There is no applicable method for the generic function
  #<STANDARD-GENERIC-FUNCTION TRADESIGNAL::TIDY (2)>
when called with arguments
  ("tidy what?").
```

This might or might not be important to you during development, but knowing this can help you keep your lisp image in sync with your source code. Otherwise, you can remove an old method when it gets on your way.

Removing a method

First, we need to find the method object:

```
(find-method #'goodbye nil (list (find-class 'child)))
;; => #<STANDARD-METHOD GOODBYE (CHILD) {10073EFD73}>
```

`find-method` takes as arguments: a function reference, a qualifier (like before, after or around), and a list of class specializers.

Once you found the method, use `remove-method`.

You could use `(fmakunbound 'goodbye)`, but this makes *all* methods unbound.

MOP

We gather here some examples that make use of the framework provided by the meta-object protocol, the configurable object system that rules Lisp's object system. We touch advanced concepts so, new reader, don't worry: you don't need to understand this section to start using the Common Lisp Object System.

We won't explain much about the MOP here, but hopefully sufficiently to make you see its possibilities or to help you understand how some CL libraries are built. We invite you to read the books referenced in the introduction.

Metaclasses

Metaclasses are needed to control the behaviour of other classes.

As announced, we won't talk much. See also Wikipedia for [metaclasses](#) or [CLOS](#).

The standard metaclass is `standard-class`:

```
(class-of p1) ;; #<STANDARD-CLASS PERSON>
```

But we'll change it to one of our own, so that we'll be able to **count the creation of instances**. This same mechanism could be used to auto increment the primary key of a database system (this is how the Postmodern or Mito libraries do), to log the creation of objects, etc.

Our metaclass inherits from `standard-class`:

```
(defclass counted-class (standard-class)
  ((counter :initform 0)))
#<STANDARD-CLASS COUNTED-CLASS>
```

```
#<STANDARD-CLASS COUNTED-CLASS>

(unintern 'person)
;; this is necessary to change the metaclass of person.
;; or (setf (find-class 'person) nil)
;; https://stackoverflow.com/questions/38811931/how-to-change-cl

(defclass person ()
  ((name
    :initarg :name
    :accessor name))
  (:metaclass counted-class)) ;; <- metaclass
;; #<COUNTED-CLASS PERSON>
;;   ^^^ not standard-class anymore.
```

The `:metaclass` class option can appear only once.

Actually you should have gotten a message asking to implement `validate-superclass`. So, still with the `closer-mop` library:

```
(defmethod closer-mop:validate-superclass ((class counted-class)
                                           (superclass standard-
  t)
```

Now we can control the creation of new `person` instances:

```
(defmethod make-instance :after ((class counted-class) &key)
  (incf (slot-value class 'counter)))
;; #<STANDARD-METHOD MAKE-INSTANCE :AFTER (COUNTED-CLASS) {1007
```

See that an `:after` qualifier is the safest choice, we let the standard method run as usual and return a new instance.

The `&key` is necessary, remember that `make-instance` is given initargs.

Now testing:

```
(defvar p3 (make-instance 'person :name "adam"))
#<PERSON {1007A8F5B3}>
```

```lisp
(slot-value p3 'counter)
;; => error. No, our new slot isn't on the person class.
(slot-value (find-class 'person) 'counter)
;; 1

(make-instance 'person :name "eve")
;; #<PERSON {1007AD5773}>
(slot-value (find-class 'person) 'counter)
;; 2
```

It's working.

Controlling the initialization of instances (initialize-instance)

To further control the creation of object instances, we can specialize the method `initialize-instance`. It is called by `make-instance`, just after a new instance was created but wasn't initialized yet with the default initargs and initforms.

It is recommended (Keene) to create an after method, since creating a primary method would prevent slots' initialization.

```lisp
(defmethod initialize-instance :after ((obj person) &key)
;; note the &key in the arglist:                    ^^^^
  (do something with obj))
```

A typical example would be to validate the initial values. Here we'll check that the person's name is longer than 3 characters:

```lisp
(defmethod initialize-instance :after ((obj person) &key)
  (with-slots (name) obj
    (assert (>= (length name) 3))))
```

So this call doesn't work anymore:

```lisp
(make-instance 'person :name "me")
;; The assertion (>= #1=(LENGTH NAME) 3) failed with #1# = 2.
;;   [Condition of type SIMPLE-ERROR]
```

We are prompted into the interactive debugger and we are given a choice of restarts (continue, retry, abort).

So while we're at it, here's an assertion that uses the debugger features to offer to change "name". We give `assert` a list of places that can be changed from the debugger:

```
(defmethod INITIALIZE-INSTANCE :after ((obj person) &key)
  (with-slots (name) obj
    (assert (>= (length name) 3)
            (name)  ;; <-- list of places
            "The value of name is ~a. It should be longer than 3
```

We get:

```
The value of name is me. It should be longer than 3 characters.
   [Condition of type SIMPLE-ERROR]

Restarts:
 0: [CONTINUE] Retry assertion with new value for NAME.
                               ^^^^^^^^^^^^ our new restart
 1: [RETRY] Retry SLIME REPL evaluation request.
 2: [*ABORT] Return to SLIME's top level.
```

Another rationale. The CLOS implementation of `make-instance` is in two stages: allocate the new object, and then pass it along with all the `make-instance` keyword arguments, to the generic function `initialize-instance`. Implementors and application writers define `:after` methods on `initialize-instance`, to initialize the slots of the instance. The system-supplied primary method does this with regard to (a) `:initform` and `:initarg` values supplied with the class was defined and (b) the keywords passed through from `make-instance`. Other methods can extend this behaviour as they see fit. For example, they might accept an additional keyword which invokes a database access to fill certain slots. The lambda list for `initialize-instance` is:

```
initialize-instance instance &rest initargs &key &allow-other-keys
```

Controlling the update of instances (update-instance-for-redefined-class)

Suppose you created a “circle” class, with coordinates and a diameter. Later on, you decide to replace the diameter by a radius. You want all the existing objects to be cleverly updated: the radius should have the diameter value, divided by 2. Use `update-instance-for-redefined-class`.

Its parameters are:

- instance: the object instance that is being updated
- added-slots: a list of added slots
- discarded-slots: a list of discarded slots
- property-list: a plist that captured the slot names and values of all the discarded-slots with values in the original instance.
- initargs: an initialization argument list. `&key` catches them below.

and it returns an object.

We actually don’t call the method direcly, but we use a `:before` method:

```
(defmethod update-instance-for-redefined-class
    :before ((obj circle) added deleted plist-values &key)
  (format t "plist values: ~a~&" plist-values)
  (let ((diameter (getf plist-values 'diameter)))
    (setf (radius obj) (/ diameter 2))))
```

Here’s how to try it. Start with a `circle` class:

```
(defclass circle ()
  ((diameter :accessor diameter :initform 9)))
```

and create a circle object:

```
(make-instance 'circle)
```

inspect it or check its diameter value.

Now write and compile a new class definition:

```lisp
(defclass circle ()
  ((radius :accessor radius)))
```

Nothing happens yet, you don’t see the output of our “plist values” print.

Inspect or `describe` the object: now it will be updated, and you’ll find the `radius` slot.

Existing objects are updated lazily.

See more on the HyperSpec or on the Community Spec.

Controlling the update of instances to new classes (update-instance-for-different-class)

Now imagine you are working with the `circle` class, but you realize you only need a `surface` kind of objects. You will discard the circle class altogether, but you want your existing objects to be updated -to this new class, and compute new slots intelligently. Use `update-instance-for-different-class`.

See more on the HyperSpec or on the Community Spec.

And see more in the books!

Type System

Common Lisp has a complete and flexible type system and corresponding tools to inspect, check and manipulate types. It allows creating custom types, adding type declarations to variables and functions and thus to get compile-time warnings and errors.

Values Have Types, Not Variables

Being different from some languages such as C/C++, variables in Lisp are just *placeholders* for objects[1]. When you `setf` a variable, an object is “placed” in it. You can place another value to the same variable later, as you wish.

This implies a fact that in Common Lisp **objects have types**, while variables do not. This might be surprising at first if you come from a C/C++ background.

For example:

```
(defvar *var* 1234)
*VAR*

(type-of *var*)
(INTEGER 0 4611686018427387903)
```

The function `type-of` returns the type of the given object. The returned result is a type-specifier. In this case the first element is the type and the remaining part is extra information (lower and upper bound) of that type. You can safely ignore it for now. Also remember that integers in Lisp have no limit!

Now let’s try to `setf` the variable:

```
* (setf *var* "hello")
"hello"

* (type-of *var*)
(SIMPLE-ARRAY CHARACTER (5))
```

You see, `type-of` returns a different result: `simple-array` of length 5 with contents of type `character`. This is because `*var*` is evaluated to string `"hello"` and the function `type-of` actually returns the type of object `"hello"` instead of variable `*var*`.

Type Hierarchy

The inheritance relationship of Lisp types consists a type graph and the root of all types is `T`. For example:

```
* (describe 'integer)
COMMON-LISP:INTEGER
  [symbol]

INTEGER names the built-in-class #<BUILT-IN-CLASS COMMON-LISP:IN
  Class precedence-list: INTEGER, RATIONAL, REAL, NUMBER, T
  Direct superclasses: RATIONAL
  Direct subclasses: FIXNUM, BIGNUM
  No direct slots.

INTEGER names a primitive type-specifier:
  Lambda-list: (&OPTIONAL (SB-KERNEL::LOW '*) (SB-KERNEL::HIGH '
```

The function `describe` shows that the symbol `integer` is a primitive type-specifier that has optional information lower bound and upper bound. Meanwhile, it is a built-in class. But why?

Most common Lisp types are implemented as CLOS classes. Some types are simply “wrappers” of other types. Each CLOS class maps to a corresponding type. In Lisp types are referred to indirectly by the use of `type specifiers`.

There are some differences between the function [`type-of`] and [`class-of`]. The function `type-of` returns the type of a given object in type specifier format while `class-of` returns the implementation details.

```
* (type-of 1234)
(INTEGER 0 4611686018427387903)

* (class-of 1234)
#<BUILT-IN-CLASS COMMON-LISP:FIXNUM>
```

Checking Types

The function `typep` can be used to check if the first argument is of the given type specified by the second argument.

```
* (typep 1234 'integer)
T
```

The function `subtypep` can be used to inspect if a type inherits from the another one. It returns 2 values:

- `T, T` means first argument is sub-type of the second one.
- `NIL, T` means first argument is *not* sub-type of the second one.
- `NIL, NIL` means “not determined”.

For example:

```
* (subtypep 'integer 'number)
T
T

* (subtypep 'string 'number)
NIL
T
```

Sometimes you may want to perform different actions according to the type of an argument. The macro `typecase` is your friend:

```
* (defun plus1 (arg)
    (typecase arg
      (integer (+ arg 1))
      (string (concatenate 'string arg "1"))
      (t 'error)))
PLUS1

* (plus1 100)
101 (7 bits, #x65, #o145, #b1100101)

* (plus1 "hello")
"hello1"

* (plus1 'hello)
ERROR
```

Type Specifier

A type specifier is a form specifying a type. As mentioned above, returning value of the function `type-of` and the second argument of `typep` are both type specifiers.

As shown above, `(type-of 1234)` returns `(INTEGER 0 4611686018427387903)`. This kind of type specifiers are called compound type specifier. It is a list whose head is a symbol indicating the type. The rest part of it is complementary information.

```
* (typep '#(1 2 3) '(vector number 3))
T
```

Here the complementary information of the type `vector` is its elements type and size respectively.

The rest part of a compound type specifier can be a `*`, which means “anything”. For example, the type specifier `(vector number *)` denotes a vector consisting of any number of numbers.

```
* (typep '#(1 2 3) '(vector number *))
T
```

The trailing parts can be omitted, the omitted elements are treated as `*`s:

```
* (typep '#(1 2 3) '(vector number))
T

* (typep '#(1 2 3) '(vector))
T
```

As you may have guessed, the type specifier above can be shortened as following:

```
* (typep '#(1 2 3) 'vector)
T
```

You may refer to the CLHS page for more information.

Defining New Types

You can use the macro `deftype` to define a new type-specifier.

Its argument list can be understood as a direct mapping to elements of rest part of a compound type specifier. They are defined as optional to allow symbol type specifier.

Its body should be a macro checking whether given argument is of this type (see `defmacro`).

We can use `member` to define enum types, for example:

```
(deftype fruit () '(member :apple :orange :pear))
```

Now let us define a new data type. The data type should be a array with at most 10 elements. Also each element should be a number smaller than 10. See following code for an example:

```
* (defun small-number-array-p (thing)
    (and (arrayp thing)
      (<= (length thing) 10)
      (every #'numberp thing)
```

```
      (every (lambda (x) (< x 10)) thing)))

* (deftype small-number-array (&optional type)
    `(and (array ,type 1)
          (satisfies small-number-array-p)))

* (typep '#(1 2 3 4) '(small-number-array number))
T

* (typep '#(1 2 3 4) 'small-number-array)
T

* (typep '#(1 2 3 4 100) 'small-number-array)
NIL

* (small-number-array-p '#(1 2 3 4 5 6 7 8 9 0 1))
NIL
```

Run-time type Checking

Common Lisp supports run-time type checking via the macro `check-type`. It accepts a `place` and a type specifier as arguments and signals an `type-error` if the contents of place are not of the given type.

```
* (defun plus1 (arg)
    (check-type arg number)
    (1+ arg))
PLUS1

* (plus1 1)
2 (2 bits, #x2, #o2, #b10)

* (plus1 "hello")
; Debugger entered on #<SIMPLE-TYPE-ERROR expected-type: NUMBER

The value of ARG is "Hello", which is not of type NUMBER.
   [Condition of type SIMPLE-TYPE-ERROR]
...
```

Compile-time type checking

You may provide type information for variables, function arguments etc via `proclaim`, `declaim` (at the toplevel) and `declare` (inside functions and macros).

However, similar to the `:type` slot introduced in CLOS section, the effects of type declarations are undefined in Lisp standard and are implementation specific. So there is no guarantee that the Lisp compiler will perform compile-time type checking.

However, it is possible, and SBCL is an implementation that does thorough type checking.

Let’s recall first that Lisp already warns about simple type warnings. The following function wrongly wants to concatenate a string and a number. When we compile it, we get a type warning.

```
(defconstant +foo+ 3)
(defun bar ()
  (concatenate 'string "+" +foo+))
; caught WARNING:
;   Constant 3 conflicts with its asserted type SEQUENCE.
;   See also:
;     The SBCL Manual, Node "Handling of Types"
```

The example is simple, but it already shows a capacity some other languages don’t have, and it is actually useful during development ;) Now, we’ll do better.

Declaring the type of variables

Use the macro `declaim` with a `type` declaration identifier (other identifiers are "ftype, inline, notinline, optimize…).

Let’s declare that our global variable `*name*` is a string. You can type the following in any order in the REPL:

```lisp
(declaim (type (string) *name*))
(defparameter *name* "book")
```

Now if we try to set it with a bad type, we get a `simple-type-error`:

```lisp
(setf *name* :me)
Value of :ME in (THE STRING :ME) is :ME, not a STRING.
   [Condition of type SIMPLE-TYPE-ERROR]
```

We can do the same with our custom types. Let's quickly declare the type `list-of-strings`:

```lisp
(defun list-of-strings-p (list)
  "Return t if LIST is non nil and contains only strings."
  (and (consp list)
       (every #'stringp list)))

(deftype list-of-strings ()
  `(satisfies list-of-strings-p))
```

Now let's declare that our `*all-names*` variables is a list of strings:

```lisp
(declaim (type (list-of-strings) *all-names*))
;; and with a wrong value:
(defparameter *all-names* "")
;; we get an error, still at compile-time:
Cannot set SYMBOL-VALUE of *ALL-NAMES* to "", not of type
(SATISFIES LIST-OF-STRINGS-P).
   [Condition of type SIMPLE-TYPE-ERROR]
```

Composing types

We can compose types. Following the previous example:

```lisp
(declaim (type (or null list-of-strings) *all-names*))
```

Declaring the input and output types of functions

We use again the `declaim` macro, with `ftype (function …)` instead of just `type`:

```lisp
(declaim (ftype (function (fixnum) fixnum) add))
;;                            ^^input ^^output [optional]
(defun add (n)
  (+ n  1))
```

With this we get nice type warnings at compile time.

If we change the function to erroneously return a string instead of a fixnum, we get a warning:

```lisp
(defun add (n)
  (format nil "~a" (+ n  1)))
; caught WARNING:
;   Derived type of ((GET-OUTPUT-STREAM-STRING STREAM)) is
;     (VALUES SIMPLE-STRING &OPTIONAL),
;   conflicting with the declared function return type
;     (VALUES FIXNUM &REST T).
```

If we use `add` inside another function, to a place that expects a string, we get a warning:

```lisp
(defun bad-concat (n)
  (concatenate 'string (add n)))
; caught WARNING:
;   Derived type of (ADD N) is
;     (VALUES FIXNUM &REST T),
;   conflicting with its asserted type
;     SEQUENCE.
```

If we use `add` inside another function, and that function declares its argument types which appear to be incompatible with those of `add`, we get a warning:

```lisp
(declaim (ftype (function (string)) bad-arg))
(defun bad-arg (n)
    (add n))
; caught WARNING:
;   Derived type of N is
;     (VALUES STRING &OPTIONAL),
;   conflicting with its asserted type
;     FIXNUM.
```

This all happens indeed *at compile time*, either in the REPL, either with a simple `C-c C-c` in Slime, or when we `load` a file.

Declaring &key parameters

Use `&key (:argument type)`.

For example:

(declaim (ftype (function (string &key (:n integer))) foo)) (defun foo (bar &key n) …)

Declaring &rest parameters

This is less evident, you might need a well-placed `declare`.

In the following, we declare a fruit type and we write a function that uses a single fruit argument, so compiling `placing-order` gives us a type warning as expected:

```
(deftype fruit () '(member :apple :orange :pear))

(declaim (ftype (function (fruit)) one-order))
(defun one-order (fruit)
  (format t "Ordering ~S~%" fruit))

(defun placing-order ()
  (one-order :bacon))
```

But in this version, we use `&rest` parameters, and we don't have a type warning anymore:

```
(declaim (ftype (function (&rest fruit)) place-order))
(defun place-order (&rest selections)
  (dolist (s selections)
    (format t "Ordering ~S~%" s)))

(defun placing-orders ()
  (place-order :orange :apple :bacon)) ;; => no type warning
```

The declaration is correct, but our compiler doesn't check it. A well-placed `declare` gives us the compile-time warning back:

```
(defun place-order (&rest selections)
  (dolist (s selections)
    (declare (type fruit s))      ;; <= declare
    (format t "Ordering ~S~%" s)))

(defun placing-orders ()
  (place-order :orange :apple :bacon))
```

=>

```
The value
  :BACON
is not of type
  (MEMBER :PEAR :ORANGE :APPLE)
```

For portable code, we would add run-time checks with an `assert`.

Declaring class slots types

A class slot accepts a `:type` slot option. It is however generally *not* used to check the type of the initform. SBCL, starting with version 1.5.9 released on november 2019, now gives those warnings, meaning that this:

```
(defclass foo ()
  ((name :type number :initform "17")))
```

throws a warning at compile time.

Note: see also sanity-clause, a data serialization/contract library to check slots' types during `make-instance` (which is not compile time).

Alternative type checking syntax: defstar, serapeum

The Serapeum library provides a shortcut that looks like this:

```
 (-> mod-fixnum+ (fixnum fixnum) fixnum)
 (defun mod-fixnum+ (x y) ...)
```

The Defstar library provides a `defun*` macro that allows to add the type declarations into the lambda list. It looks like this:

```
(defun* sum ((a real) (b real))
   (+ a b))
```

It also allows:

- to declare the return type, either in the function definition or in its body
- to quickly declare variables that are ignored, with the _ placeholder
- to add assertions for each arguments
- to do the same with `defmethod`, `defparameter`, `defvar`, `flet`, `labels`, `let*` and `lambda`.

Limitations

Complex types involving `satisfies` are not checked inside a function body by default, only at its boundaries. Even if it does a lot, SBCL doesn't do as much as a statically typed language.

Consider this example, where we badly increment an integer with a string:

```
(declaim (ftype (function () string) bad-adder))
(defun bad-adder ()
  (let ((res 10))
    (loop for name in '("alice")
       do (incf res name))  ;; <= bad
    (format nil "finally doing sth with ~a" res)))
```

Compiling this function doesn't throw a type warning.

However, if we had the problematic line at the function's boundary we'd get the warning:

```
(defun bad-adder ()
  (let ((res 10))
    (loop for name in  '("alice")
       return (incf res name))))
; in: DEFUN BAD-ADDER
```

```
;     (SB-INT:NAMED-LAMBDA BAD-ADDER
;         NIL
;       (BLOCK BAD-ADDER
;         (LET ((RES 10))
;           (LOOP FOR NAME IN *ALL-NAMES* RETURN (INCF RES NAME)
;
; caught WARNING:
;   Derived type of ("a hairy form" NIL (SETQ RES (+ NAME RES)))
;     (VALUES (OR NULL NUMBER) &OPTIONAL),
;   conflicting with the declared function return type
;     (VALUES STRING &REST T).
```

We could also use a `the` declaration in the loop body to get a compile-time warning:

```
       do (incf res (the string name)))
```

What can we conclude? This is yet another reason to decompose your code into small functions.

See also

- the article Static type checking in SBCL, by Martin Cracauer
- the article Typed List, a Primer - let's explore Lisp's fine-grained type hierarchy! with a shallow comparison to Haskell.
- the Coalton library: an efficient, statically typed functional programming language that supercharges Common Lisp. It is as an embedded DSL in Lisp that resembles Haskell or Standard ML, but lets you seamlessly interoperate with non-statically-typed Lisp code (and vice versa).
- exhaustiveness type checking at compile-time with Serapeum for enum types and union types (ecase-of, etypecase-of).

---

1. The term *object* here has nothing to do with Object-Oriented or so. It means "any Lisp datum".↩

TCP/UDP programming with sockets

This is a short guide to TCP/IP and UDP/IP client/server programming in Common Lisp using usockets.

TCP/IP

As usual, we will use quicklisp to load usocket.

(ql:quickload “usocket”)

Now we need to create a server. There are 2 primary functions that we need to call. `usocket:socket-listen` and `usocket:socket-accept`.

`usocket:socket-listen` binds to a port and listens on it. It returns a socket object. We need to wait with this object until we get a connection that we accept. That’s where `usocket:socket-accept` comes in. It’s a blocking call that returns only when a connection is made. This returns a new socket object that is specific to that connection. We can then use that connection to communicate with our client.

So, what were the problems I faced due to my mistakes?

Mistake 1 - My initial understanding was that `socket-accept` would return a stream object. NO…. It returns a socket object. In hindsight, its correct and my own mistake cost me time. So, if you want to write to the socket, you need to actually get the corresponding stream from this new socket. The socket object has a stream slot and we need to explicitly use that. And how does one know that? `(describe connection)` is your friend!

Mistake 2 - You need to close both the new socket and the server socket. Again this is pretty obvious but since my initial code was only closing the connection, I kept running into a socket in use problem. Of course one more option is to reuse the socket when we listen.

Once you get past these mistakes, it's pretty easy to do the rest. Close the connections and the server socket and boom you are done!

```
(defun create-server (port)
  (let* ((socket (usocket:socket-listen "127.0.0.1" port))
     (connection (usocket:socket-accept socket :element-type
                      'character)))
    (unwind-protect
        (progn
          (format (usocket:socket-stream connection)
                  "Hello World~%")
          (force-output (usocket:socket-stream connection)))
      (progn
        (format t "Closing sockets~%")
        (usocket:socket-close connection)
        (usocket:socket-close socket)))))
```

Now for the client. This part is easy. Just connect to the server port and you should be able to read from the server. The only silly mistake I made here was to use read and not read-line. So, I ended up seeing only a "Hello" from the server. I went for a walk and came back to find the issue and fix it.

```
(defun create-client (port)
  (usocket:with-client-socket (socket stream "127.0.0.1" port
                                      :element-type 'character)
    (unwind-protect
         (progn
           (usocket:wait-for-input socket)
           (format t "Input is: ~a~%" (read-line stream)))
      (usocket:socket-close socket))))
```

So, how do you run this? You need two REPLs, one for the server and one for the client. Load this file in both REPLs. Create the server in the first REPL.

(create-server 12321)

Now you are ready to run the client on the second REPL

(create-client 12321)

Voilà! You should see “Hello World” on the second REPL.

UDP/IP

As a protocol, UDP is connection-less, and therefore there is no concept of binding and accepting a connection. Instead we only do a `socket-connect` but pass a specific set of parameters to make sure that we create an UDP socket that’s waiting for data on a particular port.

So, what were the problems I faced due to my mistakes? Mistake 1 - Unlike TCP, you don’t pass host and port to `socket-connect`. If you do that, then you are indicating that you want to send a packet. Instead, you pass `nil` but you set `:local-host` and `:local-port` to the address and port that you want to receive data on. This part took some time to figure out, because the documentation didn’t cover it. Instead reading a bit of code from [blackthorn-engine-3d]() helped a lot.

Also, since UDP is connectionless, anyone can send data to it at any time. So, we need to know which host/port did we get data from so that we can respond on it. So we bind multiple values to `socket-receive` and use those values to send back data to our peer “client”.

```
(defun create-server (port buffer)
  (let* ((socket (usocket:socket-connect nil nil
                    :protocol :datagram
                    :element-type '(unsigned-byte 8)
                    :local-host "127.0.0.1"
                    :local-port port)))
    (unwind-protect
     (multiple-value-bind (buffer size client receive-port)
         (usocket:socket-receive socket buffer 8)
       (format t "~A~%" buffer)
       (usocket:socket-send socket (reverse buffer) size
               :port receive-port
               :host client))
      (usocket:socket-close socket))))
```

Now for the sender/receiver. This part is pretty easy. Create a socket, send data on it and receive data back.

```
(defun create-client (port buffer)
  (let ((socket (usocket:socket-connect "127.0.0.1" port
                      :protocol :datagram
                      :element-type '(unsigned-byte 8))))
    (unwind-protect
     (progn
       (format t "Sending data~%")
       (replace buffer #(1 2 3 4 5 6 7 8))
       (format t "Receiving data~%")
       (usocket:socket-send socket buffer 8)
       (usocket:socket-receive socket buffer 8)
       (format t "~A~%" buffer))
      (usocket:socket-close socket))))
```

So, how do you run this? You need again two REPLs, one for the server and one for the client. Load this file in both REPLs. Create the server in the first REPL.

(create-server 12321 (make-array 8 :element-type ’(unsigned-byte 8)))

Now you are ready to run the client on the second REPL

(create-client 12321 (make-array 8 :element-type ’(unsigned-byte 8)))

Voilà! You should see a vector `#(1 2 3 4 5 6 7 8)` on the first REPL and `#(8 7 6 5 4 3 2 1)` on the second one.

Credit

This guide originally comes from [shortsightedsid]

Interfacing with your OS

The ANSI Common Lisp standard doesn't mention this topic. (Keep in mind that it was written at a time where Lisp Machines were at their peak. On these boxes Lisp *was* your operating system!) So almost everything that can be said here depends on your OS and your implementation. There are, however, some widely used libraries, which either come with your Common Lisp implementation, or are easily available through Quicklisp. These include:

- ASDF3, which is included with almost all Common Lisp implementations, includes Utilities for Implementation- and OS-Portability (UIOP).
- osicat
- unix-opts or the newer clingon are a command-line argument parsers, similar to Python's `argparse`.

Accessing Environment variables

UIOP comes with a function that'll allow you to look at Unix/Linux environment variables on a lot of different CL implementations:

```
* (uiop:getenv "HOME")
  "/home/edi"
```

Below is an example implementation, where we can see /feature flags/ used to run code on specific implementations:

```
* (defun my-getenv (name &optional default)
    "Obtains the current value of the POSIX environment variable
    (declare (type (or string symbol) name))
    (let ((name (string name)))
      (or #+abcl (ext:getenv name)
         #+ccl (ccl:getenv name)
         #+clisp (ext:getenv name)
         #+cmu (unix:unix-getenv name) ; since CMUCL 20b
         #+ecl (si:getenv name)
```

```
         #+ecl (si:getenv name)
         #+gcl (si:getenv name)
         #+mkcl (mkcl:getenv name)
         #+sbcl (sb-ext:posix-getenv name)
         default)))
MY-GETENV
* (my-getenv "HOME")
"/home/edi"
* (my-getenv "HOM")
NIL
* (my-getenv "HOM" "huh?")
"huh?"
```

You should also note that some of these implementations also provide the ability to *set* these variables. These include ECL (`si:setenv`) and AllegroCL, LispWorks, and CLISP where you can use the functions from above together with [`setf`](). This feature might be important if you want to start subprocesses from your Lisp environment.

Also note that the [Osicat]() library has the method `(environment-variable "name")`, on POSIX-like systems including Windows. It is also `fset`-able.

Accessing the command line arguments

Basics

Accessing command line arguments is implementation-specific but it appears most implementations have a way of getting at them. UIOP with `uiop:command-line-arguments` or [Roswell]() as well as external libraries (see next section) make it portable.

[SBCL]() stores the arguments list in the special variable `sb-ext:*posix-argv*`

```
$ sbcl my-command-line-arg
```

....

```
* sb-ext:*posix-argv*

("sbcl" "my-command-line-arg")
*
```

More on using this to write standalone Lisp scripts can be found in the SBCL Manual

LispWorks has `system:*line-arguments-list*`

```
* system:*line-arguments-list*
("/Users/cbrown/Projects/lisptty/tty-lispworks" "-init" "/Users/
```

Here’s a quick function to return the argument strings list across multiple implementations:

```
(defun my-command-line ()
  (or
   #+SBCL *posix-argv*
   #+LISPWORKS system:*line-arguments-list*))
```

Now it would be handy to access them in a portable way and to parse them according to a schema definition.

Parsing command line arguments

We have a look at the Awesome CL list#scripting section and we’ll show how to use clingon.

Please see our scripting recipe.

Running external programs

uiop has us covered, and is probably included in your Common Lisp implementation.

Synchronously

`uiop:run-program` either takes a string as argument, denoting the name of the executable to run, or a list of strings, for the program and its arguments:

```
(uiop:run-program "firefox")
```

or

```
(uiop:run-program (list "firefox" "http:url"))
```

This will process the program output as specified and return the processing results when the program and its output processing are complete.

Use `:output t` to print to standard output.

This function has the following optional arguments:

```
run-program (command &rest keys &key
               ignore-error-status
               (force-shell nil force-shell-suppliedp)
               input
               (if-input-does-not-exist :error)
               output
               (if-output-exists :supersede)
               error-output
               (if-error-output-exists :supersede)
               (element-type *default-stream-element-type*)
               (external-format *utf-8-external-format*)
              allow-other-keys)
```

It will always call a shell (rather than directly executing the command when possible) if `force-shell` is specified. Similarly, it will never call a shell if `force-shell` is specified to be `nil`.

Signal a continuable `subprocess-error` if the process wasn't successful (exit-code 0), unless `ignore-error-status` is specified.

If `output` is a pathname, a string designating a pathname, or `nil` (the default) designating the null device, the file at that path is used as output. If it's `:interactive`, output is inherited from the current process; beware that this may be different from your `*standard-output*`, and under `slime` will be on your `*inferior-lisp*` buffer. If it's `t`, output goes to your current `*standard-output*` stream. Otherwise, `output` should be a value that is a suitable first argument to `slurp-input-stream` (qv.), or a list of such a value and keyword arguments. In this case, `run-program` will create a temporary stream for the program output; the program output, in that stream, will be processed by a call to `slurp-input-stream`, using `output` as the first argument (or the first element of `output`, and the rest as keywords). The primary value resulting from that call (or `nil` if no call was needed) will be the first value returned by `run-program.` E.g., using `:output :string` will have it return the entire output stream as a string. And using `:output '(:string :stripped t)` will have it return the same string stripped of any ending newline.

`if-output-exists`, which is only meaningful if `output` is a string or a pathname, can take the values `:error`, `:append`, and `:supersede` (the default). The meaning of these values and their effect on the case where `output` does not exist, is analogous to the `if-exists` parameter to `open` with `:direction :output`.

`error-output` is similar to `output`, except that the resulting value is returned as the second value of `run-program`. `t` designates the `*error-output*`. Also `:output` means redirecting the error output to the output stream, in which case `nil` is returned.

`if-error-output-exists` is similar to `if-output-exist`, except that it affects `error-output` rather than `output`.

`input` is similar to `output`, except that `vomit-output-stream` is used, no value is returned, and T designates the `*standard-input*`.

`if-input-does-not-exist`, which is only meaningful if `input` is a string or a pathname, can take the values `:create` and `:error` (the default). The meaning of these values is analogous to the `if-does-not-exist` parameter to `open` with `:direction :input`.

`element-type` and `external-format` are passed on to your Lisp implementation, when applicable, for creation of the output stream.

One and only one of the stream slurping or vomiting may or may not happen in parallel in parallel with the subprocess, depending on options and implementation, and with priority being given to output processing. Other streams are completely produced or consumed before or after the subprocess is spawned, using temporary files.

`run-program` returns 3 values:

- the result of the `output` slurping if any, or `nil`
- the result of the `error-output` slurping if any, or `nil`
- either 0 if the subprocess exited with success status, or an indication of failure via the `exit-code` of the process

Asynchronously

With `uiop:launch-program`.

Its signature is the following:

```
launch-program (command &rest keys &key
                    input
                    (if-input-does-not-exist :error)
                    output
                    (if-output-exists :supersede)
                    error-output
                    (if-error-output-exists :supersede)
                    (element-type *default-stream-element-type*)
                    (external-format *utf-8-external-format*)
                    directory
                    #+allegro separate-streams
                    &allow-other-keys)
```

Output (stdout) from the launched program is set using the `output` keyword:

- If `output` is a pathname, a string designating a pathname, or `nil` (the default) designating the null device, the file at that path is used as output.

- If it’s `:interactive`, output is inherited from the current process; beware that this may be different from your `*standard-output*`, and under Slime will be on your `*inferior-lisp*` buffer.
- If it’s `T`, output goes to your current `*standard-output*` stream.
- If it’s `:stream`, a new stream will be made available that can be accessed via `process-info-output` and read from.
- Otherwise, `output` should be a value that the underlying lisp implementation knows how to handle.

`if-output-exists`, which is only meaningful if `output` is a string or a pathname, can take the values `:error`, `:append`, and `:supersede` (the default). The meaning of these values and their effect on the case where `output` does not exist, is analogous to the `if-exists` parameter to `open` with `:DIRECTION :output`.

`error-output` is similar to `output`. T designates the `*error-output*`, `:output` means redirecting the error output to the output stream, and `:stream` causes a stream to be made available via `process-info-error-output`.

`launch-program` returns a `process-info` object, which look like the following (source):

```
(defclass process-info ()
    (
     ;; The advantage of dealing with streams instead of PID is
     ;; availability of functions like `sys:pipe-kill-process`.
     (process :initform nil)
     (input-stream :initform nil)
     (output-stream :initform nil)
     (bidir-stream :initform nil)
     (error-output-stream :initform nil)
     ;; For backward-compatibility, to maintain the property (ze
     ;; exit-code) <-> success, an exit in response to a signal
     ;; encoded as 128+signum.
     (exit-code :initform nil)
     ;; If the platform allows it, distinguish exiting with a co
     ;; >128 from exiting in response to a signal by setting thi
     (signal-code :initform nil)))
```

See the docstrings.

Test if a subprocess is alive

`uiop:process-alive-p` tests if a process is still alive, given a `process-info` object returned by `launch-program`:

```
* (defparameter *shell* (uiop:launch-program "bash"
                            :input :stream :output :stream))

;; inferior shell process now running
* (uiop:process-alive-p *shell*)
T

;; Close input and output streams
* (uiop:close-streams *shell*)
* (uiop:process-alive-p *shell*)
NIL
```

Get the exit code

We can use `uiop:wait-process`. If the process is finished, it returns immediately, and returns the exit code. If not, it waits for the process to terminate.

```
(uiop:process-alive-p *process*)
NIL
(uiop:wait-process *process*)
0
```

An exit code to 0 means success (use `zerop`).

The exit code is also stored in the `exit-code` slot of our `process-info` object. We see from the class definition above that it has no accessor, so we'll use `slot-value`. It has an `initform` to nil, so we don't have to check if the slot is bound. We can do:

```
(slot-value *my-process* 'uiop/launch-program::exit-code)
0
```

The trick is that we *must* run `wait-process` beforehand, otherwise the result will be `nil`.

Since `wait-process` is blocking, we can do it on a new thread:

```
(bt:make-thread
  (lambda ()
    (let ((exit-code (uiop:wait-process
                       (uiop:launch-program (list "of" "commands
      (if (zerop exit-code)
          (print :success)
          (print :failure)))))
  :name "Waiting for <program>")
```

Note that `run-program` returns the exit code as the third value.

Input and output from subprocess

If the `input` keyword is set to `:stream`, then a stream is created and can be written to in the same way as a file. The stream can be accessed using `uiop:process-info-input`:

```
;; Start the inferior shell, with input and output streams
* (defparameter *shell* (uiop:launch-program "bash"
                                             :input :stream :output :stream))
;; Write a line to the shell
* (write-line "find . -name '*.md'"
     (uiop:process-info-input *shell*))
;; Flush stream
* (force-output (uiop:process-info-input *shell*))
```

where write-line writes the string to the given stream, adding a newline at the end. The force-output call attempts to flush the stream, but does not wait for completion.

Reading from the output stream is similar, with `uiop:process-info-output` returning the output stream:

```
* (read-line (uiop:process-info-output *shell*))
```

In some cases the amount of data to be read is known, or there are delimiters to determine when to stop reading. If this is not the case, then calls to read-line can hang while waiting for data. To avoid this, listen can be used to test if a character is available:

```
* (let ((stream (uiop:process-info-output *shell*)))
     (loop while (listen stream) do
         ;; Characters are immediately available
         (princ (read-line stream))
         (terpri)))
```

There is also read-char-no-hang which reads a single character, or returns `nil` if no character is available. Note that due to issues like buffering, and the timing of when the other process is executed, there is no guarantee that all data sent will be received before `listen` or `read-char-no-hang` return `nil`.

Capturing standard and error output

Capturing standard output, as seen above, is easily done by telling `:output` to be `:string`, or using `:output '(:string :stripped t)` to strip any ending newline.

You can ask the same to `:error-output` and, in addition, you can ask `uiop:run-program` to *not* signal an error, thus to not enter the interactive debugger, with `:ignore-error-status t`.

In that case, you can check the success or the failure of the program with the returned `exit-code`. 0 is success.

Here’s everything together:

```
(uiop:run-program (list "git"
                        "checkout"
                        "me/does-not-exist")
                  :output :string
                  :error-output :string
                  :ignore-error-status t)
;; =>
""
```

```
"error: pathspec 'me/does-not-exist did not match any file(s) kr
"
1
```

`uiop:run-program` returns 3 values:

- the standard output (here, as a blank string)
- the error output (here, as a string with our error message)
- the exit code

We can bind them with `multiple-value-bind`:

```
(multiple-value-bind (output error-output exit-code)
    (uiop:run-program (list …))
  (unless (zerop exit-code)
    (format t "error output is: ~a" error-output)))
```

Running visual commands (htop)

Use `uiop:run-program` and set both `:input` and `:output` to `:interactive`:

```
(uiop:run-program "htop"
                  :output :interactive
                  :input :interactive)
```

This will spawn `htop` in full screen, as it should.

It works for more commands (`sudo`, `vim`…), however not for all interactive programs, such as `less` or `fzf`.

Piping

Here's an example to do the equivalent of `ls | sort`. Note that "ls" uses `launch-program` (async) and outputs to a stream, where "sort", the last command of the pipe, uses `run-program` and outputs to a string.

```
(uiop:run-program "sort"
                   :input
```

```
(uiop:process-info-output
 (uiop:launch-program "ls"
                      :output :stream))
:output :string)
```

Get Lisp’s current Process ID (PID)

Implementations provide their own functions for this.

On SBCL:

```
(sb-posix:getpid)
```

It is possible portably with the osicat library:

```
(osicat-posix:getpid)
```

Here again, we could find it by using the `apropos` function:

```
CL-USER> (apropos "pid")
OSICAT-POSIX:GETPID (fbound)
OSICAT-POSIX::PID
[…]
SB-IMPL::PID
SB-IMPL::WAITPID (fbound)
SB-POSIX:GETPID (fbound)
SB-POSIX:GETPPID (fbound)
SB-POSIX:LOG-PID (bound)
SB-POSIX::PID
SB-POSIX::PID-T
SB-POSIX:WAITPID (fbound)
[…]
```

Foreign Function Interfaces

The ANSI Common Lisp standard doesn't mention this topic. So almost everything that can be said here depends on your OS and your implementation.

Example: Calling 'gethostname' from CLISP

Note: You should read the relevant chapter from the CLISP implementation notes before you proceed.

`int gethostname(char *name, int len)` follows a typical pattern of C "out"-parameter convention - it expects a pointer to a buffer it's going to fill. So you must view this parameter as either `:OUT` or `:IN-OUT`. Additionally, one must tell the function the size of the buffer. Here `len` is just an `:IN` parameter. Sometimes this will be an `:IN-OUT` parameter, returning the number of bytes actually filled in.

So `name` is actually a pointer to an array of up to `len` characters, regardless of what the poor "`char *`" C prototype says, to be used like a C string (0-termination). How many elements are in the array? Luckily, in our case, you can find it out without calculating the `sizeof()` a C structure. It's a hostname that will be returned. The Solaris 2.x manpage says "Host names are limited to MAXHOSTNAMELEN characters, currently 256."

Also, in the present example, you can use allocation `:ALLOCA`, like you'd do in C: stack-allocate a temporary. Why make things worse when using Lisp than when using C?

This yields the following useful signature for your foreign function:

```
(ffi:def-c-call-out gethostname
     (:arguments (name (ffi:c-ptr (ffi:c-array-max ffi:char 256)
     :out :alloca)
     (len ffi:int))
     ;; (:return-type BOOLEAN) could have been used here
```

```
;; (Solaris says it's either 0 or -1).
(:return-type ffi:int))

(defun myhostname ()
(multiple-value-bind (success name)
;; :OUT or :IN-OUT parameters are returned via multiple val
(gethostname 256)
(if (zerop success)
(subseq name 0 (position #\null name))
(error ... ; errno may be set
...))))
(defvar hostname (myhostname))
```

Possibly `SUBSEQ` and `POSITION` are superfluous, thanks to `C-ARRAY-MAX` as opposed to `C-ARRAY`:

```
(defun myhostname ()
(multiple-value-bind (success name)
;; :out or :in-out parameters are returned via multiple val
(gethostname 256)
(if (zerop success) name
(error ... ; errno may be set
...))))
```

Example: Calling ‘gethostname’ from Allegro CL

This is how the same example above would be written in Allegro Common Lisp version 6 and above. ACL doesn’t explicitly distinguish between `input` and `output` arguments. The way to declare an argument as `output` (i.e., modifiable by C) is to use an array, since arrays are passed by reference and C therefore receives a pointer to a memory location (which is what it expects). In this case things are made even easier by the fact that `gethostname()` expects an array of char, and a `SIMPLE-ARRAY` of `CHARACTER` represents essentially the same thing in Lisp. The foreign function definition is therefore the following:

```
(def-foreign-call (c-get-hostname "gethostname")
((name (* :char) (simple-array 'character (*)))
```

```
    (len :int integer))
    :returning :int)
```

Let's read this line by line: this form defines a Lisp function called `C-GET-HOSTNAME` that calls the C function `gethostname()`. It takes two arguments: the first one, called `NAME`, is a pointer to a char (`*char` in C), and a `SIMPLE-ARRAY` of characters in Lisp; the second one is called `LEN`, and is an integer. The function returns an integer value.

And now the Lisp side:

```
(defun get-hostname ()
    (let* ((name (make-array 256 :element-type 'character))
    (result (c-get-hostname name 256)))
    (if (zerop result)
    (let ((pos (position #\null name)))
    (subseq name 0 pos))
    (error "gethostname() failed."))))
```

This function creates the `NAME` array, calls `C-GET-HOSTNAME` to fill it and then checks the returned value. If the value is zero, then the call was successful, and we return the contents of `NAME` up to the first 0 character (the string terminator in C), otherwise we signal an error. Note that, unlike the previous example, we allocate the string in Lisp, and we rely on the Lisp garbage collector to get rid of it after the function terminates. Here is a usage example:

```
* (get-hostname)
    "terminus"
```

Working with strings is, in general, easier than the previous example showed. Let's say you want to call `getenv()` from Lisp to access the value of an environment variable. `getenv()` takes a string argument (the variable name) and returns another string (the variable value). To be more precise, the argument is a *pointer* to a sequence of characters that should have been allocated by the caller, and the return value is a pointer to an already-existing sequence of chars (in the environment). Here is the definition of `C-GETENV`:

```
(def-foreign-call (c-getenv "getenv")
     ((var (* :char) string))
     :returning :int
     :strings-convert t)
```

The argument in this case is still a pointer to char in C, but we can declare it a `STRING` to Lisp. The return value is a pointer, so we declare it as integer. Finally, the `:STRINGS-CONVERT` keyword argument specifies that ACL should automatically translate the Lisp string passed as the first argument into a C string. Here is how it's used:

```
* (c-getenv "SHELL")
     -1073742215
```

If you are surprised by the return value, just remember that `C-GETENV` returns a pointer, and we must tell Lisp how to interpret the contents of the memory location pointed to by it. Since in this case we know that it will point to a C string, we can use the `FF:NATIVE-TO-STRING` function to convert it to a Lisp string:

```
* (native-to-string (c-getenv "SHELL"))
     "/bin/tcsh"
     9
     9
```

(The second and third values are the number of characters and bytes copied, respectively). One caveat: if you ask for the value of a non-existent variable, `C-GETENV` will return 0, and `NATIVE-TO-STRING` will fail. So a safer example would be:

```
* (let ((ptr (c-getenv "NOSUCHVAR")))
     (unless (zerop ptr)
     (native-to-string ptr)))
     NIL
```

Threads, concurrency, parallelism

Introduction

By *threads*, we mean separate execution strands within a single Lisp process, sharing the same address space. Typically, execution is automatically switched between these strands by the system (either by the lisp kernel or by the operating system) so that tasks appear to be completed in parallel (asynchronously). This page discusses the creation and management of threads and some aspects of interactions between them. For information about the interaction between lisp and other *processes*, see [Interfacing with your OS].

An instant pitfall for the unwary is that most implementations refer (in nomenclature) to threads as *processes* - this is a historical feature of a language which has been around for much longer than the term *thread*. Call this maturity a sign of stable implementations, if you will.

The ANSI Common Lisp standard doesn't mention this topic. We will present here the portable [bordeaux-threads] library, an example implementation via [SBCL threads] from the [SBCL Manual], and the [lparallel] library ([GitHub]).

Bordeaux-threads is a de-facto standard portable library, that exposes rather low-level primitives. Lparallel builds on it and features:

- a simple model of task submission with receiving queue
- constructs for expressing fine-grained parallelism
- **asynchronous condition handling** across thread boundaries
- **parallel versions of map, reduce, sort, remove**, and many others
- **promises**, futures, and delayed evaluation constructs
- computation trees for parallelizing interconnected tasks
- bounded and unbounded FIFO **queues**
- **channels**
- high and low priority tasks

- task killing by category
- integrated timeouts

For more libraries on parallelism and concurrency, see the Awesome CL list and Quickdocs such as quickdocks on thread and concurrency.

Why bother?

The first question to resolve is: why bother with threads? Sometimes your answer will simply be that your application is so straightforward that you need not concern yourself with threads at all. But in many other cases it’s difficult to imagine how a sophisticated application can be written without multi-threading. For example:

- you might be writing a server which needs to be able to respond to more than one user / connection at a time (for instance: a web server) on the Sockets page);
- you might want to perform some background activity, without halting the main application while this is going on;
- you might want your application to be notified when a certain time has elapsed;
- you might want to keep the application running and active while waiting for some system resource to become available;
- you might need to interface with some other system which requires multithreading (for example, “windows” under Windows which generally run in their own threads);
- you might want to associate different contexts (e.g. different dynamic bindings) with different parts of the application;
- you might even have the simple need to do two things at once.

What is Concurrency? What is Parallelism?

Credit: The following was first written on z0ltan.wordpress.com by Timmy Jose.

Concurrency is a way of running different, possibly related, tasks seemingly simultaneously. What this means is that even on a single processor machine, you can simulate simultaneity using threads (for instance) and context-switching them.

In the case of system (native OS) threads, the scheduling and context switching is ultimately determined by the OS. This is the case with Java threads and Common Lisp threads.

In the case of “green” threads, that is to say threads that are completely managed by the program, the scheduling can be completely controlled by the program itself. Erlang is a great example of this approach.

So what is the difference between Concurrency and Parallelism? Parallelism is usually defined in a very strict sense to mean independent tasks being run in parallel, simultaneously, on different processors or on different cores. In this narrow sense, you really cannot have parallelism on a single-core, single-processor machine.

It rather helps to differentiate between these two related concepts on a more abstract level – concurrency primarily deals with providing the illusion of simultaneity to clients so that the system doesn’t appear locked when a long running operation is underway. GUI systems are a wonderful example of this kind of system. Concurrency is therefore concerned with providing good user experience and not necessarily concerned with performance benefits.

Java’s Swing toolkit and JavaScript are both single-threaded, and yet they can give the appearance of simultaneity because of the context switching behind the scenes. Of course, concurrency is implemented using multiple threads/processes in most cases.

Parallelism, on the other hand, is mostly concerned with pure performance gains. For instance, if we are given a task to find the squares of all the even numbers in a given range, we could divide the range into chunks which are then run in parallel on different cores or different processors, and then the results can be collated together to form the final result. This is an example of Map-Reduce in action.

So now that we have separated the abstract meaning of Concurrency from that of Parallelism, we can talk a bit about the actual mechanism used to implement them. This is where most of the confusion arise for a lot of people. They tend to tie down abstract concepts with specific means of implementing them. In essence, both abstract concepts may be implemented using the same mechanisms! For instance, we may implement concurrent features and parallel features using the same basic thread mechanism in Java. It's only the conceptual intertwining or independence of tasks at an abstract level that makes the difference for us.

For instance, if we have a task where part of the work can be done on a different thread (possibly on a different core/processor), but the thread which spawns this thread is logically dependent on the results of the spawned thread (and as such has to "join" on that thread), it is still Concurrency!

So the bottomline is this – Concurrency and Parallelism are different concepts, but their implementations may be done using the same mechanisms — threads, processes, etc.

Bordeaux threads

The Bordeaux library provides a platform independent way to handle basic threading on multiple Common Lisp implementations. The interesting bit is that it itself does not really create any native threads — it relies entirely on the underlying implementation to do so.

On the other hand, it does provide some useful extra features in its own abstractions over the lower-level threads.

Also, you can see from the demo programs that a lot of the Bordeaux functions seem quite similar to those used in SBCL. I don't really think that this is a coincidence.

You can refer to the documentation for more details (check the "Wrap-up" section).

Installing Bordeaux Threads

First let's load up the Bordeaux library using Quicklisp:

```
CL-USER> (ql:quickload "bt-semaphore")
To load "bt-semaphore":
  Load 1 ASDF system:
    bt-semaphore
; Loading "bt-semaphore"

(:BT-SEMAPHORE)
```

Checking for thread support in Common Lisp

Regardless of the Common Lisp implementation, there is a standard way to check for thread support availability:

```
CL-USER> (member :thread-support *FEATURES*)
(:THREAD-SUPPORT :SWANK :QUICKLISP :ASDF-PACKAGE-SYSTEM :ASDF3.1
 :ASDF :OS-MACOSX :OS-UNIX :NON-BASE-CHARS-EXIST-P :ASDF-UNICODE
 :64-BIT-REGISTERS :ALIEN-CALLBACKS :ANSI-CL :ASH-RIGHT-VOPS :BS
 :C-STACK-IS-CONTROL-STACK :COMMON-LISP :COMPARE-AND-SWAP-VOPS
 :COMPLEX-FLOAT-VOPS :CYCLE-COUNTER :DARWIN :DARWIN9-OR-BETTER :
 :FP-AND-PC-STANDARD-SAVE :GENCGC :IEEE-FLOATING-POINT :INLINE-C
 :INODE64 :INTEGER-EQL-VOP :LINKAGE-TABLE :LITTLE-ENDIAN
 :MACH-EXCEPTION-HANDLER :MACH-O :MEMORY-BARRIER-VOPS :MULTIPLY-
 :OS-PROVIDES-BLKSIZE-T :OS-PROVIDES-DLADDR :OS-PROVIDES-DLOPEN
 :OS-PROVIDES-PUTWC :OS-PROVIDES-SUSECONDS-T :PACKAGE-LOCAL-NICK
 :PRECISE-ARG-COUNT-ERROR :RAW-INSTANCE-INIT-VOPS :SB-DOC :SB-EV
 :SB-PACKAGE-LOCKS :SB-SIMD-PACK :SB-SOURCE-LOCATIONS :SB-TEST :
 :SB-UNICODE :SBCL :STACK-ALLOCATABLE-CLOSURES :STACK-ALLOCATABL
 :STACK-ALLOCATABLE-LISTS :STACK-ALLOCATABLE-VECTORS
 :STACK-GROWS-DOWNWARD-NOT-UPWARD :SYMBOL-INFO-VOPS :UD2-BREAKPC
 :UNWIND-TO-FRAME-AND-CALL-VOP :X86-64)
```

If there were no thread support, it would show "NIL" as the value of the expression.

Depending on the specific library being used, we may also have different ways of checking for concurrency support, which may be used instead of the common check mentioned above.

For instance, in our case, we are interested in using the Bordeaux library. To check whether there is support for threads using this library, we can see whether the `*supports-threads-p*` global variable is set to NIL (no support) or T (support available):

```
CL-USER> bt:*supports-threads-p*
T
```

Okay, now that we've got that out of the way, let's test out both the platform-independent library (Bordeaux) as well as the platform-specific support (SBCL in this case).

To do this, let us work our way through a number of simple examples:

- Basics — list current thread, list all threads, get thread name
- Update a global variable from a thread
- Print a message onto the top-level using a thread
- Print a message onto the top-level — fixed
- Print a message onto the top-level — better
- Modify a shared resource from multiple threads
- Modify a shared resource from multiple threads — fixed using locks
- Modify a shared resource from multiple threads — using atomic operations
- Joining on a thread, destroying a thread example

Basics — list current thread, list all threads, get thread name

```
;;; Print the current thread, all the threads, and the curre
(defun print-thread-info ()
  (let* ((curr-thread (bt:current-thread))
         (curr-thread-name (bt:thread-name curr-thread))
         (all-threads (bt:all-threads)))
    (format t "Current thread: ~a~%~%" curr-thread)
    (format t "Current thread name: ~a~%~%" curr-thread-name

    (format t "All threads:~% ~{~a~%~}~%" all-threads))
  nil)
```

And the output:

```
CL-USER> (print-thread-info)
Current thread: #<THREAD "repl-thread" RUNNING {10043B8003}>

Current thread name: repl-thread

All threads:
 #<THREAD "repl-thread" RUNNING {10043B8003}>
#<THREAD "auto-flush-thread" RUNNING {10043B7DA3}>
#<THREAD "swank-indentation-cache-thread" waiting on: #<WAIT
#<THREAD "reader-thread" RUNNING {1003A20063}>
#<THREAD "control-thread" waiting on: #<WAITQUEUE  {1003A19E
#<THREAD "Swank Sentinel" waiting on: #<WAITQUEUE  {10037900
#<THREAD "main thread" RUNNING {1002991CE3}>

NIL
```

Update a global variable from a thread:

```
(defparameter *counter* 0)

(defun test-update-global-variable ()
  (bt:make-thread
   (lambda ()
     (sleep 1)
     (incf *counter*)))
  *counter*)
```

We create a new thread using `bt:make-thread`, which takes a lambda abstraction as a parameter. Note that this lambda abstraction cannot take any parameters.

Another point to note is that unlike some other languages (Java, for instance), there is no separation from creating the thread object and starting/running it. In this case, as soon as the thread is created, it is executed.

The output:

```
CL-USER> (test-update-global-variable)

0
CL-USER> *counter*
1
```

As we can see, because the main thread returned immediately, the initial value of `*counter*` is 0, and then around a second later, it gets updated to 1 by the anonymous thread.

Create a thread: print a message onto the top-level

```
;;; Print a message onto the top-level using a thread
(defun print-message-top-level-wrong ()
  (bt:make-thread
   (lambda ()
     (format *standard-output* "Hello from thread!"))
   :name "hello")
  nil)
```

And the output:

```
CL-USER> (print-message-top-level-wrong)
NIL
```

So what went wrong? The problem is variable binding. Now, the 't' parameter to the format function refers to the top-level, which is a Common Lisp term for the main console stream, also referred to by the global variable `*standard-output*`. So we could have expected the output to be shown on the main console screen.

The same code would have run fine if we had not run it in a separate thread. What happens is that each thread has its own stack where the variables are rebound. In this case, even for `*standard-output*`, which being a global variable, we would assume should be available to all threads, is rebound inside each thread! This is similar to the concept of ThreadLocal storage in Java.

Print a message onto the top-level — fixed

So how do we fix the problem of the previous example? By binding the top-level at the time of thread creation of course. Pure lexical scoping to the rescue!

```
;;; Print a message onto the top-level using a thread — fixe
(defun print-message-top-level-fixed ()
  (let ((top-level *standard-output*))
    (bt:make-thread
     (lambda ()
       (format top-level "Hello from thread!"))
     :name "hello"))
  nil)
```

Which produces:

```
CL-USER> (print-message-top-level-fixed)
Hello from thread!
NIL
```

Phew! However, there is another way of producing the same result using a very interesting reader macro as we'll see next.

Print a message onto the top-level — read-time eval macro

Let's take a look at the code first:

```
;;; Print a message onto the top-level using a thread - reac

(eval-when (:compile-toplevel)
  (defun print-message-top-level-reader-macro ()
    (bt:make-thread
     (lambda ()
       (format #.*standard-output* "Hello from thread!")))
    nil))

(print-message-top-level-reader-macro)
```

And the output:

```
CL-USER> (print-message-top-level-reader-macro)
Hello from thread!
NIL
```

So it works, but what's the deal with the `eval-when` and what is that strange `#.` symbol before `*standard-output*`?

`eval-when` controls when evaluation of Lisp expressions takes place. We can have three targets — `:compile-toplevel`, `:load-toplevel`, and `:execute`.

The `#.` symbol is what is called a "Reader macro". A reader (or read) macro is called so because it has special meaning to the Common Lisp Reader, which is the component that is responsible for reading in Common Lisp expressions and making sense out of them. This specific reader macro ensures that the binding of `*standard-output*` is done at read time.

Binding the value at read-time ensures that the original value of `*standard-output*` is maintained when the thread is run, and the output is shown on the correct top-level.

Now this is where the `eval-when` bit comes into play. By wrapping the whole function definition inside the `eval-when`, and ensuring that evaluation takes place during compile time, the correct value of `*standard-output*` is bound. If we had skipped the `eval-when`, we would see the following error:

```
error:
  don't know how to dump #<SWANK/GRAY::SLIME-OUTPUT-STREAM
  ==>
    #<SWANK/GRAY::SLIME-OUTPUT-STREAM {100439EEA3}>


note: The first argument never returns a value.
note:
  deleting unreachable code
  ==>
    "Hello from thread!"
```

```
Compilation failed.
```

And that makes sense because SBCL cannot make sense of what this output stream returns since it is a stream and not really a defined value (which is what the ‘format’ function expects). That is why we see the “unreachable code” error.

Note that if the same code had been run on the REPL directly, there would be no problem since the resolution of all the symbols would be done correctly by the REPL thread.

Modify a shared resource from multiple threads

Suppose we have the following setup with a minimal bank-account class (no error checks):

```
;;; Modify a shared resource from multiple threads

(defclass bank-account ()
  ((id :initarg :id
       :initform (error "id required")
       :accessor :id)
   (name :initarg :name
         :initform (error "name required")
         :accessor :name)
   (balance :initarg :balance
            :initform 0
            :accessor :balance)))

(defgeneric deposit (account amount)
  (:documentation "Deposit money into the account"))

(defgeneric withdraw (account amount)
  (:documentation "Withdraw amount from account"))

(defmethod deposit ((account bank-account) (amount real))
  (incf (:balance account) amount))
```

```lisp
(defmethod withdraw ((account bank-account) (amount real))
  (decf (:balance account) amount))
```

And we have a simple client which apparently does not believe in any form of synchronisation:

```lisp
(defparameter *rich*
  (make-instance 'bank-account
                 :id 1
                 :name "Rich"
                 :balance 0))
; compiling (DEFPARAMETER *RICH* ...)

(defun demo-race-condition ()
  (loop repeat 100
     do
       (bt:make-thread
        (lambda ()
          (loop repeat 10000 do (deposit *rich* 100))
          (loop repeat 10000 do (withdraw *rich* 100))))))
```

This is all we are doing – create a new bank account instance (balance 0), and then create a 100 threads, each of which simply deposits an amount of 100 10000 times, and then withdraws the same amount the same number of times. So the final result should be the same as that of the opening balance, which is 0, right? Let's check that and see.

On a sample run, we might get the following results:

```lisp
CL-USER> (:balance *rich*)
0
CL-USER> (dotimes (i 5)
           (demo-race-condition))
NIL
CL-USER> (:balance *rich*)
22844600
```

Whoa! The reason for this discrepancy is that incf and decf are not atomic operations — they consist of multiple sub-operations, and the order in which they are executed is not in our control.

This is what is called a “race condition” — multiple threads contending for the same shared resource with at least one modifying thread which, more likely than not, reads the wrong value of the object while modifying it. How do we fix it? One simple way it to use locks (mutex in this case, could be semaphores for more complex situations).

Modify a shared resource from multiple threads — fixed using locks

Let’s rest the balance for the account back to 0 first:

```
CL-USER> (setf (:balance *rich*) 0)
0
CL-USER> (:balance *rich*)
0
```

Now let’s modify the `demo-race-condition` function to access the shared resource using locks (created using `bt:make-lock` and used as shown):

```
(defvar *lock* (bt:make-lock))
; compiling (DEFVAR *LOCK* …)

(defun demo-race-condition-locks ()
  (loop repeat 100
     do
       (bt:make-thread

        (lambda ()
          (loop repeat 10000 do (bt:with-lock-held (*lock*)
                                  (deposit *rich* 100)))
          (loop repeat 10000 do (bt:with-lock-held (*lock*)
                                  (withdraw *rich* 100)))))))
; compiling (DEFUN DEMO-RACE-CONDITION-LOCKS ...)
```

And let’s do a bigger sample run this time around:

```
CL-USER> (dotimes (i 100)
           (demo-race-condition-locks))
NIL
```

```
CL-USER> (:balance *rich*)
0
```

Excellent! Now this is better. Of course, one has to remember that using a mutex like this is bound to affect performance. There is a better way in quite a few circumstances — using atomic operations when possible. We'll cover that next.

Modify a shared resource from multiple threads — using atomic operations

Atomic operations are operations that are guaranteed by the system to all occur inside a conceptual transaction, i.e., all the sub-operations of the main operation all take place together without any interference from outside. The operation succeeds completely or fails completely. There is no middle ground, and there is no inconsistent state.

Another advantage is that performance is far superior to using locks to protect access to the shared state. We will see this difference in the actual demo run.

The Bordeaux library does not provide any real support for atomics, so we will have to depend on the specific implementation support for that. In our case, that is SBCL, and so we will have to defer this demo to the SBCL section.

Joining on a thread, destroying a thread

To join on a thread, we use the `bt:join-thread` function, and for destroying a thread (not a recommended operation), we can use the `bt:destroy-thread` function.

A simple demo:

```
(defmacro until (condition &body body)
  (let ((block-name (gensym)))
    `(block ,block-name
       (loop
           (if ,condition
```

```
                (return-from ,block-name nil)
                (progn
                    ,@body))))))

(defun join-destroy-thread ()
  (let* ((s *standard-output*)
        (joiner-thread
          (bt:make-thread
            (lambda ()
              (loop for i from 1 to 10
                 do
                   (format s "~%[Joiner Thread]  Working...'
                   (sleep (* 0.01 (random 100)))))))
        (destroyer-thread
            (bt:make-thread
               (lambda ()
                (loop for i from 1 to 1000000
                   do
                     (format s "~%[Destroyer Thread] Working
                     (sleep (* 0.01 (random 10000))))))))
    (format t "~%[Main Thread] Waiting on joiner thread...")
    (bt:join-thread joiner-thread)
    (format t "~%[Main Thread] Done waiting on joiner threac
    (if (bt:thread-alive-p destroyer-thread)
        (progn
          (format t "~%[Main Thread] Destroyer thread alive.
          (bt:destroy-thread destroyer-thread))
        (format t "~%[Main Thread] Destroyer thread is alrea
    (until (bt:thread-alive-p destroyer-thread)
           (format t "[Main Thread] Waiting for destroyer th
    (format t "~%[Main Thread] Destroyer thread dead")
    (format t "~%[Main Thread] Adios!~%")))
```

And the output on a run:

```
CL-USER> (join-destroy-thread)

[Joiner Thread]  Working...
[Destroyer Thread] Working...
[Main Thread] Waiting on joiner thread...
[Joiner Thread]  Working...
```

```
[Joiner Thread]  Working...
[Joiner Thread]  Working...
[Joiner Thread]  Working...
[Joiner Thread]  Working...
[Joiner Thread]  Working...
[Joiner Thread]  Working...
[Joiner Thread]  Working...
[Joiner Thread]  Working...
[Main Thread] Done waiting on joiner thread
[Main Thread] Destroyer thread alive... killing it
[Main Thread] Destroyer thread dead
[Main Thread] Adios!
NIL
```

The until macro simply loops around until the condition becomes true. The rest of the code is pretty much self-explanatory — the main thread waits for the joiner-thread to finish, but it immediately destroys the destroyer-thread.

Again, it is not recommended to use `bt:destroy-thread`. Any conceivable situation which requires this function can probably be done better with another approach.

Now let's move onto some more comprehensive examples which tie together all the concepts discussed thus far.

Timeouts

We can use `bt:with-timeout`.

Sometimes we want to run a background operation, but we want to ensure that it doesn't take a maximum time limit. We can use `bt:with-timeout (n)` where n is a number of seconds. In case of a timeout, Bordeaux-threads signals a `bt:timeout` error.

In our scenario below, we create a thread that launches a potentially long operation, we `join` the thread with a timeout, and we handle any timeout error. In our case, we destroy the running thread. This also kills its underlying processes (were they run with `uiop:run-program`).

```
(defun maybe-costly-operation ()
```

```
  (print "working hard...")
  (sleep 10))

(let ((thread (bt:make-thread               ;; <--- create a threac
                 (lambda ()
                   ;; maybe a long operation:
                   (maybe-costly-operation))
                 :name "maybe-costly-thread")))
    (handler-case
        (bt:with-timeout (timeout)          ;; <-- with-timeout
          (bt:join-thread thread))          ;; <-- join the thread
      (bt:timeout ()                        ;; <-- handle timeout.
        (bt:destroy-thread thread))))
```

Useful functions

Here is a summary of the functions, macros and global variables which were used in the demo examples along with some extras. These should cover most of the basic programming scenarios:

- `bt:*supports-thread-p*` (to check for basic thread support)
- `bt:make-thread` (create a new thread)
- `bt:current-thread` (return the current thread object)
- `bt:all-threads` (return a list of all running threads)
- `bt:thread-alive-p` (checks if the thread is still alive)
- `bt:thread-name` (return the name of the thread)
- `bt:join-thread` (join on the supplied thread)
- `bt:interrupt-thread` (interrupt the given thread)
- `bt:destroy-thread` (attempt to abort the thread)
- `bt:make-lock` (create a mutex)
- `bt:with-lock-held` (use the supplied lock to protect critical code)
- `bt:with-timeout` (to signal a timeout error)

SBCL threads

SBCL provides support for native threads via its sb-thread package. These are very low-level functions, but we can build our own abstractions on top of these as shown in the demo examples.

You can refer to the documentation for more details (check the “Wrap-up” section).

You can see from the examples below that there is a strong correspondence between Bordeaux and SBCL Thread functions. In most cases, the only difference is the change of package name from bt to sb-thread.

It is evident that the Bordeaux thread library was more or less based on the SBCL implementation. As such, explanation will be provided only in those cases where there is a major difference in syntax or semantics.

Basics — list current thread, list all threads, get thread name

The code:

```
;;; Print the current thread, all the threads, and the curre

(defun print-thread-info ()
  (let* ((curr-thread sb-thread:*current-thread*)
         (curr-thread-name (sb-thread:thread-name curr-threa
         (all-threads (sb-thread:list-all-threads)))
    (format t "Current thread: ~a~%~%" curr-thread)
    (format t "Current thread name: ~a~%~%" curr-thread-name
    (format t "All threads:~% ~{~a~%~}~%" all-threads))
  nil)
```

And the output:

```
CL-USER> (print-thread-info)
Current thread: #<THREAD "repl-thread" RUNNING {10043B8003}>

Current thread name: repl-thread


All threads:
 #<THREAD "repl-thread" RUNNING {10043B8003}>
#<THREAD "auto-flush-thread" RUNNING {10043B7DA3}>
#<THREAD "swank-indentation-cache-thread" waiting on: #<WAIT
#<THREAD "reader-thread" RUNNING {1003A20063}>
```

```
#<THREAD "reader-thread" RUNNING {1003A20003}>
#<THREAD "control-thread" waiting on: #<WAITQUEUE  {1003A19E
#<THREAD "Swank Sentinel" waiting on: #<WAITQUEUE  {10037900
#<THREAD "main thread" RUNNING {1002991CE3}>

NIL
```

Update a global variable from a thread

The code:

```
;;; Update a global variable from a thread

(defparameter *counter* 0)

(defun test-update-global-variable ()
  (sb-thread:make-thread
   (lambda ()
     (sleep 1)
     (incf *counter*)))
  *counter*)
```

And the output:

```
CL-USER> (test-update-global-variable)
0
```

Print a message onto the top-level using a thread

The code:

```
;;; Print a message onto the top-level using a thread

(defun print-message-top-level-wrong ()
  (sb-thread:make-thread
   (lambda ()
     (format *standard-output* "Hello from thread!")))
  nil)
```

And the output:

```
CL-USER> (print-message-top-level-wrong)
NIL
```

Print a message onto the top-level — fixed:

The code:

```
;;; Print a message onto the top-level using a thread - fixe

(defun print-message-top-level-fixed ()
  (let ((top-level *standard-output*))
    (sb-thread:make-thread
     (lambda ()
       (format top-level "Hello from thread!"))))
  nil)
```

And the output:

```
CL-USER> (print-message-top-level-fixed)
Hello from thread!
NIL
```

Print a message onto the top-level — better

The code:

```
;;; Print a message onto the top-level using a thread - reac

(eval-when (:compile-toplevel)
  (defun print-message-top-level-reader-macro ()
    (sb-thread:make-thread
     (lambda ()
       (format #.*standard-output* "Hello from thread!")))
    nil))
```

And the output:

```
CL-USER> (print-message-top-level-reader-macro)
Hello from thread!
```

```
NIL
```

Modify a shared resource from multiple threads

The code:

```
;;; Modify a shared resource from multiple threads

(defclass bank-account ()
  ((id :initarg :id
       :initform (error "id required")
       :accessor :id)
   (name :initarg :name
         :initform (error "name required")
         :accessor :name)
   (balance :initarg :balance
            :initform 0
            :accessor :balance)))

(defgeneric deposit (account amount)
  (:documentation "Deposit money into the account"))

(defgeneric withdraw (account amount)
  (:documentation "Withdraw amount from account"))

(defmethod deposit ((account bank-account) (amount real))
  (incf (:balance account) amount))

(defmethod withdraw ((account bank-account) (amount real))
  (decf (:balance account) amount))

(defparameter *rich*
  (make-instance 'bank-account
                 :id 1
                 :name "Rich"
                 :balance 0))

(defun demo-race-condition ()
  (loop repeat 100
     do
```

```
          (sb-thread:make-thread
           (lambda ()
             (loop repeat 10000 do (deposit *rich* 100))
             (loop repeat 10000 do (withdraw *rich* 100))))))
```

And the output:

```
CL-USER> (:balance *rich*)
0
CL-USER> (demo-race-condition)
NIL
CL-USER> (:balance *rich*)
3987400
```

Modify a shared resource from multiple threads — fixed using locks

The code:

```
(defvar *lock* (sb-thread:make-mutex))

(defun demo-race-condition-locks ()
  (loop repeat 100
     do
       (sb-thread:make-thread
        (lambda ()
          (loop repeat 10000 do (sb-thread:with-mutex (*lock
                                   (deposit *rich* 100)))
          (loop repeat 10000 do (sb-thread:with-mutex (*lock
                                   (withdraw *rich* 100))))))
```

The only difference here is that instead of make-lock as in Bordeaux, we have `make-mutex` and that is used along with the macro `with-mutex` as shown in the example.

And the output:

```
CL-USER> (:balance *rich*)
0
CL-USER> (demo-race-condition-locks)
```

```
NIL
CL-USER> (:balance *rich*)
0
```

Modify a shared resource from multiple threads — using atomic operations

First, the code:

```
;;; Modify a shared resource from multiple threads - atomics

(defgeneric atomic-deposit (account amount)
  (:documentation "Atomic version of the deposit method"))

(defgeneric atomic-withdraw (account amount)
  (:documentation "Atomic version of the withdraw method"))

(defmethod atomic-deposit ((account bank-account) (amount re
  (sb-ext:atomic-incf (car (cons (:balance account) nil)) am

(defmethod atomic-withdraw ((account bank-account) (amount r
  (sb-ext:atomic-decf (car (cons (:balance account) nil)) am

(defun demo-race-condition-atomics ()
  (loop repeat 100
     do (sb-thread:make-thread
         (lambda ()
           (loop repeat 10000 do (atomic-deposit *rich* 100)
           (loop repeat 10000 do (atomic-withdraw *rich* 100
```

And the output:

```
CL-USER> (dotimes (i 5)
           (format t "~%Opening: ~d" (:balance *rich*))
           (demo-race-condition-atomics)
           (format t "~%Closing: ~d~%" (:balance *rich*)))

Opening: 0
Closing: 0
```

```
Opening: 0
Closing: 0

Opening: 0
Closing: 0

Opening: 0
Closing: 0

Opening: 0
Closing: 0
NIL
```

As you can see, SBCL's atomic functions are a bit quirky. The two functions used here: `sb-ext:incf` and `sb-ext:atomic-decf` have the following signatures:

Macro: atomic-incf [sb-ext] place &optional diff

and

Macro: atomic-decf [sb-ext] place &optional diff

The interesting bit is that the "place" parameter must be any of the following (as per the documentation):

- a defstruct slot with declared type (unsigned-byte 64) or aref of a (simple-array (unsigned-byte 64) (*)) The type `sb-ext:word` can be used for these purposes.
- car or cdr (respectively first or REST) of a cons.
- a variable defined using defglobal with a proclaimed type of fixnum.

This is the reason for the bizarre construct used in the `atomic-deposit` and `atomic-decf` methods.

One major incentive to use atomic operations as much as possible is performance. Let's do a quick run of the `demo-race-condition-locks` and demo-race-condition-atomics functions over 1000 times and check the difference in performance (if any):

With locks:

```
CL-USER> (time
                    (loop repeat 100
                      do (demo-race-condition-locks)))
Evaluation took:
  57.711 seconds of real time
  431.451639 seconds of total run time (408.014746 user, 23.
  747.61% CPU
  126,674,011,941 processor cycles
  3,329,504 bytes consed

NIL
```

With atomics:

```
CL-USER> (time
                    (loop repeat 100
                     do (demo-race-condition-atomics)))
Evaluation took:
  2.495 seconds of real time
  8.175454 seconds of total run time (6.124259 user, 2.05119
  [ Run times consist of 0.420 seconds GC time, and 7.756 se
  327.66% CPU
  5,477,039,706 processor cycles
  3,201,582,368 bytes consed

NIL
```

The results? The locks version took around 57s whereas the lockless atomics version took just 2s! This is a massive difference indeed!

Joining on a thread, destroying a thread example

The code:

```
;;; Joining on and destroying a thread
```

```lisp
;;; Joining on and destroying a thread

(defmacro until (condition &body body)
  (let ((block-name (gensym)))
    `(block ,block-name
       (loop
           (if ,condition
               (return-from ,block-name nil)
               (progn
                   ,@body))))))

(defun join-destroy-thread ()
  (let* ((s *standard-output*)
         (joiner-thread
            (sb-thread:make-thread
               (lambda ()
                 (loop for i from 1 to 10
                    do
                      (format s "~%[Joiner Thread]  Working...")
                      (sleep (* 0.01 (random 100)))))))
         (destroyer-thread
            (sb-thread:make-thread
               (lambda ()
                 (loop for i from 1 to 1000000
                    do
                      (format s "~%[Destroyer Thread] Working...'
                      (sleep (* 0.01 (random 10000))))))))

    (format t "~%[Main Thread] Waiting on joiner thread...")
    (bt:join-thread joiner-thread)
    (format t "~%[Main Thread] Done waiting on joiner thread")
    (if (sb-thread:thread-alive-p destroyer-thread)
        (progn
          (format t "~%[Main Thread] Destroyer thread alive... k
          (sb-thread:terminate-thread destroyer-thread))
        (format t "~%[Main Thread] Destroyer thread is already c
    (until (sb-thread:thread-alive-p destroyer-thread)
       (format t "[Main Thread] Waiting for destroyer thread to
    (format t "~%[Main Thread] Destroyer thread dead")
    (format t "~%[Main Thread] Adios!~%")))
```

And the output:

```
CL-USER> (join-destroy-thread)

[Joiner Thread]  Working...
[Destroyer Thread] Working...
[Main Thread] Waiting on joiner thread...
[Joiner Thread]  Working...
[Joiner Thread]  Working...
[Joiner Thread]  Working...
[Joiner Thread]  Working...
[Joiner Thread]  Working...
[Joiner Thread]  Working...
[Joiner Thread]  Working...
[Joiner Thread]  Working...
[Joiner Thread]  Working...
[Main Thread] Done waiting on joiner thread
[Main Thread] Destroyer thread alive... killing it
[Main Thread] Destroyer thread dead
[Main Thread] Adios!
NIL
```

Useful functions

Here is a summarised list of the functions, macros and global variables used in the examples along with some extras:

- `(member :thread-support *features*)` (check thread support)
- `sb-thread:make-thread` (create a new thread)
- `sb-thread:*current-thread*` (holds the current thread object)
- `sb-thread:list-all-threads` (return a list of all running threads)
- `sb-thread:thread-alive-p` (checks if the thread is still alive)
- `sb-thread:thread-name` (return the name of the thread)
- `sb-thread:join-thread` (join on the supplied thread)
- `sb-thread:interrupt-thread` (interrupt the given thread)
- `sb-thread:destroy-thread` (attempt to abort the thread)
- `sb-thread:make-mutex` (create a mutex)
- `sb-thread:with-mutex` (use supplied lock to protect critical code)

Wrap-up

As you can see, concurrency support is rather primitive in Common Lisp, but that's primarily due to the glaring absence of this important feature in the ANSI Common Lisp specification. That does not detract in the least from the support provided by Common Lisp implementations, nor wonderful libraries like the Bordeaux library.

You should follow up on your own by reading a lot more on this topic. I share some of my own references here:

- Common Lisp Recipes
- Bordeaux API Reference
- SBCL Manual on Threading
- The Common Lisp Hyperspec

Next up, the final post in this mini-series: parallelism in Common Lisp using the **lparallel** library.

Parallel programming with lparallel

It is important to note that lparallel also provides extensive support for asynchronous programming, and is not a purely parallel programming library. As stated before, parallelism is merely an abstract concept in which tasks are conceptually independent of one another.

The lparallel library is built on top of the Bordeaux threading library.

As mentioned previously, parallelism and concurrency can be (and usually are) implemented using the same means — threads, processes, etc. The difference between lies in their conceptual differences.

Note that not all the examples shown in this post are necessarily parallel. Asynchronous constructs such as Promises and Futures are, in particular, more suited to concurrent programming than parallel programming.

The modus operandi of using the lparallel library (for a basic use case) is as follows:

- Create an instance of what the library calls a kernel using `lparallel:make-kernel`. The kernel is the component that schedules and executes tasks.
- Design the code in terms of futures, promises and other higher level functional concepts. To this end, lparallel provides support for **channels**, **promises**, **futures**, and **cognates**.
- Perform operations using what the library calls cognates, which are simply functions which have equivalents in the Common Lisp language itself. For instance, the `lparallel:pmap` function is the parallel equivalent of the Common Lisp `map` function.
- Finally, close the kernel created in the first step using `lparallel:end-kernel`.

Note that the onus of ensuring that the tasks being carried out are logically parallelisable as well as taking care of all mutable state is on the developer.

Credit: this article first appeared on z0ltan.wordpress.com.

Installation

Let’s check if lparallel is available for download using Quicklisp:

```
CL-USER> (ql:system-apropos "lparallel")
#<SYSTEM lparallel / lparallel-20160825-git / quicklisp 2016-08-
#<SYSTEM lparallel-bench / lparallel-20160825-git / quicklisp 20
#<SYSTEM lparallel-test / lparallel-20160825-git / quicklisp 201
; No value
```

Looks like it is. Let’s go ahead and install it:

```
CL-USER> (ql:quickload "lparallel")
To load "lparallel":
  Load 2 ASDF systems:
    alexandria bordeaux-threads
  Install 1 Quicklisp release:
    lparallel
```

```
; Fetching #<URL "http://beta.quicklisp.org/archive/lparallel/20
; 76.71KB
==================================================
78,551 bytes in 0.62 seconds (124.33KB/sec)
; Loading "lparallel"
[package lparallel.util].........................
[package lparallel.thread-util]..................
[package lparallel.raw-queue]....................
[package lparallel.cons-queue]...................
[package lparallel.vector-queue].................
[package lparallel.queue]........................
[package lparallel.counter]......................
[package lparallel.spin-queue]...................
[package lparallel.kernel].......................
[package lparallel.kernel-util]..................
[package lparallel.promise]......................
[package lparallel.ptree]........................
[package lparallel.slet].........................
[package lparallel.defpun].......................
[package lparallel.cognate]......................
[package lparallel]
(:LPARALLEL)
```

And that's all it took! Now let's see how this library actually works.

Preamble - get the number of cores

First, let's get hold of the number of threads that we are going to use for our parallel examples. Ideally, we'd like to have a 1:1 match between the number of worker threads and the number of available cores.

We can use the great **Serapeum** library to this end, which has a `count-cpus` function, that works on all major platforms.

Install it:

```
CL-USER> (ql:quickload "serapeum")
```

and call it:

```
CL-USER> (serapeum:count-cpus)
8
```

and check that is correct.

Common Setup

In this example, we will go through the initial setup bit, and also show some useful information once the setup is done.

Load the library:

```
CL-USER> (ql:quickload "lparallel")
To load "lparallel":
  Load 1 ASDF system:
    lparallel
; Loading "lparallel"

(:LPARALLEL)
```

Initialise the lparallel kernel:

```
CL-USER> (setf lparallel:*kernel*
               (lparallel:make-kernel 8 :name "custom-kernel"))
#<LPARALLEL.KERNEL:KERNEL :NAME "custom-kernel" :WORKER-COUNT 8
```

Note that the `*kernel*` global variable can be rebound — this allows multiple kernels to co-exist during the same run. Now, some useful information about the kernel:

```
CL-USER> (defun show-kernel-info ()
           (let ((name (lparallel:kernel-name))
                 (count (lparallel:kernel-worker-count))
                 (context (lparallel:kernel-context))
                 (bindings (lparallel:kernel-bindings)))
             (format t "Kernel name = ~a~%" name)
             (format t "Worker threads count = ~d~%" count)
             (format t "Kernel context = ~a~%" context)
             (format t "Kernel bindings = ~a~%" bindings)))
```

```
WARNING: redefining COMMON-LISP-USER::SHOW-KERNEL-INFO in DEFUN
SHOW-KERNEL-INFO

CL-USER> (show-kernel-info)
Kernel name = custom-kernel
Worker threads count = 8
Kernel context = #<FUNCTION FUNCALL>
Kernel bindings = ((*STANDARD-OUTPUT* . #<SLIME-OUTPUT-STREAM {1
                   (*ERROR-OUTPUT* . #<SLIME-OUTPUT-STREAM {1004
NIL
```

End the kernel (this is important since `*kernel*` does not get garbage collected until we explicitly end it):

```
CL-USER> (lparallel:end-kernel :wait t)
(#<SB-THREAD:THREAD "custom--kernel" FINISHED values: NIL {10072
 #<SB-THREAD:THREAD "custom--kernel" FINISHED values: NIL {10072
 #<SB-THREAD:THREAD "custom--kernel" FINISHED values: NIL {10072
 #<SB-THREAD:THREAD "custom--kernel" FINISHED values: NIL {10072
 #<SB-THREAD:THREAD "custom--kernel" FINISHED values: NIL {10072
 #<SB-THREAD:THREAD "custom--kernel" FINISHED values: NIL {10072
 #<SB-THREAD:THREAD "custom--kernel" FINISHED values: NIL {10072
 #<SB-THREAD:THREAD "custom--kernel" FINISHED values: NIL {10072
```

Let’s move on to some more examples of different aspects of the lparallel library.

For these demos, we will be using the following initial setup from a coding perspective:

```
(require ‘lparallel)
(require ‘bt-semaphore)

(defpackage :lparallel-user
  (:use :cl :lparallel :lparallel.queue :bt-semaphore))

(in-package :lparallel-user)
```

```
;;; initialise the kernel
(defun init ()
  (setf *kernel* (make-kernel 8 :name "channel-queue-kernel")))

(init)
```

So we will be using a kernel with 8 worker threads (one for each CPU core on the machine).

And once we're done will all the examples, the following code will be run to close the kernel and free all used system resources:

```
;;; shut the kernel down
(defun shutdown ()
  (end-kernel :wait t))

(shutdown)
```

Using channels and queues

First some definitions are in order.

A **task** is a job that is submitted to the kernel. It is simply a function object along with its arguments.

A **channel** in lparallel is similar to the same concept in Go. A channel is simply a means of communication with a worker thread. In our case, it is one particular way of submitting tasks to the kernel.

A channel is created in lparallel using `lparallel:make-channel`. A task is submitted using `lparallel:submit-task`, and the results received via `lparallel:receive-result`.

For instance, we can calculate the square of a number as:

```
(defun calculate-square (n)
  (let* ((channel (lparallel:make-channel))
         (res nil))
```

```
    (lparallel:submit-task channel (lambda (x)
                                        (* x x))
                           n)
    (setf res (lparallel:receive-result channel))
    (format t "Square of ~d = ~d~%" n res)))
```

And the output:

```
LPARALLEL-USER> (calculate-square 100)
Square of 100 = 10000
NIL
```

Now let’s try submitting multiple tasks to the same channel. In this simple example, we are simply creating three tasks that square, triple, and quadruple the supplied input respectively.

Note that in case of multiple tasks, the output will be in non-deterministic order:

```
(defun test-basic-channel-multiple-tasks ()
  (let ((channel (make-channel))
        (res '()))
    (submit-task channel (lambda (x)
                              (* x x))
                 10)
    (submit-task channel (lambda (y)
                              (* y y y))
                 10)
    (submit-task channel (lambda (z)
                              (* z z z z))
                 10)
     (dotimes (i 3 res)
       (push (receive-result channel) res))))
```

And the output:

```
LPARALLEL-USER> (dotimes (i 3)
                               (print (test-basic-channel-multipl

(100 1000 10000)
(100 1000 10000)
(10000 1000 100)
```

```
NIL
```

lparallel also provides support for creating a blocking queue in order to enable message passing between worker threads. A queue is created using `lparallel.queue:make-queue`.

Some useful functions for using queues are:

- `lparallel.queue:make-queue`: create a FIFO blocking queue
- `lparallel.queue:push-queue`: insert an element into the queue
- `lparallel.queue:pop-queue`: pop an item from the queue
- `lparallel.queue:peek-queue`: inspect value without popping it
- `lparallel.queue:queue-count`: the number of entries in the queue
- `lparallel.queue:queue-full-p`: check if the queue is full
- `lparallel.queue:queue-empty-p:`check if the queue is empty
- `lparallel.queue:with-locked-queue`: lock the queue during access

A basic demo showing basic queue properties:

```
(defun test-queue-properties ()
  (let ((queue (make-queue :fixed-capacity 5)))
    (loop
       when (queue-full-p queue)
       do (return)
       do (push-queue (random 100) queue))
     (print (queue-full-p queue))
    (loop
       when (queue-empty-p queue)
       do (return)
       do (print (pop-queue queue)))
    (print (queue-empty-p queue)))
  nil)
```

Which produces:

```
LPARALLEL-USER> (test-queue-properties)

T
17
51
```

```
55
42
82
T
NIL
```

Note: `lparallel.queue:make-queue` is a generic interface which is actually backed by different types of queues. For instance, in the previous example, the actual type of the queue is `lparallel.vector-queue` since we specified it to be of fixed size using the `:fixed-capacity` keyword argument.

The documentation doesn't actually specify what keyword arguments we can pass to `lparallel.queue:make-queue`, so let's and find that out in a different way:

```
LPARALLEL-USER> (describe 'lparallel.queue:make-queue)
LPARALLEL.QUEUE:MAKE-QUEUE
  [symbol]

MAKE-QUEUE names a compiled function:
  Lambda-list: (&REST ARGS)
  Derived type: FUNCTION
  Documentation:
    Create a queue.

    The queue contents may be initialized with the keyword a
    `initial-contents'.

    By default there is no limit on the queue capacity. Pass
    `fixed-capacity' keyword argument limits the capacity to
    passed. `push-queue' will block for a full fixed-capacit
  Source file: /Users/z0ltan/quicklisp/dists/quicklisp/softw

MAKE-QUEUE has a compiler-macro:
  Source file: /Users/z0ltan/quicklisp/dists/quicklisp/softw
; No value
```

So, as we can see, it supports the following keyword arguments: *:fixed-capacity*, and *initial-contents*.

Now, if we do specify `:fixed-capacity`, then the actual type of the queue will be `lparallel.vector-queue`, and if we skip that keyword argument, the queue will be of type `lparallel.cons-queue` (which is a queue of unlimited size), as can be seen from the output of the following snippet:

```
(defun check-queue-types ()
  (let ((queue-one (make-queue :fixed-capacity 5))
        (queue-two (make-queue)))
    (format t "queue-one is of type: ~a~%" (type-of queue-or
    (format t "queue-two is of type: ~a~%" (type-of queue-tw



LPARALLEL-USER> (check-queue-types)

queue-one is of type: VECTOR-QUEUE
queue-two is of type: CONS-QUEUE
NIL
```

Of course, you can always create instances of the specific queue types yourself, but it is always better, when you can, to stick to the generic interface and letting the library create the proper type of queue for you.

Now, let's just see the queue in action!

```
(defun test-basic-queue ()
  (let ((queue (make-queue))
        (channel (make-channel))
        (res '()))
    (submit-task channel (lambda ()
                         (loop for entry = (pop-queue queue)
                            when (queue-empty-p queue)
                            do (return)
                            do (push (* entry entry) res))))
    (dotimes (i 100)
      (push-queue i queue))
    (receive-result channel)
    (format t "~{~d ~}~%" res)))
```

Here we submit a single task that repeatedly scans the queue till it's empty, pops the available values, and pushes them into the res list.

And the output:

```
LPARALLEL-USER> (test-basic-queue)
9604 9409 9216 9025 8836 8649 8464 8281 8100 7921 7744 7569
NIL
```

Killing tasks

A small note mentioning the `lparallel:kill-task` function would be apropos at this juncture. This function is useful in those cases when tasks are unresponsive. The lparallel documentation clearly states that this must only be used as a last resort.

All tasks which are created are by default assigned a category of :default. The dynamic property, `*task-category*` holds this value, and can be dynamically bound to different values (as we shall see).

```
;;; kill default tasks
(defun test-kill-all-tasks ()
  (let ((channel (make-channel))
        (stream *query-io*))
    (dotimes (i 10)
      (submit-task
          channel
          (lambda (x)
            (sleep (random 10))
            (format stream "~d~%" (* x x))) (random 10)))
    (sleep (random 2))
    (kill-tasks :default)))
```

Sample run:

```
LPARALLEL-USER> (test-kill-all-tasks)
16
1
8
WARNING: lparallel: Replacing lost or dead worker.
WARNING: lparallel: Replacing lost or dead worker.
WARNING: lparallel: Replacing lost or dead worker.
WARNING: lparallel: Replacing lost or dead worker.
```

```
WARNING: lparallel: Replacing lost or dead worker.
WARNING: lparallel: Replacing lost or dead worker.
WARNING: lparallel: Replacing lost or dead worker.
WARNING: lparallel: Replacing lost or dead worker.
```

Since we had created 10 tasks, all the 8 kernel worker threads were presumably busy with a task each. When we killed tasks of category :default, all these threads were killed as well and had to be regenerated (which is an expensive operation). This is part of the reason why `lparallel:kill-tasks` must be avoided.

Now, in the example above, all running tasks were killed since all of them belonged to the :default category. Suppose we wish to kill only specific tasks, we can do that by binding `*task-category*` when we create those tasks, and then specifying the category when we invoke `lparallel:kill-tasks`.

For example, suppose we have two categories of tasks – tasks which square their arguments, and tasks which cube theirs. Let’s assign them categories ’squaring-tasks and ’cubing-tasks respectively. Let’s then kill tasks of a randomly chosen category ’squaring-tasks or ’cubing-tasks.

Here is the code:

```
;;; kill tasks of a randomly chosen category
(defun test-kill-random-tasks ()
  (let ((channel (make-channel))
        (stream *query-io*))
    (let ((*task-category* 'squaring-tasks))
      (dotimes (i 5)
        (submit-task channel
                     (lambda (x)
                       (sleep (random 5))
                       (format stream "~%[Squaring] ~d = ~d"
                               x (* x x))) i)))
    (let ((*task-category* 'cubing-tasks))
      (dotimes (i 5)
        (submit-task channel
                    (lambda (x)
                       (sleep (random 5))
```

```
                         (format stream "~%[Cubing] ~d = ~d"
                                 x (* x x x))) i)))
    (sleep 1)
    (if (evenp (random 10))
        (progn
          (print "Killing squaring tasks")
          (kill-tasks 'squaring-tasks))
        (progn
          (print "Killing cubing tasks")
          (kill-tasks 'cubing-tasks)))))
```

And here is a sample run:

```
LPARALLEL-USER> (test-kill-random-tasks)

[Cubing] 2 = 8
[Squaring] 4 = 16
[Cubing] 4
 = [Cubing] 643 = 27
"Killing squaring tasks"
4
WARNING: lparallel: Replacing lost or dead worker.
WARNING: lparallel: Replacing lost or dead worker.
WARNING: lparallel: Replacing lost or dead worker.
WARNING: lparallel: Replacing lost or dead worker.

[Cubing] 1 = 1
[Cubing] 0 = 0

LPARALLEL-USER> (test-kill-random-tasks)

[Squaring] 1 = 1
[Squaring] 3 = 9
"Killing cubing tasks"
5
WARNING: lparallel: Replacing lost or dead worker.
WARNING: lparallel: Replacing lost or dead worker.
WARNING: lparallel: Replacing lost or dead worker.

[Squaring] 2 = 4
WARNING: lparallel: Replacing lost or dead worker.
WARNING: lparallel: Replacing lost or dead worker.
```

```
[Squaring] 0 = 0
[Squaring] 4 = 16
```

Using promises and futures

Promises and Futures provide support for Asynchronous Programming.

In lparallel-speak, a `lparallel:promise` is a placeholder for a result which is fulfilled by providing it with a value. The promise object itself is created using `lparallel:promise`, and the promise is given a value using the `lparallel:fulfill` macro.

To check whether the promise has been fulfilled yet or not, we can use the `lparallel:fulfilledp` predicate function. Finally, the `lparallel:force` function is used to extract the value out of the promise. Note that this function blocks until the operation is complete.

Let's solidify these concepts with a very simple example first:

```
(defun test-promise ()
  (let ((p (promise)))
    (loop
       do (if (evenp (read))
              (progn
                (fulfill p 'even-received!)
                (return))))
    (force p)))
```

Which generates the output:

```
LPARALLEL-USER> (test-promise)
5
1
3
10
EVEN-RECEIVED!
```

Explanation: This simple example simply keeps looping forever until an even number has been entered. The promise is fulfilled inside the loop using `lparallel:fulfill`, and the value is then returned from the function by forcing it with `lparallel:force`.

Now, let's take a bigger example. Assuming that we don't want to have to wait for the promise to be fulfilled, and instead have the current do some useful work, we can delegate the promise fulfillment to external explicitly as seen in the next example.

Consider we have a function that squares its argument. And, for the sake of argument, it consumes a lot of time doing so. From our client code, we want to invoke it, and wait till the squared value is available.

```
(defun promise-with-threads ()
  (let ((p (promise))
        (stream *query-io*)
        (n (progn
             (princ "Enter a number: ")
             (read))))
    (format t "In main function...~%")
    (bt:make-thread
     (lambda ()
         (sleep (random 10))
         (format stream "Inside thread... fulfilling promise~%")
         (fulfill p (* n n))))
    (bt:make-thread
     (lambda ()
         (loop
            when (fulfilledp p)
            do (return)
            do (progn
                 (format stream "~d~%" (random 100))
                 (sleep (* 0.01 (random 100)))))))

    (format t "Inside main function, received value: ~d~%"
            (force p))))
```

And the output:

```
LPARALLEL-USER> (promise-with-threads)
Enter a number: 19
In main function...
```

```
44
59
90
34
30
76
Inside thread... fulfilling promise
Inside main function, received value: 361
NIL
```

Explanation: There is nothing much in this example. We create a promise object p, and we spawn off a thread that sleeps for some random time and then fulfills the promise by giving it a value.

Meanwhile, in the main thread, we spawn off another thread that keeps checking if the promise has been fulfilled or not. If not, it prints some random number and continues checking. Once the promise has been fulfilled, we can extract the value using `lparallel:force` in the main thread as shown.

This shows that promises can be fulfilled by different threads while the code that created the promise need not wait for the promise to be fulfilled. This is especially important since, as mentioned before, `lparallel:force` is a blocking call. We want to delay forcing the promise until the value is actually available.

Another point to note when using promises is that once a promise has been fulfilled, invoking force on the same object will always return the same value. That is to say, a promise can be successfully fulfilled only once.

For instance:

```
(defun multiple-fulfilling ()
  (let ((p (promise)))
    (dotimes (i 10)
      (fulfill p (random 100))
      (format t "~d~%" (force p)))))
```

Which produces:

```
LPARALLEL-USER> (multiple-fulfilling)
15
15
15
15
15
15
15
15
15
15
NIL
```

So how does a future differ from a promise?

A `lparallel:future` is simply a promise that is run in parallel, and as such, it does not block the main thread like a default use of `lparallel:promise` would. It is executed in its own thread (by the lparallel library, of course).

Here is a simple example of a future:

```
(defun test-future ()
  (let ((f (future
             (sleep (random 5))
             (print "Hello from future!"))))
    (loop
       when (fulfilledp f)
       do (return)
       do (sleep (* 0.01 (random 100)))
         (format t "~d~%" (random 100)))
    (format t "~d~%" (force f))))
```

And the output:

```
LPARALLEL-USER> (test-future)
5
19
91
11
Hello from future!
NIL
```

Explanation: This exactly is similar to the `promise-with-threads` example. Observe two differences, however - first of all, the `lparallel:future` macro has a body as well. This allows the future to fulfill itself! What this means is that as soon as the body of the future is done executing, `lparallel:fulfilledp` will always return true for the future object.

Secondly, the future itself is spawned off on a separate thread by the library, so it does not interfere with the execution of the current thread very much unlike promises as could be seen in the promise-with-threads example (which needed an explicit thread for the fulfilling code in order to avoid blocking the current thread).

The most interesting bit is that (even in terms of the actual theory propounded by Dan Friedman and others), a Future is conceptually something that fulfills a Promise. That is to say, a promise is a contract that some value will be generated sometime in the future, and a future is precisely that “something” that does that job.

What this means is that even when using the lparallel library, the basic use of a future would be to fulfill a promise. This means that hacks like promise-with-threads need not be made by the user.

Let’s take a small example to demonstrate this point (a pretty contrived example, I must admit!).

Here’s the scenario: we want to read in a number and calculate its square. So we offload this work to another function, and continue with our own work. When the result is ready, we want it to be printed on the console without any intervention from us.

Here’s how the code looks:

```
;;; Callback example using promises and futures
(defun callback-promise-future-demo ()
  (let* ((p (promise))
         (stream *query-io*)
         (n (progn
              (princ "Enter a number: ")
              (read)))
```

```
        (f (future
             (sleep (random 10))
             (fulfill p (* n n))
             (force (future
                      (format stream "Square of ~d = ~d~%"
                              n (force p)))))))
    (loop
      when (fulfilledp f)
      do (return)
      do (sleep (* 0.01 (random 100))))))
```

And the output:

```
LPARALLEL-USER> (callback-promise-future-demo)
Enter a number: 19
Square of 19 = 361
NIL
```

Explanation: All right, so first off, we create a promise to hold the squared value when it is generated. This is the p object. The input value is stored in the local variable n.

Then we create a future object f. This future simply squares the input value and fulfills the promise with this value. Finally, since we want to print the output in its own time, we force an anonymous future which simply prints the output string as shown.

Note that this is very similar to the situation in an environment like Node, where we pass callback functions to other functions with the understanding that the callback will be called when the invoked function is done with its work.

Finally note that the following snippet is still fine (even if it uses the blocking `lparallel:force` call because it's on a separate thread):

```
(force (future
(format stream "Square of ~d = ~d~%" n (force p))))
```

To summarise, the general idiom of usage is: **define objects which will hold the results of asynchronous computations in promises, and use**

futures to fulfill those promises.

Using cognates - parallel equivalents of Common Lisp counterparts

Cognates are arguably the raison d'etre of the lparallel library. These constructs are what truly provide parallelism in the lparallel. Note, however, that most (if not all) of these constructs are built on top of futures and promises.

To put it in a nutshell, cognates are simply functions that are intended to be the parallel equivalents of their Common Lisp counterparts. However, there are a few extra lparallel cognates that have no Common Lisp equivalents.

At this juncture, it is important to know that cognates come in two basic flavours:

- Constructs for fine-grained parallelism: `defpun`, `plet`, `plet-if`, etc.
- Explicit functions and macros for performing parallel operations - `pmap`, `preduce`, `psort`, `pdotimes`, etc.

In the first case we don't have much explicit control over the operations themselves. We mostly rely on the fact that the library itself will optimise and parallelise the forms to whatever extent it can. In this post, we will focus on the second category of cognates.

Take, for instance, the cognate function `lparallel:pmap` is exactly the same as the Common Lisp equivalent, `map`, but it runs in parallel. Let's demonstrate that through an example.

Suppose we had a list of random strings of length varying from 3 to 10, and we wished to collect their lengths in a vector.

Let's first set up the helper functions that will generate the random strings:

```
(defvar *chars*
  (remove-duplicates
   (sort
    (loop for c across "The quick brown fox jumps over the lazy
```

```
      when (alpha-char-p c)
      collect (char-downcase c))
   #'char<)))

(defun get-random-strings (&optional (count 100000))

  "generate random strings between lengths 3 and 10"
  (loop repeat count
     collect
       (concatenate 'string  (loop repeat (+ 3 (random 8))
                                collect (nth (random 26) *chars*)))))
```

And here's how the Common Lisp map version of the solution might look like:

```
;;; map demo
(defun test-map ()
  (map 'vector #'length (get-random-strings 100)))
```

And let's have a test run:

```
LPARALLEL-USER> (test-map)
#(7 5 10 8 7 5 3 4 4 10)
```

And here's the `lparallel:pmap` equivalent:

```
;;;pmap demo
(defun test-pmap ()
  (pmap 'vector #'length (get-random-strings 100)))
```

which produces:

```
LPARALLEL-USER> (test-pmap)
#(8 7 6 7 6 4 5 6 5 7)
LPARALLEL-USER>
```

As you can see from the definitions of test-map and test-pmap, the syntax of the `lparallel:map` and `lparallel:pmap` functions are exactly the same (well, almost - `lparallel:pmap` has a few more optional arguments).

Some useful cognate functions and macros (all of them are functions except when marked so explicitly. Note that there are quite a few cognates, and I have chosen a few to try and represent every category through an example:

lparallel:pmap: parallel version of map.

Note that all the mapping functions (`lparallel:pmap`, **lparallel:pmapc**,`lparallel:pmapcar`, etc.) take two special keyword arguments:

- `:size`, specifying the number of elements of the input sequence(s) to process.
- `:parts` which specifies the number of parallel parts to divide the sequence(s) into.

```
;;; pmap - function
(defun test-pmap ()
  (let ((numbers (loop for i below 10
                       collect i)))
    (pmap 'vector (lambda (x)
                    (* x x))
          :parts (length numbers)
          numbers)))
```

Sample run:

```
LPARALLEL-USER> (test-pmap)

#(0 1 4 9 16 25 36 49 64 81)
```

lparallel:por: parallel version of or.

The behaviour is that it returns the first non-nil element amongst its arguments. However, due to the parallel nature of this macro, that element varies.

```
;;; por - macro
(defun test-por ()
  (let ((a 100)
```

```
            (b 200)
            (c nil)
            (d 300))
        (por a b c d)))
```

Sample run:

```
LPARALLEL-USER> (dotimes (i 10)
                  (print (test-por)))

300
300
100
100
100
300
100
100
100
100
NIL
```

In the case of the normal or operator, it would always have returned the first non-nil element viz. 100.

lparallel:pdotimes: parallel version of dotimes.

Note that this macro also take an optional `:parts` argument.

```
;;; pdotimes - macro
(defun test-pdotimes ()
  (pdotimes (i 5)
    (declare (ignore i))
    (print (random 100))))
```

Sample run:

```
LPARALLEL-USER> (test-pdotimes)

39
29
```

```
81
42
56
NIL
```

lparallel:pfuncall: parallel version of funcall.

```
;;; pfuncall - macro
(defun test-pfuncall ()
  (pfuncall #'* 1 2 3 4 5))
```

Sample run:

```
LPARALLEL-USER> (test-pfuncall)

120
```

lparallel:preduce: parallel version of reduce.

This very important function also takes two optional keyword arguments: `:parts` (same meaning as explained), and `:recurse`. If `:recurse` is non-nil, it recursively applies `lparallel:preduce` to its arguments, otherwise it default to using reduce.

```
;;; preduce - function
(defun test-preduce ()
  (let ((numbers (loop for i from 1 to 100
                    collect i)))
    (preduce #'+
             numbers
             :parts (length numbers)
             :recurse t)))
```

Sample run:

```
LPARALLEL-USER> (test-preduce)

5050
```

lparallel:premove-if-not: parallel version of remove-if-not.

This is essentially equivalent to “filter” in Functional Programming parlance.

```
;;; premove-if-not
(defun test-premove-if-not ()
  (let ((numbers (loop for i from 1 to 100
                    collect i)))
    (premove-if-not #'evenp numbers)))
```

Sample run:

```
LPARALLEL-USER> (test-premove-if-not)

(2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42
 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94
```

lparallel:pevery: parallel version of every.

```
;;; pevery - function
(defun test-pevery ()
  (let ((numbers (loop for i from 1 to 100
                    collect i)))
    (list (pevery #'evenp numbers)
          (pevery #'integerp numbers))))
```

Sample run:

```
LPARALLEL-USER> (test-pevery)

(NIL T)
```

In this example, we are performing two checks - firstly, whether all the numbers in the range [1,100] are even, and secondly, whether all the numbers in the same range are integers.

lparallel:count: parallel version of count.

```
;;; pcount - function
(defun test-pcount ()
  (let ((chars "The quick brown fox jumps over the lazy dog'
    (pcount #\e chars)))
```

Sample run:

```
LPARALLEL-USER> (test-pcount)

3
```

lparallel:psort: parallel version of sort.

```
;;; psort - function
(defstruct person
  name
  age)

(defun test-psort ()
  (let* ((names (list "Peter" "Sybil" "Basil" "Candy" "Olga'
         (people (loop for name in names
                    collect (make-person :name name
                                          :age (+ (random 20)
                                                  20)))))
    (print "Before sorting...")
    (print people)
    (fresh-line)
    (print "After sorting...")
    (psort
     people
     (lambda (x y)
         (< (person-age x)
            (person-age y)))
     :test #'=)))
```

Sample run:

```
LPARALLEL-USER> (test-psort)

"Before sorting..."
```

```
(#S(PERSON :NAME "Peter" :AGE 24) #S(PERSON :NAME "Sybil" :A
 #S(PERSON :NAME "Basil" :AGE 22) #S(PERSON :NAME "Candy" :A
 #S(PERSON :NAME "Olga" :AGE 33))

"After sorting..."
(#S(PERSON :NAME "Sybil" :AGE 20) #S(PERSON :NAME "Basil" :A
 #S(PERSON :NAME "Candy" :AGE 23) #S(PERSON :NAME "Peter" :A
 #S(PERSON :NAME "Olga" :AGE 33))
```

In this example, we first define a structure of type person for storing information about people. Then we create a list of 7 people with randomly generated ages (between 20 and 39). Finally, we sort them by age in non-decreasing order.

Error handling

To see how lparallel handles error handling (hint: with `lparallel:task-handler-bind`), please read lparallel-error-handling.

Monitoring and controlling threads with Slime

M-x slime-list-threads (you can also access it through the *slime-selector*, shortcut **t**) will list running threads by their names, and their statuses.

The thread on the current line can be killed with **k**, or if there's a lot of threads to kill, several lines can be selected and **k** will kill all the threads in the selected region.

g will update the thread list, but when you have a lot of threads starting and stopping it may be too cumbersome to always press **g**, so there's a variable `slime-threads-update-interval`, when set to a number X the thread list will be automatically updated each X seconds, a reasonable value would be 0.5.

Thanks to Slime tips.

References

There are, of course, a lot more functions, objects, and idiomatic ways of performing parallel computations using the lparallel library. This post barely scratches the surface on those. However, the general flow of operation is amply demonstrated here, and for further reading, you may find the following resources useful:

- The official homepage of the lparallel library, including documentation
- The Common Lisp Hyperspec, and, of course
- Your Common Lisp implementation's manual. For SBCL, here is a link to the official manual
- Common Lisp recipes by the venerable Edi Weitz.
- more concurrency and threading libraries on the Awesome-cl#parallelism-and-concurrency list.

Defining Systems

A **system** is a collection of Lisp files that together constitute an application or a library, and that should therefore be managed as a whole. A **system definition** describes which source files make up the system, what the dependencies among them are, and the order they should be compiled and loaded in.

ASDF

ASDF is the standard build system for Common Lisp. It is shipped in most Common Lisp implementations. It includes UIOP, *"the Utilities for Implementation- and OS- Portability"*. You can read its manual and the tutorial and best practices.

Simple examples

Loading a system definition

When you start your Lisp, it knows about its internal modules and, by default, it has no way to know that your shiny new project is located under your `~/code/foo/bar/new-ideas/` directory. So, in order to load your project in your image, you have one of three ways:

- use ASDF or Quicklisp defaults
- configure where ASDF or Quicklisp look for project definitions
- load your project definition explicitely.

Please read our section on the getting started#how-to-load-an-existing-project page.

Loading a system

Once your Lisp knows what your system is and where it lives, you can load it.

The most trivial use of ASDF is by calling `asdf:load-system` to load your library. Then you can use it. For instance, if it exports a function `some-fun` in its package `foobar`, then you will be able to call it with `(foobar:some-fun ...)` or with:

```
(in-package :foobar)
(some-fun ...)
```

You can also use Quicklisp.

Quicklisp calls ASDF under the hood, with the advantage that it will download and install any dependency if they are not already installed.

```
(ql:quickload "foobar")
;; =>
;; installs all dependencies
;; and loads the system.
```

Also, you can use SLIME to load a system, using the `M-x slime-load-system` Emacs command or the `, load-system` comma command in the prompt. The interesting thing about this way of doing it is that SLIME collects all the system warnings and errors in the process, and puts them in the `*slime-compilation*` buffer, from which you can interactively inspect them after the loading finishes.

Testing a system

To run the tests for a system, you may use:

```
(asdf:test-system :foobar)
```

The convention is that an error SHOULD be signalled if tests are unsuccessful.

Designating a system

The proper way to designate a system in a program is with lower-case strings, not symbols, as in:

```
(asdf:load-system "foobar")
(asdf:test-system "foobar")
```

How to write a trivial system definition

A trivial system would have a single Lisp file called `foobar.lisp`, located at the project's root. That file would depend on some existing libraries, say `alexandria` for general purpose utilities, and `trivia` for pattern-matching. To make this system buildable using ASDF, you create a system definition file called `foobar.asd`, with the following contents:

```
(asdf:defsystem "foobar"
  :depends-on ("alexandria" "trivia")
  :components ((:file "foobar")))
```

Note how the type `lisp` of `foobar.lisp` is implicit in the name of the file above. As for contents of that file, they would look like this:

```
(defpackage :foobar
  (:use :common-lisp :alexandria :trivia)
  (:export
   #:some-function
   #:another-function
   #:call-with-foobar
   #:with-foobar))

(in-package :foobar)

(defun some-function (...)
    ...)
...
```

Instead of `using` multiple complete packages, you might want to just import parts of them:

```
(defpackage :foobar
  (:use #:common-lisp)
```

```
  (:import-from #:alexandria
                #:some-function
                #:another-function))
  (:import-from #:trivia
                #:some-function
                #:another-function))
...)
```

Using the system you defined

Assuming your system is installed under `~/common-lisp/`, `~/quicklisp/local-projects/` or some other filesystem hierarchy already configured for ASDF, you can load it with: `(asdf:load-system "foobar")`.

If your Lisp was already started when you created that file, you may have to, either:

- load the new .asd file: `(asdf:load-asd "path/to/foobar.asd")`, or with `C-c C-k` in Slime to compile and load the whole file.
  - note: avoid using the built-in `load` for ASDF files, it may work but `asdf:load-asd` is preferred.
- `(asdf:clear-configuration)` to re-process the configuration.

How to write a trivial testing definition

Even the most trivial of systems needs some tests, if only because it will have to be modified eventually, and you want to make sure those modifications don't break client code. Tests are also a good way to document expected behavior.

The simplest way to write tests is to have a file `foobar-tests.lisp` and modify the above `foobar.asd` as follows:

```
(asdf:defsystem "foobar"
    :depends-on ("alexandria" "trivia")
    :components ((:file "foobar"))
    :in-order-to ((test-op (test-op "foobar/tests"))))
```

```
(asdf:defsystem "foobar/tests"
    :depends-on ("foobar" "fiveam")
    :components ((:file "foobar-tests"))
    :perform (test-op (o c) (symbol-call :fiveam '#:run! :foobar
```

The `:in-order-to` clause in the first system allows you to use `(asdf:test-system :foobar)` which will chain into `foobar/tests`. The `:perform` clause in the second system does the testing itself.

In the test system, `fiveam` is the name of a popular test library, and the content of the `perform` method is how to invoke this library to run the test suite `:foobar`. Obvious YMMV if you use a different library.

Create a project skeleton

cl-project can be used to generate a project skeleton. It will create a default ASDF definition, generate a system for unit testing, etc.

Install with

(ql:quickload "cl-project")

Create a project:

```
(cl-project:make-project #p"lib/cl-sample/"
:author "Eitaro Fukamachi"
:email "e.arrows@gmail.com"
:license "LLGPL"
:depends-on '(:clack :cl-annot))
;-> writing /Users/fukamachi/Programs/lib/cl-sample/.gitignore
;   writing /Users/fukamachi/Programs/lib/cl-sample/README.markd
;   writing /Users/fukamachi/Programs/lib/cl-sample/cl-sample-te
;   writing /Users/fukamachi/Programs/lib/cl-sample/cl-sample.as
;   writing /Users/fukamachi/Programs/lib/cl-sample/src/hogehoge
;   writing /Users/fukamachi/Programs/lib/cl-sample/t/hogehoge.l
;=> T
```

And you're done.

Debugging

You entered this new world of Lisp and now wonder: how can we debug what's going on? How is it more interactive than other platforms? What does the interactive debugger bring, apart from stack traces?

Print debugging

Well of course we can use the famous technique of "print debugging". Let's just recap a few print functions.

`print` works, it prints a `read`able representation of its argument, which means what is `print`ed can be `read` back in by the Lisp reader. It accepts only one argument.

`princ` focuses on an *aesthetic* representation.

`(format t "~a" …)`, with the *aesthetic* directive, prints a string (in `t`, the standard output stream) and returns nil, whereas `format nil …` doesn't print anything and returns a string. With many format controls we can print several variables at once.

`print` has this useful debugging feature that it prints *and* returns the result form it was given as argument. You can intersperse `print` statements in the middle of your algorithm, it won't break it.

```
(+ 2 (print 40))
```

Logging

Logging is already a good evolution from print debugging ;)

[log4cl] is the popular, de-facto logging library although it isn't the only one. Download it:

```
(ql:quickload "log4cl")
```

and let's have a dummy variable:

```lisp
(defvar *foo* '(:a :b :c))
```

We can use log4cl with its `log` nickname, then it is as simple to use as:

```lisp
(log:info *foo*)
;; <INFO> [13:36:49] cl-user () - *FOO*: (:A :B :C)
```

We can interleave strings and expressions, with or without `format` control strings:

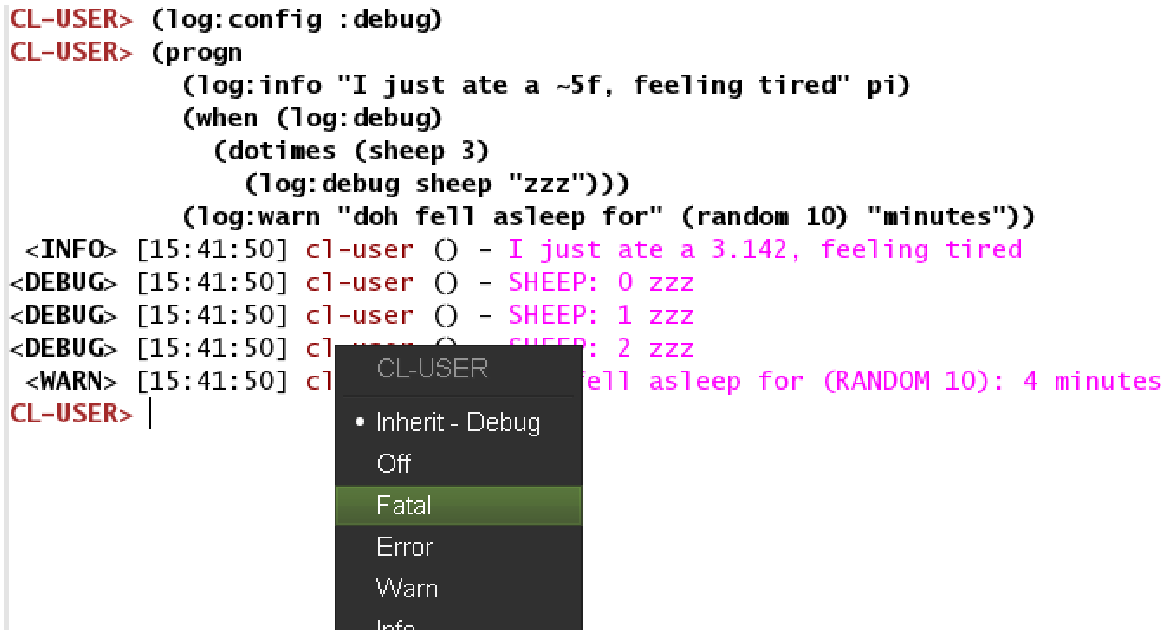
```lisp
(log:info "foo is " *foo*)
;; <INFO> [13:37:22] cl-user () - foo is *FOO*: (:A :B :C)
(log:info "foo is ~{~a~}" *foo*)
;; <INFO> [13:39:05] cl-user () - foo is ABC
```

With its companion library `log4slime`, we can interactively change the log level:

- globally
- per package
- per function
- and by CLOS methods and CLOS hierarchy (before and after methods)

It is very handy, when we have a lot of output, to turn off the logging of functions or packages we know to work, and thus narrowing our search to the right area. We can even save this configuration and re-use it in another image, be it on another machine.

We can do all this through commands, keyboard shortcuts and also through a menu or mouse clicks.



“changing the log level with log4slime”

We invite you to read log4cl’s README.

Using the powerful REPL

Part of the joy of Lisp is the excellent REPL. Its existence usually delays the need to use other debugging tools, if it doesn’t annihilate them for the usual routine.

As soon as we define a function, we can try it in the REPL. In Slime, compile a function with `C-c C-c` (the whole buffer with `C-c C-k`), switch to the REPL with `C-c C-z` and try it. Eventually enter the package you are working on with `(in-package :your-package)` or `C-c ~` `(slime-sync-package-and-default-directory`, which will also change the default working directory to the package definition’s directory).

The feedback is immediate. There is no need to recompile everything, nor to restart any process, nor to create a main function and define command line arguments for use in the shell (which we can of course do later on when needed).

We usually need to create some data to test our function(s). This is a subsequent art of the REPL existence and it may be a new discipline for newcomers. A trick is to write the test data alongside your functions but below a `#+nil` feature test (or safer, `+(or)`: it is still possible that someone pushed `NIL` to the `*features*` list) so that only you can manually compile them:

```
#+nil
(progn
   (defvar *test-data* nil)
   (setf *test-data* (make-instance 'foo …)))
```

When you load this file, `*test-data*` won't exist, but you can manually create it with `C-c C-c`.

We can define tests functions like this.

Some do similarly inside `#| … |#` comments.

All that being said, keep in mind to write unit tests when time comes ;)

Inspect and describe

These two commands share the same goal, printing a description of an object, `inspect` being the interactive one.

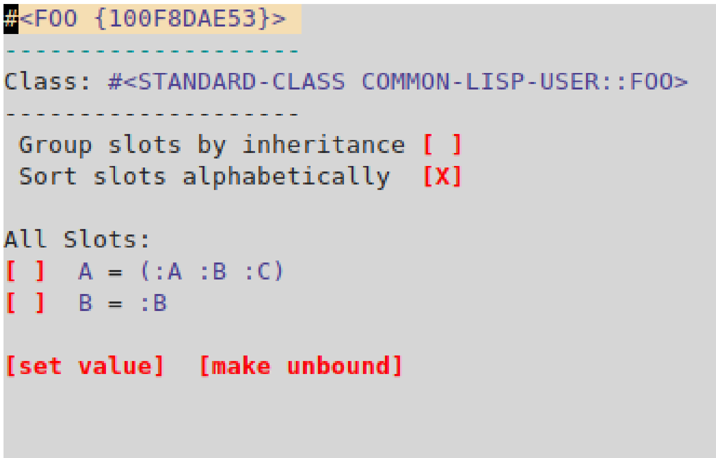
```
(inspect *foo*)

The object is a proper list of length 3.
0. 0: :A
1. 1: :B

2. 2: :C
> q
```

We can also, in editors that support it, right-click on any object in the REPL and `inspect` them (or `C-c I` on the object to inspect in Slime). We are presented a screen where we can dive deep inside the data structure and even change it.

Let's have a quick look with a more interesting structure, an object:

```
(defclass foo ()
    ((a :accessor foo-a :initform '(:a :b :c))
     (b :accessor foo-b :initform :b)))
;; #<STANDARD-CLASS FOO>
(make-instance 'foo)
;; #<FOO {100F2B6183}>
```

We right-click on the `#<FOO` object and choose "inspect". We are presented an interactive pane (in Slime):

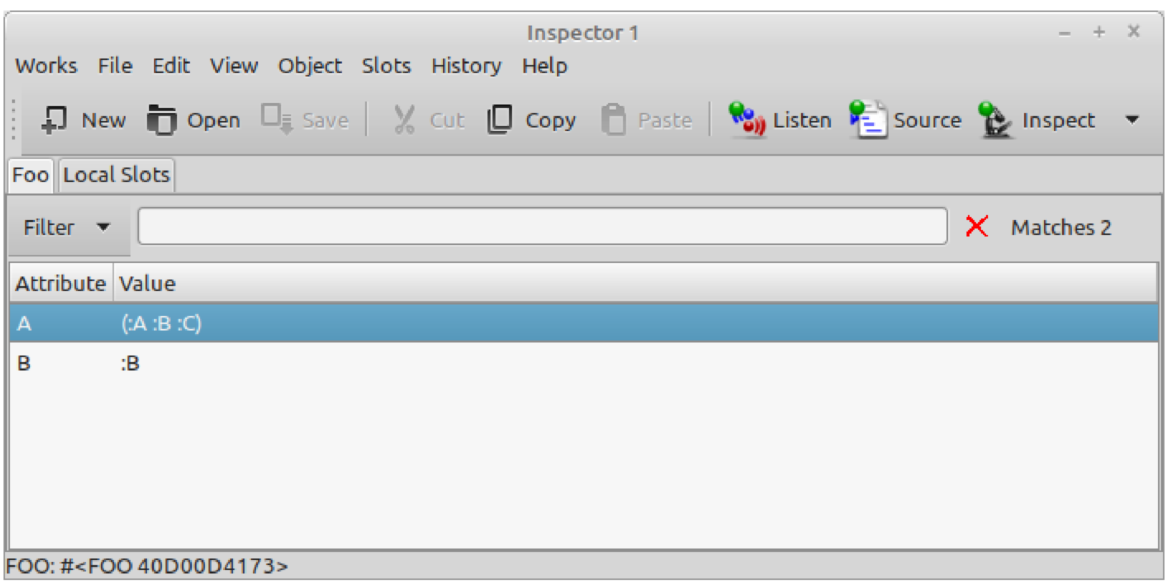


"Slime's inspector, a textual window with buttons"

When we click or press enter on the line of slot A, we inspect it further:

```
#<CONS {100F5E2A07}>
      --
A proper list:
0: :A
1: :B
2: :C
```

In LispWorks, we can use a graphical inspector:



"The LispWorks inspector window"

Trace

[trace](#) allows us to see when a function was called, what arguments it received, and the value it returned.

```
(defun factorial (n)
  (if (plusp n)
    (* n (factorial (1- n)))
    1))
```

To start tracing a function, just call `trace` with the function name (or several function names):

```
(trace factorial)
```

```
(factorial 2)
  0: (FACTORIAL 3)
    1: (FACTORIAL 2)
      2: (FACTORIAL 1)
        3: (FACTORIAL 0)
        3: FACTORIAL returned 1
      2: FACTORIAL returned 1
    1: FACTORIAL returned 2
  0: FACTORIAL returned 6
6
```

```
(untrace factorial)
```

To untrace all functions, just evaluate `(untrace)`.

To get a list of currently traced functions, evaluate `(trace)` with no arguments.

In Slime we have the shortcut `C-c M-t` to trace or untrace a function.

If you don't see recursive calls, that may be because of the compiler's optimizations. Try this before defining the function to be traced:

```
(declaim (optimize (debug 3)))  ;; or C-u C-c C-c to compile with max
```

The output is printed to `*trace-output*` (see the CLHS).

In Slime, we also have an interactive trace dialog with `M-x slime-trace-dialog` bound to `C-c T`.

Trace options

`trace` accepts options. For example, you can use `:break t` to invoke the debugger at the start of the function, before it is called (more on break below):

```
(trace factorial :break t)
(factorial 2)
```

We can define many things in one call to `trace`. For instance, options that appear before the first function name to trace are *global*, they affect all traced functions that we add afterwards. Here, `:break t` is set for every function that follows: `factorial`, `foo` and `bar`:

```
(trace :break t factorial foo bar)
```

On the contrary, if an option comes after a function name, it acts as a *local* option, only for its *preceding* function. That's how we first did. Below `foo` and `bar` come after, they are not affected by `:break`:

```
(trace factorial :break t foo bar)
```

But do you actually want to `break` *before* the function call or just *after* it? With `:break` as with many options, you can choose. These are the options for `:break`:

```
:break form  ;; before
:break-after form
:break-all form ;; before and after
```

`form` can be any form that evaluates to true.

Note that we explained the trace function of SBCL. Other implementations may have the same feature with another syntax and other option names. For example, in LispWorks it is ":break-on-exit" instead of ":break-after", and we write `(trace (factorial :break t))`.

Below are some other options but first, a trick with `:break`.

Trace options: break

The argument to an option can be any form. Here's a trick, on SBCL, to get the break window when we are about to call `factorial` with 0. `(sb-debug:arg 0)` refers to `n`, the first argument.

```
CL-USER> (trace factorial :break (equal 0 (sb-debug:arg 0)))
;; WARNING: FACTORIAL is already TRACE'd, untracing it first.
```

```
;; (FACTORIAL)
```

Running it again:

```
CL-USER> (factorial 3)
  0: (FACTORIAL 3)
    1: (FACTORIAL 2)
      2: (FACTORIAL 1)
        3: (FACTORIAL 0)

breaking before traced call to FACTORIAL:
   [Condition of type SIMPLE-CONDITION]

Restarts:
 0: [CONTINUE] Return from BREAK.
 1: [RETRY] Retry SLIME REPL evaluation request.
 2: [*ABORT] Return to SLIME's top level.
   3:   [ABORT]   abort   thread   (#<THREAD   "repl-thread"   RUNNING
{1003551BC3}>)

Backtrace:
  0: (FACTORIAL 1)
      Locals:
         N = 1   <---------- before calling (factorial 0), n equals
1.
```

Trace options: trace on conditions, trace if called from another function

`:condition` enables tracing only if the condition in `form` evaluates to true.

```
:condition form
:condition-after form
:condition-all form
```

> If :condition is specified, then trace does nothing unless Form evaluates to true at the time of the call. :condition-after is similar, but suppresses the initial printout, and is tested when the function returns. :condition-all tries both before and after.

`:wherein` can be super useful:

```
:wherein Names
```

> If specified, Names is a function name or list of names. trace does nothing unless a call to one of those functions encloses the call to this function (i.e. it

would appear in a backtrace.) Anonymous functions have string names like "DEFUN FOO".

`:report Report-Type`

If Report-Type is trace (the default) then information is reported by printing immediately. If Report-Type is nil, then the only effect of the trace is to execute other options (e.g. print or break). Otherwise, Report-Type is treated as a function designator and, for each trace event, funcalled with 5 arguments: trace depth (a non-negative integer), a function name or a function object, a keyword (:enter, :exit or :non-local-exit), a stack frame, and a list of values (arguments or return values).

See also `:print` to enrich the trace output.

It is expected that implementations extend `trace` with non-standard options. And we didn't list all available options, so please refer to your implementation's documentation:

- SBCL trace
- CCL trace
- LispWorks trace
- Allegro trace

Tracing method invocation

In SBCL, we can use `(trace foo :methods t)` to trace the execution order of method combination (before, after, around methods). For example:

```
(trace foo :methods t)

(foo 2.0d0)
  0: (FOO 2.0d0)
    1: ((SB-PCL::COMBINED-METHOD FOO) 2.0d0)
      2: ((METHOD FOO (FLOAT)) 2.0d0)
        3: ((METHOD FOO (T)) 2.0d0)
        3: (METHOD FOO (T)) returned 3
      2: (METHOD FOO (FLOAT)) returned 9
      2: ((METHOD FOO :AFTER (DOUBLE-FLOAT)) 2.0d0)
      2: (METHOD FOO :AFTER (DOUBLE-FLOAT)) returned DOUBLE
    1: (SB-PCL::COMBINED-METHOD FOO) returned 9
```

```
  0: FOO returned 9
9
```

It is also possible in CCL.

See the CLOS section for a tad more information.

The interactive debugger

Whenever an exceptional situation happens (see error handling), or when you ask for it (using `step` or `break`), the interactive debugger pops up.

It presents the error message, the available actions (*restarts*), and the backtrace. A few remarks:

- the restarts are programmable, we can create our own.
- in Slime, press `v` on a stack trace frame to view the corresponding source file location.
- hit Enter (or `t`) on a frame to toggle more details,
- use `e` to evaluate some code from within that frame,
- hit `r` to restart a given frame (see below).
- we can explore the functionality with the menu that should appear in our editor.

Compile with maximum debugging information

Usually your compiler will optimize things out and this will reduce the amount of information available to the debugger. For example sometimes we can't see intermediate variables of computations. We can change the optimization choices with:

```
(declaim (optimize (speed 0) (space 0) (debug 3)))
```

and recompile our code. You can achieve the same with a handy shortcut: `C-u C-c C-c`: the form is compiled with maximum debug settings. You can on the contrary use a negative prefix argument (`M--`) to compile for speed. And use a numeric argument to set the setting to it (you should read the docstring of `slime-compile-defun`).

Step

step is an interactive command with similar scope than `trace`. This:

```
;; note: we copied factorial over to a file, to have more debug infor
(step (factorial 3))
```

gives an interactive pane with available actions (restarts) and the backtrace:

```
Evaluating call:
  (FACTORIAL 3)
With arguments:
  3
   [Condition of type SB-EXT:STEP-FORM-CONDITION]

Restarts:
 0: [STEP-CONTINUE] Resume normal execution   <---------- stepping
actions
 1: [STEP-OUT] Resume stepping after returning from this function
 2: [STEP-NEXT] Step over call
 3: [STEP-INTO] Step into call
 4: [RETRY] Retry SLIME REPL evaluation request.
 5: [*ABORT] Return to SLIME's top level.
 --more--

Backtrace:
  0: (FACTORIAL 3)     <----------- press Enter to fold/unfold.
      Locals:
         N = 3           <----------- want to check? Move the point
here and
                                       press "e" to evaluate code on
that frame.

     1:  (SB-INT:SIMPLE-EVAL-IN-LEXENV  (LET  ((SB-IMPL::*STEP-OUT*
:MAYBE)) (UNWIND-PROTECT (SB-IMPL::WITH-STEPPING-ENABLED #))) #S(SB-
KERNEL:LEXENV  :FUNS  NIL  :VARS  NIL  :BLOCKS  NIL  :TAGS  NIL  :TYPE-
RESTRICTIONS ..
    2:  (SB-INT:SIMPLE-EVAL-IN-LEXENV  (STEP  (FACTORIAL  3))  #<NULL-
LEXENV>)
  3: (EVAL (STEP (FACTORIAL 3)))
 --more--
```

(again, be sure you compiled your function with maximum debug settings (see above). Otherwise, your compiler might do optimizations under the hood and you might not see useful information such as local variables, or you might not be able to step at all.)

You have many options here. If you are using Emacs (or any other editor actually), keep in mind that you have a “SLDB” menu that shows you the available actions, in addition to the step window.

- follow the restarts to **continue stepping**: continue the execution, step out of this function, step into the function call the point is on, step over to the next function call, or abort everything. The shortcuts are:
    - `c`: continue
    - `s`: step
    - `x`: step next
    - `o`: step out

- **inspect the backtrace** and the source code. You can go to the source file with `v`, on each stackframe (each line of the backtrace). Press `Enter` or `t` ("toggle details") on the stackframe to see more information, such as the function parameters for this call. Use `n` and `p` to navigate, use `M-n` and `M-p` to navigate to the next or previous stackframe *and* to open the corresponding source file at the same time. The point will be placed on the function being called.

- **evaluate code from within the context** of that stackframe. In Slime, use `e` ("eval in frame" and `d` to pretty-pint the result) and type a Lisp form. It will be executed in the context of the stackframe the point is on. Look, you can even inspect variables and have Slime open another inspector window. If you are on the first frame (`0:`), press `i`, then "n" to inspect the intermediate variable.

- **resume execution** from where you want. Use `r` to restart the frame the point is on. For example, go change the source code (without quitting the interactive debugger), re-compile it, re-run the frame to see if it works better. You didn't restart all the program execution, you just restarted your program from a precise point. Use `R` to return from a stackframe, by giving its return value.

NB: let's think about it, **this is awesome!** We just restarted our program from any point in time. If we work with long-running computations, we don't need to restart it from the start. We can change, re-compile our erroneous code and resume execution from where it is needed to pass, no more.

Stepping is precious. However, if you find yourself inspecting the behaviour of a function a lot, it may be a sign that you need to simplify it and divide it in smaller pieces.

And again, LispWorks has a graphical stepper.

Resume a program execution from anywhere in the stack

In this video you will find a demo that shows the process explained above: how to fix a buggy function and how to resume the program execution from anywhere in the stack, without running everything from zero again. The video shows it with Emacs and Slime, the Lem editor, both with SBCL.

Break

A call to break makes the program enter the debugger, from which we can inspect the call stack, and do everything described above in the stepper.

Breakpoints in Slime

Look at the `SLDB` menu, it shows navigation keys and available actions. Of which:

- `e` (*sldb-eval-in-frame*) prompts for an expression and evaluates it in the selected frame. This is how we can explore our intermediate variables
- `d` is similar with the addition of pretty printing the result

Once we are in a frame and detect a suspicious behavior, we can even re-compile a function at runtime and resume the program execution from where it stopped (using the "step-continue" restart or using `r` ("restart frame") on a given stackframe).

See also the Slime-star Emacs extension to set breakpoints without code annotations.

Advise and watch

advise and *watch* are available in some implementations, like CCL (advise and watch) and LispWorks. They do exist in SBCL but are not exported. `advise` allows to modify a function without changing its source, or to do something before or after its execution, similar to CLOS method combination (before, after, around methods).

`watch` will signal a condition when a thread attempts to write to an object being watched. It can be coupled with the display of the watched objects in a GUI. For a certain class of bugs (someone is changing this value, but I don't know who), this can be extremely helpful.

Cross-referencing

Your Lisp can tell you all the places where a function is referenced or called, where a global variable is set, where a macro is expanded, and so on. For example, `slime-who-calls` (`C-c C-w C-c` or the Slime > Cross-Reference menu) will show you all the places where a function is called.

See our Emacs page for a complete list of commands.

SLY stepper and SLY stickers

SLY has an improved stepper and a unique feature, stickers. You mark a piece of code, you run your code, SLY captures the results for each sticker and lets you examine the program execution interactively. It allows to see what sticker was captured, or not, so we can see at a glance the code coverage of that function call.

They are a non-intrusive alternative to `print` and `break`.

Unit tests

Last but not least, automatic testing of functions in isolation might be what you're looking for! See the testing section and a list of test frameworks and libraries.

Remote debugging

You can have your software running on a machine over the network, connect to it and debug it from home, from your development environment.

The steps involved are to start a **Swank server** on the remote machine (Swank is the backend companion of Slime), create an ssh tunnel and connect to the Swank server from our editor. Then we can browse and evaluate code on the running instance transparently.

To test this, let's define a function that prints forever.

If needed, import the dependencies first:

```
(ql:quickload '("swank" "bordeaux-threads"))

;; a little common lisp swank demo
;; while this program is running, you can connect to it from
;; another terminal or machine
;; and change the definition of doprint to print something else out!

(require :swank)
(require :bordeaux-threads)

(defparameter *counter* 0)

(defun dostuff ()
  (format t "hello world ~a!~%" *counter*))

(defun runner ()
  (swank:create-server :port 4006)
  (format t "we are past go!~%")
  (bt:make-thread (lambda ()
                    (loop repeat 5 do
                          (sleep 5)
                          (dostuff)
                          (incf *counter*)))
                  :name "do-stuff"))

(runner)
```

On the server, we can run this code with

sbcl –load demo.lisp

If you check with `(bt:all-threads)`, you'll see your Swank server running on port 4006, as well as the other thread ready to do stuff:

(#<SB-THREAD:THREAD "do-stuff" RUNNING {10027CEDC3}> #<SB-THREAD:THREAD "Swank Sentinel" waiting on: #<WAITQUEUE {10027D0003}> {10027CE8B3}> #<SB-THREAD:THREAD "Swank 4006" RUNNING {10027CEB63}> #<SB-THREAD:THREAD "main thread" RUNNING {1007C40393}>)

We do port forwarding on our development machine:

ssh -L4006:127.0.0.1:4006 username@example.com

this will securely forward port 4006 on the server at example.com to our local computer's port 4006 (Swank only accepts connections from localhost).

We connect to the running Swank with `M-x slime-connect`, choosing localhost for the host and port 4006.

We can write new code:

```
(defun dostuff ()
  (format t "goodbye world ~a!~%" *counter*))
(setf *counter* 0)
```

and eval it as usual with `C-c C-c` or `M-x slime-eval-region` for instance. The output should change.

That's how Ron Garret debugged the Deep Space 1 spacecraft from the earth in 1999:

> We were able to debug and fix a race condition that had not shown up during ground testing. (Debugging a program running on a $100M piece of hardware that is 100 million miles away is an interesting experience. Having a read-eval-print loop running on the spacecraft proved invaluable in finding and fixing the problem.

References

- "How to understand and use Common Lisp", chap. 30, David Lamkins (book download from author's site)
- Malisper: debugging Lisp series
- Two Wrongs: debugging Common Lisp in Slime
- Slime documentation: connecting to a remote Lisp
- cvberrycom: remotely modifying a running Lisp program using Swank
- Ron Garret: Lisping at the JPL
- the Remote Agent experiment: debugging code from 60 million miles away (youtube) ("AMA" on reddit)

Performance Tuning and Tips

Many Common Lisp implementations translate the source code into assembly language, so the performance is really good compared with some other interpreted languages.

However, sometimes we just want the program to be faster. This chapter introduces some techniques to squeeze the CPU power out.

Finding Bottlenecks

Acquiring Execution Time

The macro `time` is very useful for finding out bottlenecks. It takes a form, evaluates it and prints timing information in `*trace-output*`, as shown below:

```
* (defun collect (start end)
    "Collect numbers [start, end] as list."
    (loop for i from start to end
          collect i))

* (time (collect 1 10))

Evaluation took:
  0.000 seconds of real time
  0.000001 seconds of total run time (0.000001 user, 0.000000 sys
  100.00% CPU
  3,800 processor cycles
  0 bytes consed
```

By using the `time` macro it is fairly easy to find out which part of your program takes too much time.

Please note that the timing information provided here is not guaranteed to be reliable enough for marketing comparisons. It should only be used for tuning purpose, as demonstrated in this chapter.

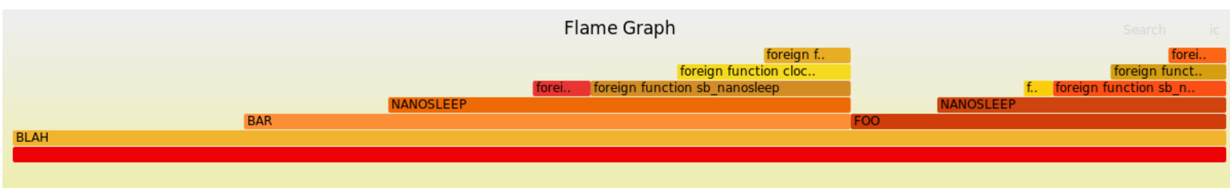
Know your Lisp's statistical profiler

Implementations ship their own profilers. SBCL has sb-profile, a "classic, per-function-call" deterministic profiler and sb-sprof, a statistical profiler. The latter works by taking samples of the program execution at regular intervals, instead of instrumenting functions like `sb-profile:profile` does.

> You might find sb-sprof more useful than the deterministic profiler when profiling functions in the common-lisp-package, SBCL internals, or code where the instrumenting overhead is excessive.

Use flamegraphs and other tracing profilers

cl-flamegraph is a wrapper around SBCL's statistical profiler to generate FlameGraph charts. Flamegraphs are a very visual way to search for hotspots in your code:



See also tracer, a tracing profiler for SBCL. Its output is suitable for display in Chrome's or Chromium's Tracing Viewer (`chrome://tracing`).

Checking Assembly Code

The function `disassemble` takes a function and prints the compiled code of it to `*standard-output*`. For example:

```
* (defun plus (a b)
    (+ a b))
PLUS

* (disassemble 'plus)
; disassembly for PLUS
```

```
; Size: 37 bytes. Origin: #x52B8063B
; 3B:   498B5D60     MOV RBX, [R13+96]  ; no-arg-parsing entry po:
                                        ; thread.binding-stack-po:
; 3F:   48895DF8     MOV [RBP-8], RBX
; 43:   498BD0       MOV RDX, R8
; 46:   488BFE       MOV RDI, RSI
; 49:   FF14250102   CALL QWORD PTR [#x52100] ; GENERIC-+
; 50:   488B75E8     MOV RSI, [RBP-24]
; 54:   4C8B45F0     MOV R8, [RBP-16]
; 58:   488BE5       MOV RSP, RBP
; 5B:   F8           CLC
; 5C:   5D           POP RBP
; 5D:   C3           RET
; 5E:   CC0F         BREAK 15   ; Invalid argument count trap
```

The code above was evaluated in SBCL. In some other implementations such as CLISP, `disassembly` might return something different:

```
* (defun plus (a b)
    (+ a b))
PLUS

* (disassemble 'plus)
Disassembly of function PLUS
2 required arguments
0 optional arguments
No rest parameter
No keyword parameters
4 byte-code instructions:
0     (LOAD&PUSH 2)
1     (LOAD&PUSH 2)
2     (CALLSR 2 55)                     ; +
5     (SKIP&RET 3)
NIL
```

It is because SBCL compiles the Lisp code into machine code, while CLISP does not.

Using Declare Expression

The *declare expression* can be used to provide hints for compilers to perform various optimization. Please note that these hints are implementation-dependent. Some implementations such as SBCL support this feature, and you may refer to their own documentation for detailed information. Here only some basic techniques mentioned in CLHS are introduced.

In general, declare expressions can occur only at the beginning of the bodies of certain forms, or immediately after a documentation string if the context allows. Also, the content of a declare expression is restricted to limited forms. Here we introduce some of them that are related to performance tuning.

Please keep in mind that these optimization skills introduced in this section are strongly connected to the Lisp implementation selected. Always check their documentation before using `declare`!

Speed and Safety

Lisp allows you to specify several quality properties for the compiler using the declaration `optimize`. Each quality may be assigned a value from 0 to 3, with 0 being "totally unimportant" and 3 being "extremely important".

The most significant qualities might be `safety` and `speed`.

By default, Lisp considers code safety to be much more important than speed. But you may adjust the weight for more aggressive optimization.

```
* (defun max-original (a b)
    (max a b))
MAX-ORIGINAL

* (disassemble 'max-original)
; disassembly for MAX-ORIGINAL
; Size: 144 bytes. Origin: #x52D450EF
; 7A7:       8D46F1      lea eax, [rsi-15]              ; no-arg
; 7AA:       A801        test al, 1
; 7AC:       750E        jne L0
; 7AE:       3C0A        cmp al, 10
; 7B0:       740A        jeq L0
```

```
; 7B2:       A80F        test al, 15
; 7B4:       7576        jne L5
; 7B6:       807EF11D    cmp byte ptr [rsi-15], 29
; 7BA:       7770        jnbe L5
; 7BC: L0:   8D43F1      lea eax, [rbx-15]
; 7BF:       A801        test al, 1
; 7C1:       750E        jne L1
; 7C3:       3C0A        cmp al, 10
; 7C5:       740A        jeq L1
; 7C7:       A80F        test al, 15
; 7C9:       755A        jne L4
; 7CB:       807BF11D    cmp byte ptr [rbx-15], 29
; 7CF:       7754        jnbe L4
; 7D1: L1:   488BD3      mov rdx, rbx
; 7D4:       488BFE      mov rdi, rsi
; 7D7:       B9C1030020  mov ecx, 536871873   ; generic->
; 7DC:       FFD1        call rcx
; 7DE:       488B75F0    mov rsi, [rbp-16]
; 7E2:       488B5DF8    mov rbx, [rbp-8]
; 7E6:       7E09        jle L3
; 7E8:       488BD3      mov rdx, rbx
; 7EB: L2:   488BE5      mov rsp, rbp
; 7EE:       F8          clc
; 7EF:       5D          pop rbp
; 7F0:       C3          ret
; 7F1: L3:   4C8BCB      mov r9, rbx
; 7F4:       4C894DE8    mov [rbp-24], r9
; 7F8:       4C8BC6      mov r8, rsi
; 7FB:       4C8945E0    mov [rbp-32], r8
; 7FF:       488BD3      mov rdx, rbx
; 802:       488BFE      mov rdi, rsi
; 805:       B929040020  mov ecx, 536871977   ; generic-=
; 80A:       FFD1        call rcx
; 80C:       4C8B45E0    mov r8, [rbp-32]
; 810:       4C8B4DE8    mov r9, [rbp-24]
; 814:       488B75F0    mov rsi, [rbp-16]
; 818:       488B5DF8    mov rbx, [rbp-8]
; 81C:       498BD0      mov rdx, r8
; 81F:       490F44D1    cmoveq rdx, r9
; 823:       EBC6        jmp L2
; 825: L4:   CC0A        break 10            ; error trap
; 827:       04          byte #X04
; 828:       13          byte #X13           ; OBJECT-NOT-REAL-E
```

```
; 829:       FE9B01      byte #XFE, #X9B, #X01      ; RBX
; 82C: L5:   CC0A        break 10           ; error trap
; 82E:       04          byte #X04
; 82F:       13          byte #X13          ; OBJECT-NOT-REAL-E
; 830:       FE1B03      byte #XFE, #X1B, #X03           ; RSI
; 833:       CC0A        break 10           ; error trap
; 835:       02          byte #X02
; 836:       19          byte #X19          ; INVALID-ARG-COUNT
; 837:       9A          byte #X9A          ; RCX

* (defun max-with-speed-3 (a b)
    (declare (optimize (speed 3) (safety 0)))
    (max a b))
MAX-WITH-SPEED-3

* (disassemble 'max-with-speed-3)
; disassembly for MAX-WITH-SPEED-3
; Size: 92 bytes. Origin: #x52D452C3
; 3B:       48895DE0        mov [rbp-32], rbx               ;
; 3F:       488945E8        mov [rbp-24], rax
; 43:       488BD0          mov rdx, rax
; 46:       488BFB          mov rdi, rbx
; 49:       B9C1030020      mov ecx, 536871873     ; generic->
; 4E:       FFD1            call rcx
; 50:       488B45E8        mov rax, [rbp-24]
; 54:       488B5DE0        mov rbx, [rbp-32]
; 58:       7E0C            jle L1
; 5A:       4C8BC0          mov r8, rax
; 5D: L0:   498BD0          mov rdx, r8
; 60:       488BE5          mov rsp, rbp
; 63:       F8              clc
; 64:       5D              pop rbp
; 65:       C3              ret
; 66: L1:   488945E8        mov [rbp-24], rax
; 6A:       488BF0          mov rsi, rax
; 6D:       488975F0        mov [rbp-16], rsi
; 71:       4C8BC3          mov r8, rbx
; 74:       4C8945F8        mov [rbp-8], r8
; 78:       488BD0          mov rdx, rax
; 7B:       488BFB          mov rdi, rbx
; 7E:       B929040020      mov ecx, 536871977      ; generic-=
; 83:       FFD1            call rcx
; 85:       488B45E8        mov rax, [rbp-24]
```

```
; 89:       488B75F0         mov rsi, [rbp-16]
; 8D:       4C8B45F8         mov r8, [rbp-8]
; 91:       4C0F44C6         cmoveq r8, rsi
; 95:       EBC6             jmp L0
```

As you can see, the generated assembly code is much shorter (92 bytes VS 144). The compiler was able to perform optimizations. Yet we can do better by declaring types.

Type Hints

As mentioned in the [*Type System*]() chapter, Lisp has a relatively powerful type system. You may provide type hints so that the compiler may reduce the size of the generated code.

```
* (defun max-with-type (a b)
    (declare (optimize (speed 3) (safety 0)))
    (declare (type integer a b))
    (max a b))
MAX-WITH-TYPE

* (disassemble 'max-with-type)
; disassembly for MAX-WITH-TYPE
; Size: 42 bytes. Origin: #x52D48A23
; 1B:       488BF7           mov rsi, rdi                     ; 
; 1E:       488975F0         mov [rbp-16], rsi
; 22:       488BD8           mov rbx, rax
; 25:       48895DF8         mov [rbp-8], rbx
; 29:       488BD0           mov rdx, rax
; 2C:       B98C030020       mov ecx, 536871820    ; generic-<
; 31:       FFD1             call rcx
; 33:       488B75F0         mov rsi, [rbp-16]
; 37:       488B5DF8         mov rbx, [rbp-8]
; 3B:       480F4CDE         cmovl rbx, rsi
; 3F:       488BD3           mov rdx, rbx
; 42:       488BE5           mov rsp, rbp
; 45:       F8               clc
; 46:       5D               pop rbp
; 47:       C3               ret
```

The size of generated assembly code shrunk to about 1/3 of the size. What about speed?

```
* (time (dotimes (i 10000) (max-original 100 200)))
Evaluation took:
  0.000 seconds of real time
  0.000107 seconds of total run time (0.000088 user, 0.000019 sy
  100.00% CPU
  361,088 processor cycles
  0 bytes consed

* (time (dotimes (i 10000) (max-with-type 100 200)))
Evaluation took:
  0.000 seconds of real time
  0.000044 seconds of total run time (0.000036 user, 0.000008 sy
  100.00% CPU
  146,960 processor cycles
  0 bytes consed
```

You see, by specifying type hints, our code runs much faster!

But wait…What happens if we declare wrong types? The answer is: it depends.

For example, SBCL treats type declarations in a special way. It performs different levels of type checking according to the safety level. If safety level is set to 0, no type checking will be performed. Thus a wrong type specifier might cause a lot of damage.

More on Type Declaration with `declaim`

If you try to evaluate a `declare` form in the top level, you might get the following error:

```
Execution of a form compiled with errors.
Form:
  (DECLARE (SPEED 3))
Compile-time error:
  There is no function named DECLARE.  References to DECLARE in s
(like starts of blocks) are unevaluated expressions, but here th
```

```
being evaluated, which invokes undefined behaviour.
   [Condition of type SB-INT:COMPILED-PROGRAM-ERROR]
```

This is because type declarations have [scopes](). In the examples above, we have seen type declarations applied to a function.

During development it is usually useful to raise the importance of safety in order to find out potential problems as soon as possible. On the contrary, speed might be more important after deployment. However, it might be too verbose to specify declaration expression for each single function.

The macro [`declaim`]() provides such possibility. It can be used as a top level form in a file and the declarations will be made at compile-time.

```
* (declaim (optimize (speed 0) (safety 3)))
NIL

* (defun max-original (a b)
    (max a b))
MAX-ORIGINAL

* (disassemble 'max-original)
; disassembly for MAX-ORIGINAL
; Size: 181 bytes. Origin: #x52D47D9C
...

* (declaim (optimize (speed 3) (safety 3)))
NIL

* (defun max-original (a b)
    (max a b))
MAX-ORIGINAL

* (disassemble 'max-original)
; disassembly for MAX-ORIGINAL
; Size: 142 bytes. Origin: #x52D4815D
```

Please note that `declaim` works in **compile-time** of a file. It is mostly used to make some declares local to that file. And it is unspecified whether or not the compile-time side-effects of a declaim persist after the file has been compiled.

Declaring function types

Another useful declaration is a `ftype` declaration which establishes the relationship between the function argument types and the return value type. If the type of passed arguments matches the declared types, the return value type is expected to match the declared one. Because of that, a function can have more than one `ftype` declaration associated with it. A `ftype` declaration restricts the type of the argument every time the function is called. It has the following form:

```
 (declaim (ftype (function (arg1 arg2 ...) return-value)
                 function-name1))
```

If the function returns `nil`, its return type is `null`. This declaration does not put any restriction on the types of arguments by itself. It only takes effect if the provided arguments have the specified types – otherwise no error is signaled and declaration has no effect. For example, the following declamation states that if the argument to the function `square` is a `fixnum`, the value of the function will also be a `fixnum`:

```
(declaim (ftype (function (fixnum) fixnum) square))
(defun square (x) (* x x))
```

If we provide it with the argument which is not declared to be of type `fixnum`, no optimization will take place:

```
(defun do-some-arithmetic (x)
  (the fixnum (+ x (square x))))
```

Now let's try to optimize the speed. The compiler will state that there is type uncertainty:

```
(defun do-some-arithmetic (x)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (the fixnum (+ x (square x))))
```

```
; compiling (DEFUN DO-SOME-ARITHMETIC ...)

; file: /tmp/slimeRzDh1R
 in: DEFUN DO-SOME-ARITHMETIC
;     (+ TEST-FRAMEWORK::X (TEST-FRAMEWORK::SQUARE TEST-FRAMEWORK
;
; note: forced to do GENERIC-+ (cost 10)
;       unable to do inline fixnum arithmetic (cost 2) because:
;       The first argument is a NUMBER, not a FIXNUM.
;       unable to do inline (signed-byte 64) arithmetic (cost 5)
;       The first argument is a NUMBER, not a (SIGNED-BYTE 64).
;       etc.
;
; compilation unit finished
;   printed 1 note



      (disassemble 'do-some-arithmetic)
; disassembly for DO-SOME-ARITHMETIC
; Size: 53 bytes. Origin: #x52CD1D1A
; 1A:       488945F8         MOV [RBP-8], RAX   ; no-arg-parsing
; 1E:       488BD0           MOV RDX, RAX
; 21:       4883EC10         SUB RSP, 16
; 25:       B902000000       MOV ECX, 2
; 2A:       48892C24         MOV [RSP], RBP
; 2E:       488BEC           MOV RBP, RSP
; 31:       E8C2737CFD       CALL #x504990F8    ; #<FDEFN SQUARE>
; 36:       480F42E3         CMOVB RSP, RBX
; 3A:       488B45F8         MOV RAX, [RBP-8]
; 3E:       488BFA           MOV RDI, RDX
; 41:       488BD0           MOV RDX, RAX
; 44:       E807EE42FF       CALL #x52100B50    ; GENERIC-+
; 49:       488BE5           MOV RSP, RBP
; 4C:       F8               CLC
; 4D:       5D               POP RBP
; 4E:       C3               RET
NIL
```

Now we can add a type declaration for `x`, so the compiler can assume that the expression `(square x)` is a `fixnum`, and use the fixnum-specific `+`:

```
(defun do-some-arithmetic (x)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (declare (type fixnum x))
  (the fixnum (+ x (square x))))
```

```
     (disassemble 'do-some-arithmetic)
```

```
; disassembly for DO-SOME-ARITHMETIC
; Size: 48 bytes. Origin: #x52C084DA
; 4DA:       488945F8         MOV [RBP-8], RAX     ; no-arg-parsin
; 4DE:       4883EC10         SUB RSP, 16
; 4E2:       488BD0           MOV RDX, RAX
; 4E5:       B902000000       MOV ECX, 2
; 4EA:       48892C24         MOV [RSP], RBP
; 4EE:       488BEC           MOV RBP, RSP
; 4F1:       E8020C89FD       CALL #x504990F8      ; #<FDEFN SQUAR
; 4F6:       480F42E3         CMOVB RSP, RBX
; 4FA:       488B45F8         MOV RAX, [RBP-8]
; 4FE:       4801D0           ADD RAX, RDX
; 501:       488BD0           MOV RDX, RAX
; 504:       488BE5           MOV RSP, RBP
; 507:       F8               CLC
; 508:       5D               POP RBP
; 509:       C3               RET
NIL
```

Code Inline

The declaration inline replaces function calls with function body, if the compiler supports it. It will save the cost of function calls but will potentially increase the code size. The best situation to use `inline` might be those small but frequently used functions. The following snippet shows how to encourage and prohibit code inline.

```
;; The globally defined function DISPATCH should be open-coded,
;; if the implementation supports inlining, unless a NOTINLINE
;; declaration overrides this effect.
(declaim (inline dispatch))
(defun dispatch (x) (funcall (get (car x) 'dispatch) x))
```

```lisp
;; Here is an example where inlining would be encouraged.
;; Because function DISPATCH was defined as INLINE, the code
;; inlining will be encouraged by default.
(defun use-dispatch-inline-by-default ()
  (dispatch (read-command)))

;; Here is an example where inlining would be prohibited.
;; The NOTINLINE here only affects this function.
(defun use-dispatch-with-declare-notinline  ()
  (declare (notinline dispatch))
  (dispatch (read-command)))

;; Here is an example where inlining would be prohibited.
;; The NOTINLINE here affects all following code.
(declaim (notinline dispatch))
(defun use-dispatch-with-declaim-noinline ()
  (dispatch (read-command)))

;; Inlining would be encouraged because you specified it.
;; The INLINE here only affects this function.
(defun use-dispatch-with-inline ()
  (declare (inline dispatch))
  (dispatch (read-command)))
```

Please note that when the inlined functions change, all the callers must be re-compiled.

Optimizing Generic Functions

Using Static Dispatch

Generic functions provide much convenience and flexibility during development. However, the flexibility comes with cost: generic methods are much slower than trivial functions. The performance cost becomes a burden especially when the flexibility is not needed.

The package [inlined-generic-function]() provides functions to convert generic functions to static dispatch, moving the dispatch cost to compile-time. You just need to define generic function as a `inlined-generic-function`.

Caution

This package is declared as experimental thus is not recommended to be used in a serious software production. Use it at your own risk!

```
* (defgeneric plus (a b)
    (:generic-function-class inlined-generic-function))
#<INLINED-GENERIC-FUNCTION HELLO::PLUS (2)>

* (defmethod plus ((a fixnum) (b fixnum))
    (+ a b))
#<INLINED-METHOD HELLO::PLUS (FIXNUM FIXNUM) {10056D7513}>

* (defun func-using-plus (a b)
    (plus a b))
FUNC-USING-PLUS

* (defun func-using-plus-inline (a b)
    (declare (inline plus))
    (plus a b))
FUNC-USING-PLUS-INLINE

* (time
   (dotimes (i 100000)
     (func-using-plus 100 200)))
Evaluation took:
  0.018 seconds of real time
  0.017819 seconds of total run time (0.017800 user, 0.000019 sy
  100.00% CPU
  3 lambdas converted
  71,132,440 processor cycles
  6,586,240 bytes consed

* (time
   (dotimes (i 100000)
     (func-using-plus-inline 100 200)))
Evaluation took:
  0.001 seconds of real time
  0.000326 seconds of total run time (0.000326 user, 0.000000 sy
  0.00% CPU
  1,301,040 processor cycles
  0 bytes consed
```

The inlining is not enabled by default because once inlined, changes made to methods will not be reflected.

It can be enabled globally by adding `:inline-generic-function` flag in [`*features*`].

```
* (push :inline-generic-function *features*)
(:INLINE-GENERIC-FUNCTION :SLYNK :CLOSER-MOP :CL-FAD :BORDEAUX-TI
:THREAD-SUPPORT :CL-PPCRE ALEXANDRIA.0.DEV::SEQUENCE-EMPTYP :QUI(
:QUICKLISP-SUPPORT-HTTPS :SB-BSD-SOCKETS-ADDRINFO :ASDF3.3 :ASDF:
:ASDF3 :ASDF2 :ASDF :OS-UNIX :NON-BASE-CHARS-EXIST-P :ASDF-UNICOI
:X86-64 :64-BIT :64-BIT-REGISTERS :ALIEN-CALLBACKS :ANSI-CL :AVX:
:C-STACK-IS-CONTROL-STACK :CALL-SYMBOL :COMMON-LISP :COMPACT-INS
:COMPARE-AND-SWAP-VOPS :CYCLE-COUNTER :ELF :FP-AND-PC-STANDARD-S/
```

When this feature is present, all inlinable generic functions are inlined unless it is declared `notinline`.

Block compilation

SBCL got block compilation on version 2.0.2, which was in CMUCL since 1991 but a little forgotten since.

You can enable block compilation with a one-liner:

```
(setq *block-compile-default* t)
```

But what is it?

Block compilation addresses a known aspect of dynamic languages: function calls to global, top-level functions are expensive.

> Much more expensive than in a statically compiled language. They're slow because of the late-bound nature of top-level defined functions, allowing arbitrary redefinition while the program is running and forcing runtime checks on whether the function is being called with the right number or types of arguments. This type of call is known as a "full

call” in Python (the compiler used in CMUCL and SBCL, not to be confused with the programming language), and their calling convention permits the most runtime flexibility.

But local calls, the ones inside a top-level functions (for example `lambdas`, `labels` and `flets`) are fast.

> These calls are more ‘static’ in the sense that they are treated more like function calls in static languages, being compiled “together” and at the same time as the local functions they reference, allowing them to be optimized at compile-time. For example, argument checking can be done at compile time because the number of arguments of the callee is known at compile time, unlike in the full call case where the function, and hence the number of arguments it takes, can change dynamically at runtime at any point. Additionally, the local call calling convention can allow for passing unboxed values like floats around, as they are put into unboxed registers never used in the full call convention, which must use boxed argument and return value registers.

So enabling block compilation kind of turns your code into a giant `labels` form.

One evident possible drawback, dependending on your application, is that you can’t redefine functions at runtime anymore.

We can also enable block compilation with the `:block-compile` keyword to `compile-file`:

```
(defun foo (x y)
  (print (bar x y))
  (bar x y))

(defun bar (x y)
  (+ x y))

(defun fact (n)
  (if (zerop n)
      1
      (* n (fact (1- n)))))
```

```
> (compile-file "foo.lisp" :block-compile t :entry-points nil)
> (load "foo.fasl")

> (sb-disassem:disassemble-code-component #'foo)
```

If you inspect the assembly,

> you [will] see that FOO and BAR are now compiled into the same component (with local calls), and both have valid external entry points. This improves locality of code quite a bit and still allows calling both FOO and BAR externally from the file (e.g. in the REPL). […]

But there is one more goody block compilation adds…

> Notice we specified `:entry-points` nil above. That's telling the compiler to still create external entry points to every function in the file, since we'd like to be able to call them normally from outside the code component (i.e. the compiled compilation unit, here the entire file).

For more explanations, I refer you to the mentioned blog post, the current de-facto documentation for SBCL, in addition to [CMUCL's documentation](#) (note that the form-by-form level granularity in CMUCL (`(declaim (start-block ...)) ... (declaim (end-block ..))`) is missing in SBCL, at the time of writing).

Finally, be aware that "block compiling and inlining currently does not interact very well [in SBCL]".

Scripting. Command line arguments. Executables.

Using a program from a REPL is fine and well, but once it's ready we'll surely want to call it from the terminal. We can run Lisp **scripts** for this.

Next, if we want to distribute our program easily, we'll want to build an **executable**.

Lisp implementations differ in their processes, but they all create **self-contained executables**, for the architecture they are built on. The final user doesn't need to install a Lisp implementation, he can run the software right away.

Start-up times are near to zero, specially with SBCL and CCL.

Binaries **size** are large-ish. They include the whole Lisp including its libraries, the names of all symbols, information about argument lists to functions, the compiler, the debugger, source code location information, and more.

Note that we can similarly build self-contained executables for **web apps**.

Scripting with Common Lisp

Create a file named `hello` (you can drop the .lisp extension) and add this:

```
#!/usr/bin/env -S sbcl --script
(require :uiop)
(format t "hello ~a!~&" (uiop:getenv "USER"))
```

Make the script executable (`chmod +x hello`) and run it:

```
$ ./hello
hello me!
```

Nice! We can use this to a great extent already.

In addition, the script was quite fast to start, 0.03s on my system.

However, we will get longer startup times as soon as we add dependencies. The solution is to build a binary. They start even faster, with all dependencies compiled.

Quickloading dependencies from a script

Say you don't bother with an .asd project definition yet, you just want to write a quick script, but you need to load a quicklisp dependency. You'll need a bit more ceremony:

```
#!/usr/bin/env -S sbcl --script

(require :uiop)

;; We want quicklisp, which is loaded from our initfile,
;; after a classical installation.
;; However the --script flag doesn't load our init file:
;; it implies --no-sysinit --no-userinit --disable-debugger --er
;; So, please load it:
(load "~/.sbclrc")

;; Load a quicklisp dependency silently.
(ql:quickload "str" :silent t)

(princ (str:concat "hello " (uiop:getenv "USER") "!"))
```

Accordingly, you could only use `require`, if the quicklisp dependency is already installed:

```
;; replace loading sbclrc and ql:quickload.
(require :str)
```

Also note that when you put a `ql:quickload` in the middle of your code, you can't load the file anymore, you can't `C-c C-k` from your editor. This is because the reader will see the "quickload" without running it yet, then sees “str:concat”, a call to a package that doesn’t exist (it wasn’t loaded yet). Common Lisp has you covered, with a form that executes code during the read phase:

```
;; you shouldn't need this. Use an .asd system definition!
(eval-when (:load-toplevel :compile-toplevel :execute)
  (ql:quickload "str" :silent t))
```

but ASDF project definitions are here for a reason. Find me another language that makes you install dependencies in the middle of the application code.

Building a self-contained executable

With SBCL - Images and Executables

How to build (self-contained) executables is, by default, implementation-specific (see below for portable ways). With SBCL, as says [its documentation](), it is a matter of calling `save-lisp-and-die` with the `:executable` argument to T:

```
(sb-ext:save-lisp-and-die #P"path/name-of-executable"
                          :toplevel #'my-app:main-function
                          :executable t)
```

`sb-ext` is an SBCL extension to run external processes. See other [SBCL extensions]() (many of them are made implementation-portable in other libraries).

`:executable t` tells to build an executable instead of an image. We could build an image to save the state of our current Lisp image, to come back working with it later. This is especially useful if we made a lot of work that is computing intensive. In that case, we re-use the image with `sbcl --core name-of-image`.

`:toplevel` gives the program’s entry point, here `my-app:main-function`. Don’t forget to `export` the symbol, or use `my-app::main-function` (with two colons).

If you try to run this in Slime, you'll get an error about threads running:

> Cannot save core with multiple threads running.

We must run the command from a simple SBCL repl, from the terminal.

I suppose your project has Quicklisp dependencies. You must then:

- ensure Quicklisp is installed and loaded at the Lisp startup (you completed Quicklisp installation),
- `asdf:load-asd` the project's .asd (recommended instead of just `load`),
- install the dependencies,
- build the executable.

That gives:

```
(asdf:load-asd "my-app.asd")
(ql:quickload "my-app")
(sb-ext:save-lisp-and-die #p"my-app-binary"
                          :toplevel #'my-app:main
                          :executable t)
```

From the command line, or from a Makefile, use `--load` and `--eval`:

```
build:
    sbcl --load my-app.asd \
         --eval '(ql:quickload :my-app)' \
                --eval  "(sb-ext:save-lisp-and-die  #p\"my-app\"
:toplevel #'my-app:main :executable t)"
```

With ASDF

Now that we've seen the basics, we need a portable method. Since its version 3.1, ASDF allows to do that. It introduces the `make` command, that reads parameters from the .asd. Add this to your .asd declaration:

```
:build-operation "program-op" ;; leave as is
:build-pathname "<here your final binary name>"
:entry-point "<my-package:main-function>"
```

and call `asdf:make :my-package`.

So, in a Makefile:

```
LISP ?= sbcl

build:
    $(LISP) --load my-app.asd \
        --eval '(ql:quickload :my-app)' \
        --eval '(asdf:make :my-app)' \
        --eval '(quit)'
```

With Deploy - ship foreign libraries dependencies

All this is good, you can create binaries that work on your machine… but maybe not on someone else's or on your server. Your program probably relies on C shared libraries that are defined somewhere on your filesystem. For example, `libssl` might be located on

/usr/lib/x86_64-linux-gnu/libssl.so.1.1

but on your VPS, maybe somewhere else.

Deploy to the rescue.

It will create a `bin/` directory with your binary and the required foreign libraries. It will auto-discover the ones your program needs, but you can also help it (or tell it to not do so much).

Its use is very close to the above recipe with `asdf:make` and the `.asd` project configuration. Use this:

```
:defsystem-depends-on (:deploy)  ;; (ql:quickload "deploy") befo
:build-operation "deploy-op"     ;; instead of "program-op"
:build-pathname "my-application-name"  ;; doesn't change
:entry-point "my-package:my-start-function"  ;; doesn't change
```

and build your binary with `(asdf:make :my-app)` like before.

Now, ship the `bin/` directory to your users.

When you run the binary, you'll see it uses the shipped libraries:

```
$ ./my-app
 ==> Performing warm boot.
   -> Runtime directory is /home/debian/projects/my-app/bin/
   -> Resource directory is /home/debian/projects/my-app/bin/
 ==> Running boot hooks.
 ==> Reloading foreign libraries.
   -> Loading foreign library #<LIBRARY LIBRT>.
   -> Loading foreign library #<LIBRARY LIBMAGIC>.
 ==> Launching application.
 […]
```

Success!

A note regarding `libssl`. It's easier, on Linux at least, to rely on your OS' current installation, so we'll tell Deploy to not bother shipping it (nor `libcrypto`):

```
#+linux (deploy:define-library cl+ssl::libssl :dont-deploy T)
#+linux (deploy:define-library cl+ssl::libcrypto :dont-deploy T)
```

The day you want to ship a foreign library that Deploy doesn't find, you can instruct it like this:

```
(deploy:define-library cl+ssl::libcrypto
  ;;                   ^^^ CFFI system name.
  ;;                   Find it with a call to "apropos".
  :path "/usr/lib/x86_64-linux-gnu/libcrypto.so.1.1")
```

A last remark. Once you built your binary and you run it for the first time, you might get a funny message from ASDF that tries to upgrade itself, finds nothing into a `~/common-lisp/asdf/` repository, and quits. To tell it to not upgrade itself, add this into your .asd:

```
;; Tell ASDF to not update itself.
(deploy:define-hook (:deploy asdf) (directory)
  (declare (ignorable directory))
  #+asdf (asdf:clear-source-registry)
  #+asdf (defun asdf:upgrade-asdf () nil))
```

You can also silence Deploy's start-up messages by adding this in your build script, before `asdf:make` is called:

(push :deploy-console *features*)

And there is more, so we refer you to Deploy's documentation.

With Roswell or Buildapp

Roswell, an implementation manager, script launcher and much more, has the `ros build` command, that should work for many implementations.

This is how we can make our application easily installable by others, with a `ros install my-app`. See Roswell's documentation.

Be aware that `ros build` adds core compression by default. That adds a significant startup overhead of the order of 150ms (for a simple app, startup time went from about 30ms to 180ms). You can disable it with `ros build <app.ros> --disable-compression`. Of course, core compression reduces your binary size significantly. See the table below, "Size and startup times of executables per implementation".

We'll finish with a word on Buildapp, a battle-tested and still popular "application for SBCL or CCL that configures and saves an executable Common Lisp image".

Example usage:

```
buildapp --output myapp \
         --asdf-path . \
         --asdf-tree ~/quicklisp/dists \
         --load-system my-app \
         --entry my-app:main
```

Many applications use it (for example, pgloader), it is available on Debian: `apt install buildapp`, but you shouldn't need it now with asdf:make or Roswell.

For web apps

We can similarly build a self-contained executable for our web appplication. It would thus contain a web server and would be able to run on the command line:

$ ./my-web-app Hunchentoot server is started. Listening on localhost:9003.

Note that this runs the production webserver, not a development one, so we can run the binary on our VPS right away and access the application from the outside.

We have one thing to take care of, it is to find and put the thread of the running web server on the foreground. In our `main` function, we can do something like this:

```
(defun main ()
  (start-app :port 9003) ;; our start-app, for example clack:cla
  ;; let the webserver run.
  ;; warning: hardcoded "hunchentoot".
  ;; You can simply run (sleep most-positive-fixnum)
  (handler-case (bt:join-thread (find-if (lambda (th)
                                            (search "hunchentoot
                                          (bt:all-threads)))
    ;; Catch a user's C-c
    (#+sbcl sb-sys:interactive-interrupt
      #+ccl  ccl:interrupt-signal-condition
      #+clisp system::simple-interrupt-condition
      #+ecl ext:interactive-interrupt
      #+allegro excl:interrupt-signal
      () (progn
           (format *error-output* "Aborting.~&")
           (clack:stop *server*)
           (uiop:quit)))
    (error (c) (format t "Woops, an unknown error occured:~&~a~&
```

We used the `bordeaux-threads` library (`(ql:quickload "bordeaux-threads")`, alias `bt`) and `uiop`, which is part of ASDF so already loaded, in order to exit in a portable way (`uiop:quit`, with an optional return code, instead of `sb-ext:quit`).

Size and startup times of executables per implementation

SBCL isn't the only Lisp implementation. ECL, Embeddable Common Lisp, transpiles Lisp programs to C. That creates a smaller executable.

According to this reddit source, ECL produces indeed the smallest executables of all, an order of magnitude smaller than SBCL, but with a longer startup time.

CCL's binaries seem to be as fast to start up as SBCL and nearly half the size.

```
| program size | implementation |  CPU | startup time |
|--------------+----------------+------+--------------|
|           28 | /bin/true      |  15% |        .0004 |
|         1005 | ecl            | 115% |        .5093 |
|        48151 | sbcl           |  91% |        .0064 |
|        27054 | ccl            |  93% |        .0060 |
|        10162 | clisp          |  96% |        .0170 |
|         4901 | ecl.big        | 113% |        .8223 |
|        70413 | sbcl.big       |  93% |        .0073 |
|        41713 | ccl.big        |  95% |        .0094 |
|        19948 | clisp.big      |  97% |        .0259 |
```

You'll also want to investigate the proprietary Lisps' tree shakers capabilities.

Regarding compilation times, CCL is famous for being fast in that regards. ECL is more involved and takes the longer to compile of these three implementations.

Building a smaller binary with SBCL's core compression

Building with SBCL's core compression can dramatically reduce your application binary's size. In our case, it reduced it from 120MB to 23MB, for a loss of a dozen milliseconds of start-up time, which was still under 50ms.

```
<strong>Note:</strong> SBCL 2.2.6 switched to compression with
zstd instead of zlib, which provides smaller binaries and
faster compression and decompression times. Un-official numbers
are: about 4x faster compression, 2x faster decompression, and
smaller binaries by 10%.
```

Your SBCL must be built with core compression, see the documentation: [Saving-a-Core-Image]

Is it the case ?

```
(find :sb-core-compression *features*)
:SB-CORE-COMPRESSION
```

Yes, it is the case with this SBCL installed from Debian.

With SBCL

In SBCL, we would give an argument to `save-lisp-and-die`, where `:compression`

> may be an integer from -7 to 22, corresponding to zstd compression levels, or t (which is equivalent to the default compression level, 9).

For a simple "Hello, world" program:

Program size	Compression level
46MB	Without compression
22MB	-7
12MB	9
11MB	22

For a bigger project like StumpWM, an X window manager written in Lisp:

Program size	Compression level
58MB	Without compression
27MB	-7
15MB	9
13MB	22

With ASDF

However, we prefer to do this with ASDF (or rather, UIOP). Add this in your .asd:

```
#+sb-core-compression
(defmethod asdf:perform ((o asdf:image-op) (c asdf:system))
  (uiop:dump-image (asdf:output-file o c)
                   :executable t
                   :compression t))
```

With Deploy

Also, the Deploy library can be used to build a fully standalone application. It will use compression if available.

Deploy is specifically geared towards applications with foreign library dependencies. It collects all the foreign shared libraries of dependencies, such as libssl.so in the `bin` subdirectory.

And voilà !

Parsing command line arguments

SBCL stores the command line arguments into `sb-ext:*posix-argv*`.

But that variable name differs from implementations, so we want a way to handle the differences for us.

We have `(uiop:command-line-arguments)`, shipped in ASDF and included in nearly all implementations. From anywhere in your code, you can simply check if a given string is present in this list:

```
(member "-h" (uiop:command-line-arguments) :test #'string-equal)
```

That's good, but we also want to parse the arguments, have facilities to check short and long options, build a help message automatically, etc.

We chose the Clingon library, because it may have the richest feature set:

- it handles subcommands,
- it supports various kinds of options (flags, integers, booleans, counters, enums…),
- it generates Bash and Zsh completion files as well as man pages,
- it is extensible in many ways,
- we can easily try it out on the REPL
- etc

Let's download it:

(ql:quickload "clingon")

As often, work happens in two phases:

- we first declare the options that our application accepts, their kind (flag, string, integer…), their long and short names and the required ones.
- we ask Clingon to parse the command-line options and run our app.

Declaring options

We want to represent a command-line tool with this possible usage:

$ myscript [-h, –help] [-n, –name NAME]

Ultimately, we need to create a Clingon command (with `clingon:make-command`) to represent our application. A command is composed of options and of a handler function, to do the logic.

So first, let's create options. Clingon already handles "–help" for us, but not the short version. Here's how we use `clingon:make-option` to create an option:

```
(clingon:make-option
 :flag                ;; <--- option kind. A "flag" does not exp
 :description "short help"
 ;; :long-name "help" ;; <--- long name, sans the "--" prefix, b
 :short-name #\h      ;; <--- short name, a character
 ;; :required t       ;; <--- is this option always required? In
```

```
 :key :help)           ;; <--- the internal reference to use with
```

This is a **flag**: if “-h” is present on the command-line, the option’s value will be truthy, otherwise it will be falsy. A flag does not expect an argument, it’s here for itself.

Similar kind of options would be:

- `:boolean`: that one expects an argument, which can be “true” or 1 to be truthy. Anything else is considered falsy.
- `:counter`: a counter option counts how many times the option is provided on the command line. Typically, use it with `-v` / `--verbose`, so the user could use `-vvv` to have extra verbosity. In that case, the option value would be 3. When this option is not provided on the command line, Clingon sets its value to 0.

We’ll create a second option (“–name” or “-n” with a parameter) and we put everything in a litle function.

```
;; The naming with a "/" is just our convention.
(defun cli/options ()
  "Returns a list of options for our main command"
  (list
   (clingon:make-option
    :flag
    :description "short help."
    :short-name #\h
    :key :help)
   (clingon:make-option
    :string              ;; <--- string type: expects one parame
    :description "Name to greet"
    :short-name #\n
    :long-name "name"
    :env-vars '("USER")     ;; <-- takes this default value if t
    :initial-value "lisper" ;; <-- default value if nothing else
    :key :name)))
```

The second option we created is of kind `:string`. This option expects one argument, which will be parsed as a string. There is also `:integer`, to parse the argument as an integer.

There are more option kinds of Clingon, which you will find on its good documentation: `:choice`, `:enum`, `:list`, `:filepath`, `:switch` and so on.

Top-level command

We have to tell Clingon about our top-level command. `clingon:make-command` accepts some descriptive fields, and two important ones:

- `:options` is a list of Clingon options, each created with `clingon:make-option`
- `:handler` is the function that will do the app's logic.

And finally, we'll use `clingon:run` in our main function (the entry point of our binary) to parse the command-line arguments, and apply our command's logic. During development, we can also manually call `clingon:parse-command-line` to try things out.

Here's a minimal command. We'll define our handler function afterwards:

```
(defun cli/command ()
  "A command to say hello to someone"
  (clingon:make-command
   :name "hello"
   :description "say hello"
   :version "0.1.0"
   :authors '("John Doe <john.doe@example.org")
   :license "BSD 2-Clause"
   :options (cli/options) ;; <-- our options
   :handler #'null))  ;; <--  to change. See below.
```

At this point, we can already test things out on the REPL.

Testing options parsing on the REPL

Use `clingon:parse-command-line`: it wants a top-level command, and a list of command-line arguments (strings):

```
CL-USER> (clingon:parse-command-line (cli/command) '("-h" "-n" '
```

```
#<CLINGON.COMMAND:COMMAND name=hello options=5 sub-commands=0>
```

It works!

We can even `inspect` this command object, we would see its properties (name, hooks, description, context…), its list of options, etc.

Let’s try again with an unknown option:

```
CL-USER> (clingon:parse-command-line (cli/command) '("-x"))
;; => debugger: Unknown option -x of kind SHORT
```

In that case, we are dropped into the interactive debugger, which says

```
Unknown option -x of kind SHORT
   [Condition of type CLINGON.CONDITIONS:UNKNOWN-OPTION]
```

and we are provided a few restarts:

```
Restarts:
 0: [DISCARD-OPTION] Discard the unknown option
 1: [TREAT-AS-ARGUMENT] Treat the unknown option as a free argument
 2: [SUPPLY-NEW-VALUE] Supply a new value to be parsed
 3: [RETRY] Retry SLIME REPL evaluation request.
 4: [*ABORT] Return to SLIME's top level.
```

which are very practical. If we needed, we could create an `:around` method for `parse-command-line`, handle Clingon’s conditions with `handler-bind` and use its restarts, to do something different with unknown options. But we don’t need that yet, if ever: we want our command-line parsing engine to warn us on invalid options.

Last but not least, we can see how Clingon prints our CLI tool’s usage information:

```
CL-USER> (clingon:print-usage (cli/command) t)
NAME:
  hello - say hello

USAGE:
```

```
  hello [options] [arguments ...]

OPTIONS:
      --help          display usage information and exit
      --version       display version and exit
  -h                  short help.
   -n, --name <VALUE>   Name to greet [default: lisper] [env:
$USER]

AUTHORS:
  John Doe <john.doe@example.org

LICENSE:
  BSD 2-Clause
```

We can tweak the "USAGE" part with the `:usage` key parameter of the lop-level command.

Handling options

When the parsing of command-line arguments succeeds, we need to do something with them. We introduce two new Clingon functions:

- `clingon:getopt` is used to get an option's value by its `:key`
- `clingon:command-arguments` gets use the free arguments remaining on the command-line.

Here's how to use them:

```
CL-USER> (let ((command (clingon:parse-command-line (cli/commanc
           (format t "name is: ~a~&" (clingon:getopt command :na
           (format t "free args are: ~s~&" (clingon:command-argu
name is: you
free args are: ("last")
NIL
```

It is with them that we will write the handler of our top-level command:

```
(defun cli/handler (cmd)
  "The handler function of our top-level command"
```

```
  (let ((free-args (clingon:command-arguments cmd))
        (name (clingon:getopt cmd :name)))  ;; <-- using the opt
    (format t "Hello, ~a!~%" name)
    (format t "You have provided ~a more free arguments~%"
            (length free-args))
    (format t "Bye!~%")))
```

We must tell our top-level command to use this handler:

```
;; from above:
(defun cli/command ()
  "A command to say hello to someone"
  (clingon:make-command
   ...
   :handler #'cli/handler))  ;; <-- changed.
```

We now only have to write the main entry point of our binary and we're done.

By the way, `clingon:getopt` returns 3 values:

- the option's value
- a boolean, indicating wether this option was provided on the command-line
- the command which provided the option for this value.

See also `clingon:opt-is-set-p`.

Main entry point

This can be any function, but to use Clingon, use its `run` function:

```
(defun main ()
  "The main entrypoint of our CLI program"
  (clingon:run (cli/command)))
```

To use this main function as your binary entry point, see above how to build a Common Lisp binary. A reminder: set it in your .asd system declaration:

```
:entry-point "my-package::main"
```

And that's about it. Congratulations, you can now properly parse command-line arguments!

Go check Clingon's documentation, because there is much more to it: sub-commands, contexts, hooks, handling a C-c, developing new options such as an email kind, Bash and Zsh completion…

Catching a C-c termination signal

By default, **Clingon provides a handler for SIGINT signals**. It makes the application to immediately exit with the status code 130.

If your application needs some clean-up logic, you can use an `unwind-protect` form. However, it might not be appropriate for all cases, so Clingon advertises to use the [with-user-abort]() helper library.

Below we show how to catch a C-c manually. Because by default, you would get a Lisp stacktrace.

We built a simple binary, we ran it and pressed `C-c`. Let's read the stacktrace:

```
$ ./my-app
sleep…
^C
debugger invoked on a SB-SYS:INTERACTIVE-INTERRUPT in thread
<== condition name
#<THREAD "main thread" RUNNING {1003156A03}>:
  Interactive interrupt at #x7FFFF6C6C170.

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from
SBCL.

restarts (invokable by number or by possibly-abbreviated name):
     0: [CONTINUE            ] Return from SB-UNIX:SIGINT.
<== it was a SIGINT indeed
  1: [RETRY-REQUEST] Retry the same request.
```

The signaled condition is named after our implementation: `sb-sys:interactive-interrupt`. We just have to surround our application code with a `handler-case`:

```lisp
(handler-case
    (run-my-app free-args)
  (sb-sys:interactive-interrupt ()
      (progn
        (format *error-output* "Abort.~&")
        (opts:exit))))
```

This code is only for SBCL though. We know about trivial-signal, but we were not satisfied with our test yet. So we can use something like this:

```lisp
(handler-case
    (run-my-app free-args)
  (#+sbcl sb-sys:interactive-interrupt
   #+ccl  ccl:interrupt-signal-condition
   #+clisp system::simple-interrupt-condition
   #+ecl ext:interactive-interrupt
   #+allegro excl:interrupt-signal
   ()
   (opts:exit)))
```

here `#+` includes the line at compile time depending on the implementation. There's also `#-`. What `#+` does is to look for symbols in the `*features*` list. We can also combine symbols with `and`, `or` and `not`.

Continuous delivery of executables

We can make a Continuous Integration system (Travis CI, Gitlab CI,…) build binaries for us at every commit, or at every tag pushed or at whichever other policy.

See Continuous Integration.

See also

- SBCL-GOODIES - Allows to distribute SBCL binaries with foreign libraries: `libssl`, `libcrypto` and `libfixposix` are statically baked in. This removes the need of Deploy, when only these three foreign libraries are used.
  - it was released on February, 2023.

Credit

- cl-torrents’ tutorial
- lisp-journey/web-dev

Testing the code

So you want to easily test the code you're writing? The following recipe covers how to write automated tests and see their code coverage. We also give pointers to plug those in modern continuous integration services like GitHub Actions, Gitlab CI, Travis CI or Coveralls.

We will be using a mature testing framework called FiveAM. It supports test suites, random testing, test fixtures (to a certain extent) and, of course, interactive development.

Previously on the Cookbook, the recipe was cooked with Prove. It used to be a widely liked testing framework but, because of some shortcomings, its repository was later archived. Its successor Rove is not stable enough and lacks some features, so we didn't pick it. There are also some other testing frameworks to explore if you feel like it.

FiveAM has an API documentation. You may inspect it or simply read the docstrings in code. Most of the time, they would provide sufficient information that answers your questions… if you didn't find them here. Let's get started.

Testing with FiveAM

FiveAM has 3 levels of abstraction: check, test and suite. As you may have guessed:

1. A **check** is a single assertion that checks that its argument is truthy. The most used check is `is`. For example, `(is (= 2 (+ 1 1)))`.
2. A **test** is the smallest runnable unit. A test case may contain multiple checks. Any check failure leads to the failure of the whole test.
3. A **suite** is a collection of tests. When a suite is run, all tests inside would be performed. A suite allows paternity, which means that running a suite will run all the tests defined in it and in its children suites.

A simple code sample containing the 3 basic blocks mentioned above can be shown as follows:

```
(def-suite* my-suite)

(test my-test
  (is (= 2 (+ 1 1))))
```

It is totally up to the user to decide the hierarchy of tests and suites. Here we mainly focus on the usage of FiveAM.

Suppose we have built a rather complex system and the following functions are part of it:

```
;; We have a custom "file doesn't exist" condition.
(define-condition file-not-existing-error (error)
  ((filename :type string :initarg :filename :reader filename)))

;; We have a function that tries to read a file and signals the a
;; if the file doesn't exist.
(defun read-file-as-string (filename &key (error-if-not-exists t)
  "Read file content as string. FILENAME specifies the path of fi

Keyword ERROR-IF-NOT-EXISTS specifies the operation to perform wh
is not found. T (by default) means an error will be signaled. Whe
the function will return NIL in that case."
  (cond
    ((uiop:file-exists-p filename)
     (uiop:read-file-string filename))
    (error-if-not-exists
     (error 'file-not-existing-error :filename filename))
    (t nil)))
```

We will write tests for that code. In particular, we must ensure:

- that the content read in a file is the expected content,
- that the condition is signaled if the file doesn’t exist.

Install and load

`FiveAM` is in Quicklisp and can be loaded with the following command:

```
(ql:quickload "fiveam")
```

The package is named `fiveam` with a nickname `5am`. For the sake of simplicity, we will ignore the package prefix in the following code samples.

It is like we `:used` fiveam in our test package definition. You can also follow along in the REPL with `(use-package :fiveam)`.

Here is a package definition you can use:

```
(in-package :cl-user)
(defpackage my-fiveam-test
  (:use :cl
        :fiveam))
(in-package :my-fiveam-test)
```

Defining suites (`def-suite`, `def-suite*`)

Testing in FiveAM usually starts by defining a suite. A suite helps separating tests to smaller collections that makes them more organized. It is highly recommended to define a single *root* suite for the sake of ASDF integration. We will talk about it later, now let's focus on the testing itself.

The code below defines a suite named `my-system`. We will use it as the root suite for the whole system.

```
(def-suite my-system
  :description "Test my system")
```

Then let's define another suite for testing the `read-file-as-string` function.

```
;; Define a suite and set it as the default for the following te
(def-suite read-file-as-string
  :description "Test the read-file-as-string function."
  :in my-system)
(in-suite read-file-as-string)
```

```
;; Alternatively, the following line is a combination of the 2 l
(def-suite* read-file-as-string :in my-system)
```

Here a new suite named `read-file-as-string` has been defined. It is declared to be a child suite of `my-system` as specified by the `:in` keyword. The macro `in-suite` sets it as the default suite for the tests defined later.

Defining tests

Before diving into tests, here is a brief introduction of the available checks you may use inside tests:

- The `is` macro is likely the most used check. It simply checks if the given expression returns a true value and generates a `test-passed` or `test-failure` result accordingly.
- The `skip` macro takes a reason and generates a `test-skipped` result.
- The `signals` macro checks if the given condition was signaled during execution.

There is also:

- `finishes`: passes if the assertion body executes to normal completion. In other words if body does signal, return-from or throw, then this test fails.
- `pass`: just make the test pass.
- `is-true`: like `is`, but unlike it this check does not inspect the assertion body to determine how to report the failure. Similarly, there is `is-false`.

Please note that all the checks accept an optional reason, as string, that can be formatted with format directives (see more below). When omitted, FiveAM generates a report that explains the failure according to the arguments passed to the function.

The `test` macro provides a simple way to define a test with a name.

Note that below, we expect two files to exist: `/tmp/hello.txt` should contain "hello" and `/tmp/empty.txt` should be empty.

```
;; Our first "base" case: we read a file that contains "hello".
(test read-file-as-string-normal-file
  (let ((result (read-file-as-string "/tmp/hello.txt")))
    ;; Tip: put the expected value as the first argument of = or
    ;; FiveAM generates a more readable report following this co
    (is (string= "hello" result))))

;; We read an empty file.
(test read-file-as-string-empty-file
  (let ((result (read-file-as-string "/tmp/empty.txt")))
    (is (not (null result)))
    ;; The reason can be used to provide formatted text.
    (is (= 0 (length result)))
        "Empty string expected but got ~a" result))

;; Now we test that reading a non-existing file signals our cond
(test read-file-as-string-non-existing-file
  (let ((result (read-file-as-string "/tmp/non-existing-file.txt"
                                     :error-if-not-exists nil)))
    (is (null result)
      "Reading a file should return NIL when :ERROR-IF-NOT-EXISTS
  ;; SIGNALS accepts the unquoted name of a condition and a body
  ;; Here it checks if FILE-NOT-EXISTING-ERROR is signaled.
  (signals file-not-existing-error
    (read-file-as-string "/tmp/non-existing-file.txt"
                         :error-if-not-exists t)))
```

In the above code, three tests were defined with 5 checks in total. Some checks were actually redundant for the sake of demonstration. You may put all the checks in one big test, or in multiple scenarios. It is up to you.

The macro `test` is a convenience for `def-test` to define simple tests. You may read its docstring for a more complete introduction, for example to read about `:depends-on`.

Running tests

FiveAm provides multiple ways to run tests. The macro `run!` is a good start point during development. It accepts a name of suite or test and run it, then prints testing report in standard output. Let’s run the tests now!

```
(run! 'my-system)
; Running test suite MY-SYSTEM
;  Running test READ-FILE-AS-STRING-EMPTY-FILE ..
;  Running test READ-FILE-AS-STRING-NON-EXISTING-FILE ..
;  Running test READ-FILE-AS-STRING-NORMAL-FILE .
;  Did 5 checks.
;     Pass: 5 (100%)
;     Skip: 0 ( 0%)
;     Fail: 0 ( 0%)
;  => T, NIL, NIL
```

If we mess `read-file-as-string-non-existing-file` up by replacing `/tmp/non-existing-file.txt` with `/tmp/hello.txt`, the test would fail (sure!) as expected:

```
(run! 'read-file-as-string-non-existing-file)
; Running test READ-FILE-AS-STRING-NON-EXISTING-FILE ff
;  Did 2 checks.
;     Pass: 0 ( 0%)
;     Skip: 0 ( 0%)
;     Fail: 2 (100%)
;  Failure Details:
;  --------------------------------
;  READ-FILE-AS-STRING-NON-EXISTING-FILE []:
;       Should return NIL when :ERROR-IF-NOT-EXISTS is set to NI
;  --------------------------------
;  --------------------------------
;  READ-FILE-AS-STRING-NON-EXISTING-FILE []:
;       Failed to signal a FILE-NOT-EXISTING-ERROR.
;  --------------------------------
;  => NIL
; (#<IT.BESE.FIVEAM::TEST-FAILURE {10064485F3}>
;  #<IT.BESE.FIVEAM::TEST-FAILURE {1006438663}>)
; NIL
```

The behavior of the suite/test runner can be customized by the `*on-failure*` variable, which controls what to do when a check failure happens. It can be set to one of the following values:

- `:debug` to drop to the debugger.
- `:backtrace` to print a backtrace and continue.

- `NIL` (default) to simply continue and print the report.

There is also `*on-error*`.

Running tests as they are compiled

Under normal circumstances, a test is written and compiled (with the usual `C-c C-c` in Slime) separately from the moment it is run. If you want to run the test when it is defined (with `C-c C-c`), set this:

```
(setf fiveam:*run-test-when-defined* t)
```

Custom and shorter tests explanations

We said earlier that a check accepts an optional custom reason that can be formatted with `format` directives. Here's a simple example.

We are testing a math function:

```
(fiveam:test simple-maths
  (is (= 3 (+ 1 1))))
```

When we `run!` it, we see this somewhat lengthy but informative output (and that's very important):

```
Running test suite NIL
 Running test SIMPLE-MATHS f
 Did 1 check.
    Pass: 0 ( 0%)
    Skip: 0 ( 0%)
    Fail: 1 (100%)

 Failure Details:
 --------------------------------
 SIMPLE-MATHS []:

(+ 1 1)

 evaluated to

2
```

```
 which is not

=

 to

3


 --------------------------------
```

Now, we can give it a custom reason:

```
(fiveam:test simple-maths
  (is (= 3 (+ 1 1))
      "Maths should work, right? ~a. Another parameter is: ~S" t
```

And we will see:

```
Running test suite NIL
 Running test SIMPLE-MATHS f
 Did 1 check.
    Pass: 0 ( 0%)
    Skip: 0 ( 0%)
    Fail: 1 (100%)

 Failure Details:
 --------------------------------
 SIMPLE-MATHS []:
      Maths should work, right? T. Another parameter is: :FOO
 --------------------------------
```

Fixtures

FiveAM also provides a feature called **fixtures** for setting up testing context. The goal is to ensure that some functions are not called and always return the same result. Think functions hitting the network: you want to isolate the network call in a small function and write a fixture so that in your tests, this function always returns the same, known result. (But if you do so, you might also need an “end to end” test that tests with real data and all your code…)

However, FiveAM’s fixture system is nothing more than a macro, it is not fully-featured compared to other libraries such as Mockingbird, and even

FiveAM's maintainer encourages to "just use a macro" instead.

Mockingbird (and maybe other libraries), in addition to the basic feature descibed above, also allows to count the number of times a function was called, with what arguments, and so on.

Random checking

The goal of random testing is to assist the developer in generating test cases, and thus, to find cases that the developer would not have thought about.

We have a few data generators at our disposal, for example:

```
(gen-float)
#<CLOSURE (LAMBDA () :IN GEN-FLOAT) {1005A906AB}>

(funcall (gen-float))
9.220082e37

(funcall (gen-integer :max 27 :min -16))
26
```

or again, `gen-string`, `gen-list`, `gen-tree`, `gen-buffer`, `gen-character`.

And we have a function to run 100 checks, taking each turn a new value from the given generators: `for-all`:

```
(test randomtest
  (for-all ((a (gen-integer :min 1 :max 10))
            (b (gen-integer :min 1 :max 10)))
    "Test random tests."
    (is (<= a b))))
```

When you `run! 'randomtest` this, I expect you will hit an error. You can't possibly always get `a` lower than `b`, can you?

For more, see FiveAM's documentation.

See also cl-quickcheck and Check-it, inspired by Haskell's QuickCheck test framework.

ASDF integration

So it would be nice to provide a one-line trigger to test our `my-system` system. Recall that we said it is better to provide a root suite? Here is the reason:

```
(asdf:defsystem my-system
  ;; Parts omitted.
  :in-order-to ((test-op (test-op :my-system/test))))

(asdf:defsystem mitogratOr/test
  ;; Parts omitted.
  :perform (test-op (op c)
                    (symbol-call :fiveam :run!
                                 (find-symbol* :my-system :my-sys
```

The last line tells ASDF to load symbol `:my-system` from `my-system/test` package and call `fiveam:run!`. It fact, it is equivalent to `(run! 'my-system)` as mentioned above.

Running tests on the terminal

Until now, we ran our tests from our editor's REPL. How can we run them from a terminal window?

As always, the required steps are as follow:

- start our Lisp
- make sure Quicklisp is enabled (if we have external dependencies)
- load our main system
- load the test system
- run the FiveAM tests.

You could put them in a new `run-tests.lisp` file:

```
(load "mysystem.lisp")
(load "mysystem-tests.lisp") ;; <-- where all the FiveAM tests a
(in-package :mysystem-tests)
```

```
(run!)  ;; <-- run all the tests and print the report.
```

and you could invoke it like so, from a source file or from a Makefile:

```
rlwrap sbcl --non-interactive --load mysystem.asd --eval '(ql:qu
;; we assume Quicklisp is installed and loaded. This can be done
```

Before going that route however, have a look at the `CI-Utils` tool that we use in the Continuous Integration section below. It provides a `run-fiveam` command that can do all that for you.

But let us highlight something you'll have to take care of if you ran your tests like this: the **exit code**. Indeed, `(run!)` prints a report, but it doesn't say to your Lisp wether the tests were successful or not, and wether to exit with an exit code of 0 (for success) or more (for errors). So, if your testst were run on a CI system, the CI status would be always green, even if tests failed. To remedy that, replace `run!` by:

```
(let ((result (run!)))
  (cond
    ((null result)
     (log:info "Tests failed!")  ;; FiveAM printed the report al
     (uiop:quit 1))
    (t
     (log:info "All pass.")
     (uiop:quit))))
```

Check with `echo $?` on your shell that the exit code is correct.

Testing report customization

It is possible to generate our own testing report. The macro `run!` is nothing more than a composition of `explain!` and `run`.

Instead of generating a testing report like its cousin `run!`, the function `run` runs suite or test passed in and returns a list of `test-result` instance, usually instances of `test-failure` or `test-passed` sub-classes.

A class `text-explainer` is defined as a basic class for testing report generator. A generic function `explain` is defined to take a `text-plainer` instance and a `test-result` instance (returned by `run`) and generate testing report. The following 2 code snippets are equivalent:

```
(run! 'read-file-as-string-non-existing-file)

(explain (make-instance '5am::detailed-text-explainer)
         (run 'read-file-as-string-non-existing-file))
```

By creating a new sub-class of `text-explainer` and a method `explain` for it, it is possible to define a new test reporting system.

The following code just provides a proof-of-concept implementation. You may need to read the source code of `5am::detailed-text-explainer` to fully understand it.

```
(defclass my-explainer (5am::text-explainer)
  ())

(defmethod 5am:explain ((explainer my-explainer) results &optiona
  (loop for result in results
        do (case (type-of result)
             ('5am::test-passed
              (format stream "~%Test ~a passed" (5am::name (5am:
             ('5am::test-failure
              (format stream "~%Test ~a failed" (5am::name (5am:

(explain (make-instace 'my-explainer)
         (run 'read-file-as-string-non-existing-file))
; Test READ-FILE-AS-STRING-NON-EXISTING-FILE failed
; Test READ-FILE-AS-STRING-NON-EXISTING-FILE passed => NIL
```

Interactively fixing unit tests

Common Lisp is interactive by nature (or so are most implementations), and testing frameworks make use of it. It is possible to ask the framework to open the debugger on a failing test, so that we can inspect the stack trace and go to the erroneous line instantly, fix it and re-run the test from where it left off, by choosing the suggested *restart*.

With FiveAM, set `fiveam:*on-failure*` to `:debug`:

```
(setf fiveam:*on-failure* :debug)
```

You will be dropped into the interactive debugger if an error occurs.

Use `:backtrace` to print a backtrace, continue to run the following tests and print FiveAM's report.

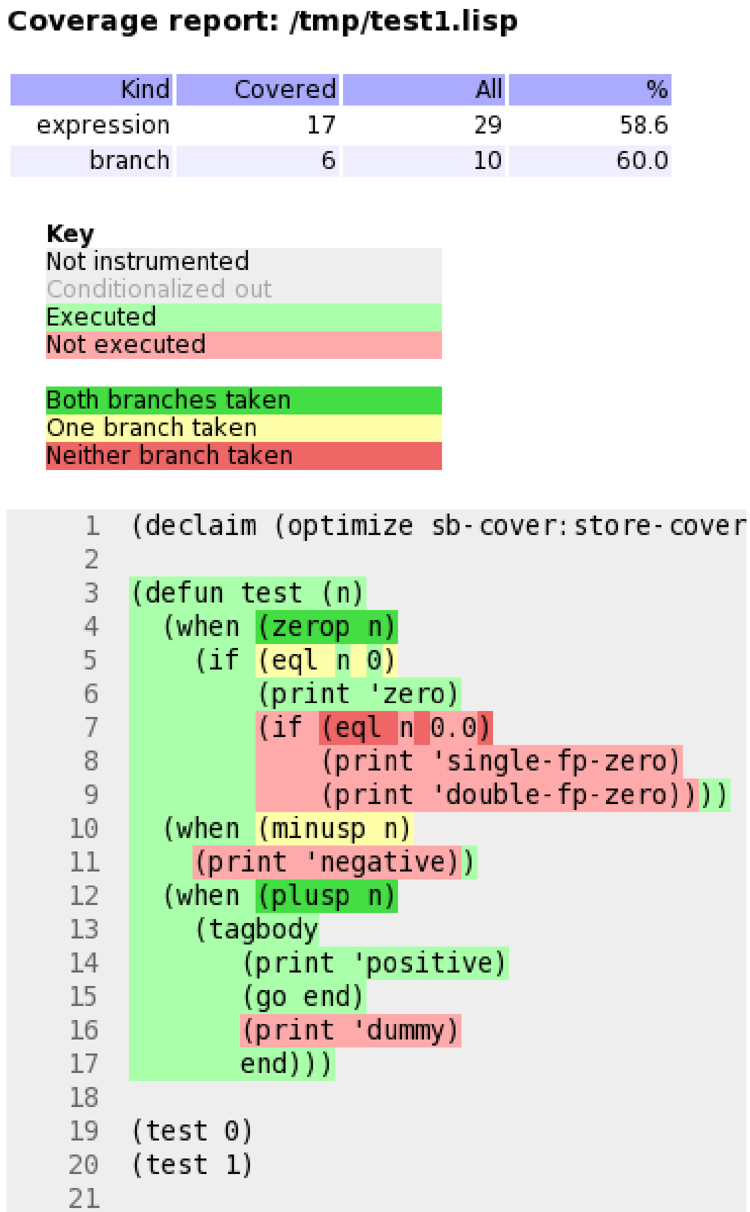
The default is `nil`: carry on the tests execution and print the report.

Note that in the debugger:

- `<enter>` on a backtrace shows more of it
- `v` on a backtrace goes to the corresponding line or function.
- you can discover more options with the menu.

Code coverage

A code coverage tool produces a visual output that allows to see what parts of our code were tested or not:



Such capabilities are included into Lisp implementations. For example, SBCL has the sb-cover module and the feature is also built-in in CCL or LispWorks.

Generating an html test coverage output

Let’s do it with SBCL’s sb-cover.

Coverage reports are only generated for code compiled using `compile-file` with the value of the `sb-cover:store-coverage-data` optimization quality set to 3.

```
;;; Load SB-COVER
(require :sb-cover)

;;; Turn on generation of code coverage instrumentation
;;; in the compiler
(declaim (optimize sb-cover:store-coverage-data))

;;; Load some code, ensuring that it's recompiled
;;; with the new optimization policy.
(asdf:oos 'asdf:load-op :cl-ppcre-test :force t)

;;; Run the test suite.
(fiveam:run! yoursystem-test)
```

Produce a coverage report, set the output directory:

```
(sb-cover:report "coverage/")
```

Finally, turn off instrumentation:

```
(declaim (optimize (sb-cover:store-coverage-data 0)))
```

You can open your browser at `../yourproject/t/coverage/cover-index.html` to see the report like the capture above or like this code coverage of cl-ppcre.

Continuous Integration

Continuous Integration is important to run automatic tests after a commit or before a pull request, to run code quality checks, to build and distribute your software… well, to automate everything about software.

We want our programs to be portable across Lisp implementations, so we’ll set up our CI pipeline to run our tests against several of them (it could be

SBCL and CCL of course, but while we're at it ABCL, ECL and possibly more).

We have a choice of Continuous Integration services: Travis CI, Circle, Gitlab CI, now also GitHub Actions, etc (many existed before GitHub Actions, if you wonder). We'll have a look at how to configure a CI pipeline for Common Lisp, and we'll focus a little more on Gitlab CI on the last part.

We'll also quickly show how to publish coverage reports to the Coveralls service. cl-coveralls helps to post our coverage to the service.

GitHub Actions, Circle CI, Travis… with CI-Utils

We'll use CI-Utils, a set of utilities that comes with many examples. It also explains more precisely what is a CI system and compares a dozen of services.

It relies on Roswell to install the Lisp implementations and to run the tests. They all are installed with a bash one-liner:

curl -L https://raw.githubusercontent.com/roswell/roswell/release/scripts/install-for-ci.sh | bash

(note that on the Gitlab CI example, we use a ready-to-use Docker image that contains them all)

It also ships with a test runner for FiveAM, which eases some rough parts (like returning the right error code to the terminal). We install ci-utils with Roswell, and we get the `run-fiveam` executable.

Then we can run our tests:

run-fiveam -e t -l foo/test :foo-tests # foo is our project

Following is the complete `.travis.yml` file.

The first part should be self-explanatory:

```
""" Example configuration for Travis CI """
```

```
### Example configuration for Travis CI ###
language: generic

addons:
  homebrew:
    update: true
    packages:
    - roswell
  apt:
    packages:
      - libc6-i386 # needed for a couple implementations
      - default-jre # needed for abcl


# Runs each lisp implementation on each of the listed OS
os:
  - linux
#  - osx # OSX has a long setup on travis, so it's likely easier
#          to just run select implementations on OSX.
```

This is how we configure the implementations matrix, to run our tests on several Lisp implementations. We also send the test coverage made with SBCL to Coveralls.

```
env:
  global:
    - PATH=~/.roswell/bin:$PATH
    - ROSWELL_INSTALL_DIR=$HOME/.roswell
#    - COVERAGE_EXCLUDE=t  # for rove
  jobs:
    # The implementation and whether coverage
    # is sent to coveralls are controlled
    # with these environmental variables
    - LISP=sbcl-bin COVERALLS=true
    - LISP=ccl-bin
    - LISP=abcl
    - LISP=ecl   # warn: in our experience,
    # compilations times can be long on ECL.

# Additional OS/Lisp combinations can be added
# to those generated above
jobs:
  include:
```

```
    - os: osx
      env: LISP=sbcl-bin
    - os: osx
      env: LISP=ccl-bin
```

Some jobs can be marked as allowed to fail:

```
# Note that this should only be used if there is no interest
# for the library to work on that system
#  allow_failures:
#    - env: LISP=abcl
#    - env: LISP=ecl
#    - env: LISP=cmucl
#    - env: LISP=alisp
#      os: osx

  fast_finish: true
```

We finally install Roswell, the implementations, and we run our tests.

```
cache:
  directories:
    - $HOME/.roswell
    - $HOME/.config/common-lisp

install:
  - curl -L https://raw.githubusercontent.com/roswell/roswell/re
  - ros install ci-utils #for run-fiveam
#  - ros install rove #for [run-] rove

  # If asdf 3.16 or higher is needed, uncomment the following li
  #- mkdir -p ~/common-lisp
  #- if [ "$LISP" == "ccl-bin" ]; then git clone https://gitlab.

script:
  - run-fiveam -e t -l foo/test :foo-tests
  #- rove foo.asd
```

Below with Gitlab CI, we’ll use a Docker image that already contains the Lisp binaries and every Debian package required to build Quicklisp libraries.

Gitlab CI

Gitlab CI is part of Gitlab and is available on Gitlab.com, for public and private repositories. Let's see straight away a simple `.gitlab-ci.yml`:

```
variables:
  QUICKLISP_ADD_TO_INIT_FILE: "true"

image: clfoundation/sbcl:latest

before_script:
  - install-quicklisp
    -  git  clone  https://github.com/foo/bar  ~/quicklisp/local-
projects/

test:
  script:
    - make test
```

Gitlab CI is based on Docker. With `image` we tell it to use the `latest` tag of the clfoundation/sbcl image. This includes the latest version of SBCL, many OS packages useful for CI purposes, and a script to install Quicklisp. Gitlab will load the image, clone our project and put us at the project root with administrative rights to run the rest of the commands.

`test` is a "job" we define, `script` is a recognized keywords that takes a list of commands to run.

Suppose we must install dependencies before running our tests: `before_script` will run before each job. Here we install Quicklisp (adding it to SBCL's init file), and clone a library where Quicklisp can find it.

We can try locally ourselves. If we already installed Docker and started its daemon (`sudo service docker start`), we can do:

docker run –rm -it -v /path/to/local/code:/usr/local/share/common-lisp/source clfoundation/sbcl:latest bash

This will download the lisp image (±300MB compressed), mount some local code in the image where indicated, and drop us in bash. Now we can try a `make test`.

Here is a more complete example that tests against several CL implementations in parallel:

```
variables:
  IMAGE_TAG: latest
  QUICKLISP_ADD_TO_INIT_FILE: "true"
  QUICKLISP_DIST_VERSION: latest

image: clfoundation/$LISP:$IMAGE_TAG

stages:
  - test
  - build

before_script:
  - install-quicklisp
  - git clone https://github.com/foo/bar ~/quicklisp/local-projec

.test:
  stage: test
  script:
    - make test

abcl test:
  extends: .test
  variables:
    LISP: abcl

ccl test:
  extends: .test
  variables:
    LISP: ccl

ecl test:
  extends: .test
  variables:
    LISP: ecl

sbcl test:
  extends: .test
  variables:
```

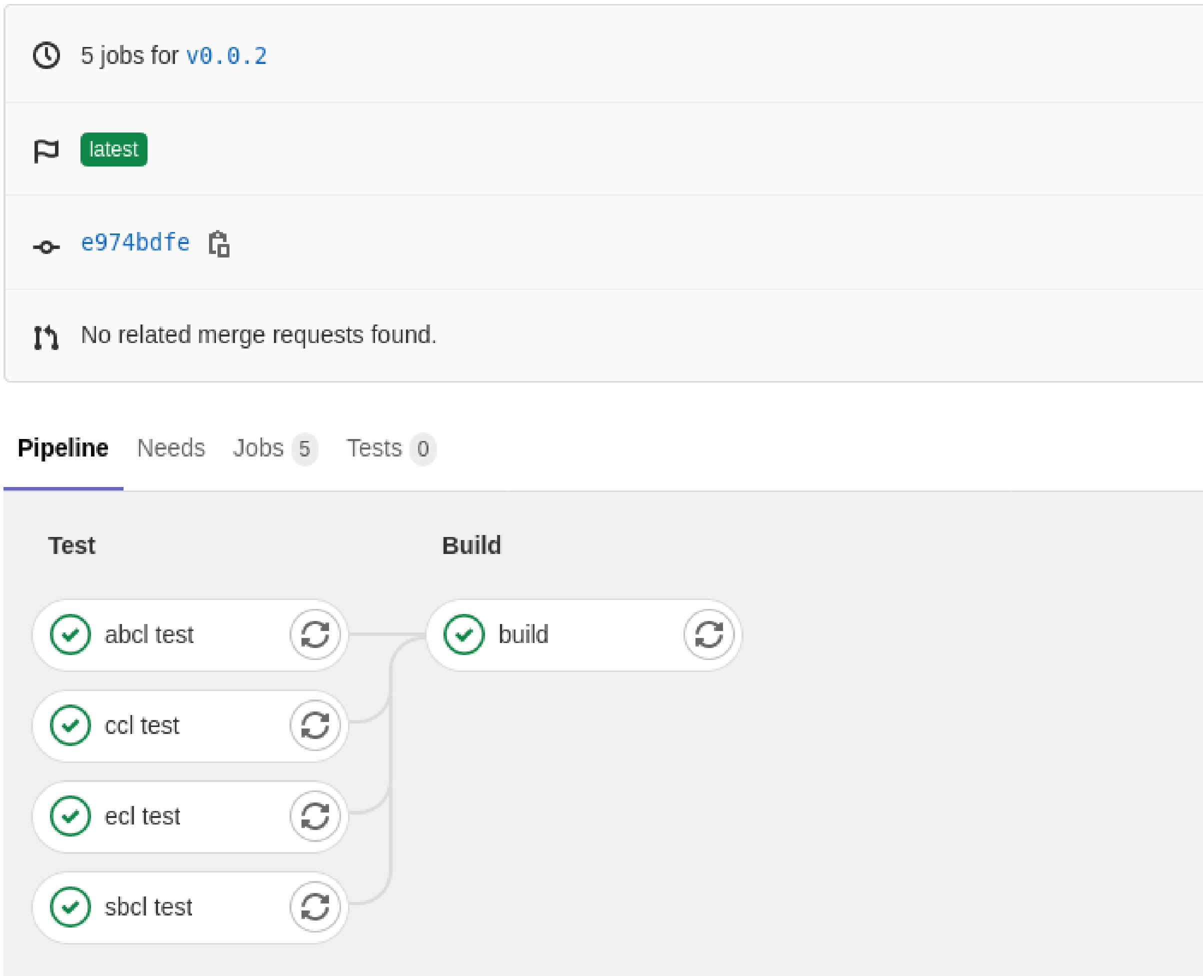
```
    LISP: sbcl

build:
  stage: build
  variables:
    LISP: sbcl
  only:
    - tags
  script:
    - make build
  artifacts:
    paths:
      - some-file-name
```

Here we defined two `stages` (see environments), "test" and "build", defined to run one after another. A "build" stage will start only if the "test" one succeeds.

"build" is asked to run `only` when a new tag is pushed, not at every commit. When it succeeds, it will make the files listed in `artifacts`'s `paths` available for download. We can download them from Gitlab's Pipelines UI, or with an url. This one will download the file "some-file-name" from the latest "build" job:

https://gitlab.com/username/project-name/-/jobs/artifacts/master/raw/some-file-name?job=build

When the pipelines pass, you will see:



You now have a ready to use Gitlab CI.

SourceHut

It's very easy to set up [SourceHut]'s CI system for Common Lisp. Here is a minimal `.build.yml` file that you can test via the [build manifest tester]:

```
image: archlinux
packages:
- sbcl
- quicklisp
sources:
- https://git.sr.ht/~fosskers/cl-transducers
tasks:
# If our project isn't in the special `common-lisp` directory, qu
# be able to find it for loading.
```

```
- move: |
    mkdir common-lisp
    mv cl-transducers ~/common-lisp
- quicklisp: |
    sbcl --non-interactive --load /usr/share/quicklisp/quicklisp
- test: |
    cd common-lisp/cl-transducers
    sbcl --non-interactive --load ~/quicklisp/setup.lisp --load r
```

Since the Docker image we're given is nearly empty, we need to install `sbcl` and `quicklisp` manually. Notice also that we're running a `run-tests.lisp` file to drive the tests. Here's what it could look like:

```
(ql:quickload :transducers/tests)
(in-package :transducers/tests)

(let ((status (parachute:status (parachute:test 'transducers/test
  (cond ((eq :PASSED status) (uiop:quit))
        (t (uiop:quit 1))))
```

Here, examples of the Parachute testing library are shown. As shown elsewhere, in order for the CI job to fail when any test fails, we manually check the test result status and return `1` when there's a problem.

Emacs integration: running tests using Slite

Slite stands for SLIme TEst runner. It allows you to see the summary of test failures, jump to test definitions, rerun tests with the debugger… all from inside Emacs. We get a dashboard-like buffer with green and red badges, from where we can act on tests. It makes the testing process *even more* integrated and interactive.

It consists of an ASDF system and an Emacs package. It is a new project (it appeared mid 2021) so, as of September 2021, neither can be installed via Quicklisp or MELPA yet. Please refer to its repository for instructions.

References

- Tutorial: Working with FiveAM, by Tomek “uint” Kurcz
- Comparison of Common Lisp Testing Frameworks, by Sabra Crolleton.
- the CL Foundation Docker images

See also

- cl-cookieproject, a project skeleton with a FiveAM tests structure.

Database Access and Persistence

The Database section on the Awesome-cl list is a resource listing popular libraries to work with different kind of databases. We can group them roughly in four categories:

- wrappers to one database engine (cl-sqlite, postmodern, cl-redis,…),
- interfaces to several DB engines (clsql, sxql,…),
- persistent object databases (bknr.datastore (see chap. 21 of "Common Lisp Recipes"), ubiquitous,…),
- Object Relational Mappers (Mito),

and other DB-related tools (pgloader).

We'll begin with an overview of Mito. If you must work with an existing DB, you might want to have a look at cl-dbi and clsql. If you don't need a SQL database and want automatic persistence of Lisp objects, you also have a choice of libraries.

The Mito ORM and SxQL

Mito is in Quicklisp:

```
(ql:quickload "mito")
```

Overview

Mito is "an ORM for Common Lisp with migrations, relationships and PostgreSQL support".

- it **supports MySQL, PostgreSQL and SQLite3**,
- when defining a model, it adds an `id` (serial primary key), `created_at` and `updated_at` fields by default like Ruby's ActiveRecord or Django,
- handles DB **migrations** for the supported backends,
- permits DB **schema versioning**,

- is tested under SBCL and CCL.

As an ORM, it allows to write class definitions, to specify relationships, and provides functions to query the database. For custom queries, it relies on SxQL, an SQL generator that provides the same interface for several backends.

Working with Mito generally involves these steps:

- connecting to the DB
- writing CLOS classes to define models
- running migrations to create or alter tables
- creating objects, saving same in the DB,

and iterating.

Connecting to a DB

Mito provides the function `connect-toplevel` to establish a connection to RDBMs:

```
(mito:connect-toplevel :mysql
                       :database-name "myapp"
                       :username "fukamachi"
                       :password "c0mon-1isp")
```

The driver type can be of `:mysql`, `:sqlite3` and `:postgres`.

With sqlite you don’t need the username and password:

```
(mito:connect-toplevel :sqlite3 :database-name "myapp")
```

As usual, you need to create the MySQL or PostgreSQL database beforehand. Refer to their documentation.

Connecting sets `mito:*connection*` to the new connection and returns it.

Disconnect with `disconnect-toplevel`.

You might make good use of a wrapper function:

```
(defun connect ()
  "Connect to the DB."
  (mito:connect-toplevel :sqlite3 :database-name "myapp"))
```

Models

Defining models

In Mito, you can define a class which corresponds to a database table with the `deftable` macro:

```
(mito:deftable user ()
  ((name :col-type (:varchar 64))
   (email :col-type (or (:varchar 128) :null))))
```

Alternatively, you can specify `(:metaclass mito:dao-table-class)` in a regular class definition.

The `deftable` macro automatically adds some slots: a primary key named `id` if there's no primary key, and `created_at` and `updated_at` for recording timestamps. Specifying `(:auto-pk nil)` and `(:record-timestamps nil)` in the `deftable` form will disable these behaviours. A `deftable` class will also come with initializers, named after the slot, and accessors, of form `<class-name>-<slot-name>`, for each named slot. For example, for the `name` slot in the above table definition, the initarg `:name` will be added to the constuctor, and the accessor `user-name` will be created.

You can inspect the new class:

```
(mito.class:table-column-slots (find-class 'user))
;=> (#<MITO.DAO.COLUMN:DAO-TABLE-COLUMN-CLASS MITO.DAO.MIXIN::ID
;    #<MITO.DAO.COLUMN:DAO-TABLE-COLUMN-CLASS COMMON-LISP-USER::
;    #<MITO.DAO.COLUMN:DAO-TABLE-COLUMN-CLASS COMMON-LISP-USER::
;    #<MITO.DAO.COLUMN:DAO-TABLE-COLUMN-CLASS MITO.DAO.MIXIN::CR
;    #<MITO.DAO.COLUMN:DAO-TABLE-COLUMN-CLASS MITO.DAO.MIXIN::UP
```

The class inherits `mito:dao-class` implicitly.

```
(find-class 'user)
;=> #<MITO.DAO.TABLE:DAO-TABLE-CLASS COMMON-LISP-USER::USER>

(c2mop:class-direct-superclasses *)
;=> (#<STANDARD-CLASS MITO.DAO.TABLE:DAO-CLASS>)
```

This may be useful when you define methods which can be applied for all table classes.

For more information on using the Common Lisp Object System, see the clos page.

Creating the tables

After defining the models, you must create the tables:

```
(mito:ensure-table-exists 'user)
```

So a helper function:

```
(defun ensure-tables ()
  (mapcar #'mito:ensure-table-exists '(user foo bar)))
```

See Mito's documentation for a couple more ways.

When you alter the model you'll need to run a DB migration, see the next section.

Fields

Fields types

Field types are:

(:varchar <integer>), text,

:serial, :bigserial, :integer, :bigint, :unsigned,

`:timestamp`, `:timestamptz`,

`:bytea`,

Optional fields

Use `(or <real type> :null)`:

```
   (email :col-type (or (:varchar 128) :null))
```

Field constraints

`:unique-keys` can be used like so:

```
(mito:deftable user ()
  ((name :col-type (:varchar 64))
   (email :col-type (:varchar 128))
  (:unique-keys email))
```

We already saw `:primary-key`.

You can change the table name with `:table-name`.

Relationships

You can define a relationship by specifying a foreign class with `:col-type`:

```
(mito:deftable tweet ()
  ((status :col-type :text)
   ;; This slot refers to USER class
   (user :col-type user))

(table-definition (find-class 'tweet))
;=> (#<SXQL-STATEMENT: CREATE TABLE tweet (
;        id BIGSERIAL NOT NULL PRIMARY KEY,
;        status TEXT NOT NULL,
;        user_id BIGINT NOT NULL,
;        created_at TIMESTAMP,
;        updated_at TIMESTAMP
;    )>)
```

Now you can create or retrieve a `TWEET` by a `USER` object, not a `USER-ID`.

```
(defvar *user* (mito:create-dao 'user :name "Eitaro Fukamachi"))
(mito:create-dao 'tweet :user *user*)

(mito:find-dao 'tweet :user *user*)
```

Mito doesn't add foreign key constraints for referring tables.

One-to-one

A one-to-one relationship is simply represented with a simple foreign key on a slot (as `:col-type user` in the `tweet` class). Besides, we can add a unicity constraint, as with `(:unique-keys email)`.

One-to-many, many-to-one

The tweet example above shows a one-to-many relationship between a user and his tweets: a user can write many tweets, and a tweet belongs to only one user.

The relationship is defined with a foreign key on the "many" side linking back to the "one" side. Here the `tweet` class defines a `user` foreign key, so a tweet can only have one user. You didn't need to edit the `user` class.

A many-to-one relationship is actually the contrary of a one-to-many. You have to put the foreign key on the appropriate side.

Many-to-many

A many-to-many relationship needs an intermediate table, which will be the "many" side for the two tables it is the intermediary of.

And, thanks to the join table, we can store more information about the relationship.

Let's define a `book` class:

```
(mito:deftable book ()
    ((title :col-type (:varchar 128))
     (ean :col-type (or (:varchar 128) :null))))
```

A user can have many books, and a book (as the title, not the physical copy) is likely to be in many people's library. Here's the intermediate class:

```
(mito:deftable user-books ()
    ((user :col-type user)
     (book :col-type book)))
```

Each time we want to add a book to a user's collection (say in a `add-book` function), we create a new `user-books` object.

But someone may very well own many copies of one book. This is an information we can store in the join table:

```
(mito:deftable user-books ()
    ((user :col-type user)
     (book :col-type book)
    ;; Set the quantity, 1 by default:
     (quantity :col-type :integer)))
```

Inheritance and mixin

A subclass of DAO-CLASS is allowed to be inherited. This may be useful when you need classes which have similar columns:

```
(mito:deftable user ()
  ((name :col-type (:varchar 64))
   (email :col-type (:varchar 128)))
  (:unique-keys email))

(mito:deftable temporary-user (user)
  ((registered-at :col-type :timestamp)))

(mito:table-definition 'temporary-user)
;=> (#<SXQL-STATEMENT: CREATE TABLE temporary_user (
;        id BIGSERIAL NOT NULL PRIMARY KEY,
;        name VARCHAR(64) NOT NULL,
```

```
;         email VARCHAR(128) NOT NULL,
;         registered_at TIMESTAMP NOT NULL,
;         created_at TIMESTAMP,
;         updated_at TIMESTAMP,
;         UNIQUE (email)
;     )>)
```

If you need a ‘template’ for tables which aren’t related to any database tables, you can use `DAO-TABLE-MIXIN` in a `defclass` form. The `has-email` class below will not create a table.

```
(defclass has-email ()
  ((email :col-type (:varchar 128)
          :initarg :email
          :accessor object-email))
  (:metaclass mito:dao-table-mixin)
  (:unique-keys email))
;=> #<MITO.DAO.MIXIN:DAO-TABLE-MIXIN COMMON-LISP-USER::HAS-EMAIL

(mito:deftable user (has-email)
  ((name :col-type (:varchar 64))))
;=> #<MITO.DAO.TABLE:DAO-TABLE-CLASS COMMON-LISP-USER::USER>

(mito:table-definition 'user)
;=> (#<SXQL-STATEMENT: CREATE TABLE user (
;        id BIGSERIAL NOT NULL PRIMARY KEY,
;        name VARCHAR(64) NOT NULL,
;        email VARCHAR(128) NOT NULL,
;        created_at TIMESTAMP,
;        updated_at TIMESTAMP,
;        UNIQUE (email)
;    )>)
```

See more examples of use in mito-auth.

Troubleshooting

“Cannot CHANGE-CLASS objects into CLASS metaobjects.”

If you get the following error message:

```
Cannot CHANGE-CLASS objects into CLASS metaobjects.
       [Condition   of   type   SB-PCL::METAOBJECT-INITIALIZATION-
VIOLATION]
See also:
  The Art of the Metaobject Protocol, CLASS [:initialization]
```

it is certainly because you first wrote a class definition and *then* added the Mito metaclass and tried to evaluate the class definition again.

If this happens, you must remove the class definition from the current package:

```
(setf (find-class 'foo) nil)
```

or, with the Slime inspector, click on the class and find the "remove" button.

More info here.

Migrations

We can run database migrations manually, as shown below, or we can automatically run migrations after a change to the model definitions. To enable automatic migrations, set `mito:*auto-migration-mode*` to `t`.

The first step is to create the tables, if needed:

```
(ensure-table-exists 'user)
```

then alter the tables:

```
(mito:migrate-table 'user)
```

You can check the SQL generated code with `migration-expressions 'class`. For example, we create the `user` table:

```
(ensure-table-exists 'user)
;-> ;; CREATE TABLE IF NOT EXISTS "user" (
```

```
;       "id" BIGSERIAL NOT NULL PRIMARY KEY,
;       "name" VARCHAR(64) NOT NULL,
;       "email" VARCHAR(128),
;       "created_at" TIMESTAMP,
;       "updated_at" TIMESTAMP
;   ) () [0 rows] | MITO.DAO:ENSURE-TABLE-EXISTS
```

There are no changes from the previous user definition:

```
(mito:migration-expressions 'user)
;=> NIL
```

Now let's add a unique `email` field:

```
(mito:deftable user ()
  ((name :col-type (:varchar 64))
   (email :col-type (:varchar 128)))
  (:unique-keys email))
```

The migration will run the following code:

```
(mito:migration-expressions 'user)
;=> (#<SXQL-STATEMENT: ALTER TABLE user ALTER COLUMN email TYPE
;    #<SXQL-STATEMENT: CREATE UNIQUE INDEX unique_user_email ON
```

so let's apply it:

```
(mito:migrate-table 'user)
;-> ;; ALTER TABLE "user" ALTER COLUMN "email" TYPE character va
;   ;; CREATE UNIQUE INDEX "unique_user_email" ON "user" ("email
;-> (#<SXQL-STATEMENT: ALTER TABLE user ALTER COLUMN email TYPE
;    #<SXQL-STATEMENT: CREATE UNIQUE INDEX unique_user_email ON
```

Queries

Creating objects

We can create user objects with the regular `make-instance`:

```
(defvar me
```

```
  (make-instance 'user :name "Eitaro Fukamachi" :email "e.arrows
;=> USER
```

To save it in DB, use `insert-dao`:

```
(mito:insert-dao me)
;-> ;; INSERT INTO `user` (`name`, `email`, `created_at`, `updat
;=> #<USER {10053C4453}>
```

Do the two steps above at once:

```
(mito:create-dao 'user :name "Eitaro Fukamachi" :email "e.arrows
```

You should not export the `user` class and create objects outside of its package (it is good practice anyway to keep all database-related operations in say a `models` package and file). You should instead use a helper function:

```
(defun make-user (&key name)
  (make-instance 'user :name name))
```

Updating fields

```
(setf (slot-value me 'name) "nitro_idiot")
;=> "nitro_idiot"
```

and save it:

```
(mito:save-dao me)
```

Deleting

```
(mito:delete-dao me)
;-> ;; DELETE FROM `user` WHERE (`id` = ?) (1) [0 rows] | MITO.D

;; or:
(mito:delete-by-values 'user :id 1)
;-> ;; DELETE FROM `user` WHERE (`id` = ?) (1) [0 rows] | MITO.D
```

Get the primary key value

```
(mito:object-id me)
;=> 1
```

Count

```
(mito:count-dao 'user)
;=> 1
```

Find one

```
(mito:find-dao 'user :id 1)
;-> ;; SELECT * FROM `user` WHERE (`id` = ?) LIMIT 1 (1) [1 row]
;=> #<USER {10077C6073}>
```

So here's a possibility of generic helpers to find an object by a given key:

```
(defgeneric find-user (key-name key-value)
  (:documentation "Retrieves an user from the data base by one c
keys."))

(defmethod find-user ((key-name (eql :id)) (key-value integer))
  (mito:find-dao 'user key-value))

(defmethod find-user ((key-name (eql :name)) (key-value string))
  (first (mito:select-dao 'user
                        (sxql:where (:= :name key-value)))))
```

Find all

Use the macro `select-dao`.

Get a list of all users:

```
(mito:select-dao 'user)
;(#<USER {10077C6073}>)
;#<SXQL-STATEMENT: SELECT * FROM user>
```

Find by relationship

As seen above:

```
(mito:find-dao 'tweet :user *user*)
```

Custom queries

It is with `select-dao` that you can write more precise queries by giving it SxQL statements.

Example:

```
(select-dao 'tweet
    (where (:like :status "%Japan%")))
```

another:

```
(select (:id :name :sex)
  (from (:as :person :p))
  (where (:and (:>= :age 18)
               (:< :age 65)))
  (order-by (:desc :age)))
```

You can compose your queries with regular Lisp code:

```
(defun find-tweets (&key user)
  (select-dao 'tweet
    (when user
      (where (:= :user user)))
    (order-by :object-created)))
```

`select-dao` is a macro that expands to the right thing©.

Note: if you didn't `use` SXQL, then write `(sxql:where …)` and `(sxql:order-by …)`.

You can compose your queries further with the backquote syntax.

Imagine you receive a `query` string, maybe composed of space-separated words, and you want to search for books that have either one of these words in their title or in their author's name. Searching for "bob adventure" would return a book that has "adventure" in its title and "bob" in its author name, or both in the title.

For the example sake, an author is a string, not a link to another table:

```
(mito:deftable book ()
    ((title :col-type (:varchar 128))
     (author :col-type (:varchar 128))
     (ean :col-type (or (:varchar 128) :null))))
```

You want to add a clause that searches on both fields for each word.

```
(defun find-books (&key query (order :desc))
  "Return a list of books.
If a query string is given, search on both the title
and the author fields."
  (mito:select-dao 'book
    (when (str:non-blank-string-p query)
      (sxql:where
       `(:and
         ,@(loop for word in (str:words query)
              :collect `(:or (:like :title
                                    ,(str:concat "%" word "%"))
                             (:like :authors
                                    ,(str:concat "%" word "%")))
       (sxql:order-by `(,order :created-at))))
```

By the way, we are still using a `LIKE` statement, but with a non-small dataset you'll want to use your database's full text search engine.

Clauses

See the SxQL documentation.

Examples:

```
(select-dao 'foo
  (where (:and (:> :age 20) (:<= :age 65))))
```

```
(order-by :age (:desc :id))
```

```
(group-by :sex)
```

```
(having (:>= (:sum :hoge) 88))
```

```
(limit 0 10)
```

and `joins`, etc.

Operators

```
:not
:is-null, :not-null
:asc, :desc
:distinct
:=, :!=
:<, :>, :<= :>=
:a<, :a>
:as
:in, :not-in
:like
:and, :or
:+, :-, :* :/ :%
:raw
```

Triggers

Since `insert-dao`, `update-dao` and `delete-dao` are defined as generic functions, you can define `:before`, `:after` or `:around` methods to those, like regular method combination.

```
(defmethod mito:insert-dao :before ((object user))
  (format t "~&Adding ~S...~%" (user-name object)))
```

```
(mito:create-dao 'user :name "Eitaro Fukamachi" :email "e.arrows
```

```
;-> Adding "Eitaro Fukamachi"...
;   ;; INSERT INTO "user" ("name", "email", "created_at", "updat
;=> #<USER {100835FB33}>
```

Inflation/Deflation

Inflation/Deflation is a function to convert values between Mito and RDBMS.

```
(mito:deftable user-report ()
  ((title :col-type (:varchar 100))
   (body :col-type :text
         :initform "")
   (reported-at :col-type :timestamp
                :initform (local-time:now)
                :inflate #'local-time:universal-to-timestamp
                :deflate #'local-time:timestamp-to-universal)))
```

Eager loading

One of the pains in the neck to use ORMs is the “N+1 query” problem.

```
;; BAD EXAMPLE

(use-package '(:mito :sxql))

(defvar *tweets-contain-japan*
  (select-dao 'tweet
    (where (:like :status "%Japan%"))))

;; Getting names of tweeted users.
(mapcar (lambda (tweet)
          (user-name (tweet-user tweet)))
        *tweets-contain-japan*)
```

This example sends a query to retrieve a user like “SELECT * FROM user WHERE id = ?” at each iteration.

To prevent this performance issue, add `includes` to the above query which only sends a single WHERE IN query instead of N queries:

```
;; GOOD EXAMPLE with eager loading

(use-package '(:mito :sxql))

(defvar *tweets-contain-japan*
  (select-dao 'tweet
    (includes 'user)
    (where (:like :status "%Japan%"))))
;-> ;; SELECT * FROM `tweet` WHERE (`status` LIKE ?) ("%Japan%")
;-> ;; SELECT * FROM `user` WHERE (`id` IN (?, ?, ?)) (1, 3, 12)
;=> (#<TWEET {1003513EC3}> #<TWEET {1007BABEF3}> #<TWEET {1007BE


;; No additional SQLs will be executed.
(tweet-user (first *))
;=> #<USER {100361E813}>
```

Schema versioning

```
$ ros install mito
$ mito
Usage: mito command [option...]

Commands:
    generate-migrations
    migrate

Options:
     -t, --type DRIVER-TYPE            DBI driver type (one of
"mysql", "postgres" or "sqlite3")
    -d, --database DATABASE-NAME    Database name to use
    -u, --username USERNAME         Username for RDBMS
    -p, --password PASSWORD         Password for RDBMS
      -s, --system SYSTEM                 ASDF system to load
(several -s's allowed)
     -D, --directory DIRECTORY         Directory path to keep
migration            SQL            files            (default:
"/Users/nitro_idiot/Programs/lib/mito/db/")
     --dry-run                          List SQL expressions to
migrate
```

Introspection

Mito provides some functions for introspection.

We can access the information of **columns** with the functions in `(mito.class.column:...)`:

- `table-column-[class, name, info, not-null-p,...]`
- `primary-key-p`

and likewise for **tables** with `(mito.class.table:...)`.

Given we get a list of slots of our class:

```
(ql:quickload "closer-mop")

(closer-mop:class-direct-slots (find-class 'user))
;; (#<MITO.DAO.COLUMN:DAO-TABLE-COLUMN-CLASS NAME>
;;  #<MITO.DAO.COLUMN:DAO-TABLE-COLUMN-CLASS EMAIL>)

(defparameter user-slots *)
```

We can answer the following questions:

What is the type of this column ?

```
(mito.class.column:table-column-type (first user-slots))
;; (:VARCHAR 64)
```

Is this column nullable ?

```
(mito.class.column:table-column-not-null-p
  (first user-slots))
;; T
(mito.class.column:table-column-not-null-p
  (second user-slots))
;; NIL
```

Testing

We don’t want to test DB operations against the production one. We need to create a temporary DB before each test.

The macro below creates a temporary DB with a random name, creates the tables, runs the code and connects back to the original DB connection.

```
(defpackage my-test.utils
  (:use :cl)
  (:import-from :my.models
                :*db*
                :*db-name*
                :connect
                :ensure-tables-exist
                :migrate-all)
  (:export :with-empty-db))

(in-package my-test.utils)

(defun random-string (length)
  ;; thanks 40ants/hacrm.
  (let ((chars "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuv
    (coerce (loop repeat length
                  collect (aref chars (random (length chars))))
            'string)))

(defmacro with-empty-db (&body body)
  "Run `body` with a new temporary DB."
  `(let* ((*random-state* (make-random-state t))
          (prefix (concatenate 'string
                               (random-string 8)
                               "/"))
          ;; Save our current DB connection.
          (connection mito:*connection*))
     (uiop:with-temporary-file (:pathname name :prefix prefix)
       ;; Bind our *db-name* to a new name, so as to create a ne
       (let* ((*db-name* name))
         ;; Always re-connect to our real DB even in case of
         ;; error in body.
         (unwind-protect
           (progn
             ;; our functions to connect to the DB, create the t
             ;; and run the migrations.
```

```
            (connect)
            (ensure-tables-exist)
            (migrate-all)
            ,@body)

          (setf mito:*connection* connection))))))
```

Use it like this:

```
(prove:subtest "Creation in a temporary DB."
  (with-empty-db
    (let ((user (make-user :name "Cookbook")))
      (save-user user)

      (prove:is (name user)
                "Cookbook"
                "Test username in a temp DB."))))
;; Creation in a temporary DB
;;  CREATE TABLE "user" (
;;       id BIGSERIAL NOT NULL PRIMARY KEY,
;;       name VARCHAR(64) NOT NULL,
;;       email VARCHAR(128) NOT NULL,
;;       created_at TIMESTAMP,
;;       updated_at TIMESTAMP,
;;       UNIQUE (email)
;; ) () [0 rows] | MITO.DB:EXECUTE-SQL
;; ✓ Test username in a temp DB.
```

See also

- exploring an existing (PostgreSQL) database with postmodern
- mito-attachment
- mito-auth
- can a role-based access right control library

GUI toolkits

Lisp has a long and rich history and so does the development of Graphical User Interfaces in Lisp. In fact, the first GUI builder was written in Lisp (and sold to Apple. It is now Interface Builder).

Lisp is also famous and unrivalled for its interactive development capabilities, a feature even more worth having to develop GUI applications. Can you imagine compiling one function and seeing your GUI update instantly? We can do this with many GUI frameworks today, even though the details differ from one to another.

Finally, a key part in building software is how to build it and ship it to users. Here also, we can build self-contained binaries, for the three main operating systems, that users can run with a double click.

We aim here to give you the relevant information to help you choose the right GUI framework and to put you on tracks. Don't hesitate to contribute, to send more examples and to furnish the upstream documentations.

Introduction

In this recipe, we'll present the following GUI toolkits:

- Tk with Ltk and nodgui
- Qt4 with Qtools
- IUP with lispnik/iup
- Gtk3 with cl-cffi-gtk
  - if you want Gtk4 bindings, see cl-gtk4. They are new bindings, released in September, 2022.
- Nuklear with Bodge-Nuklear

In addition, you might want to have a look to:

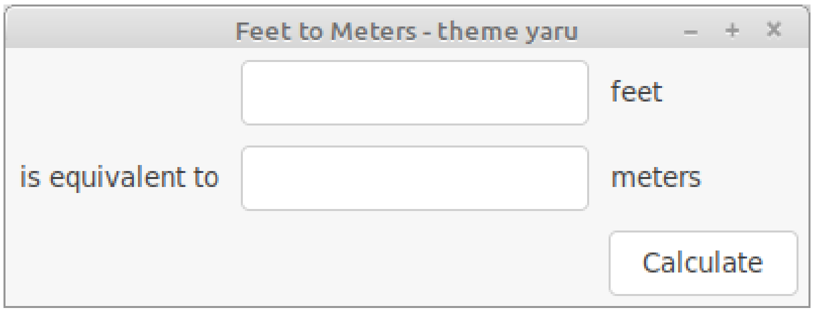
- the CAPI toolkit (Common Application Programming Interface), which is proprietary and made by LispWorks. It is a complete and cross-platform toolkit (Windows, Gtk+, Cocoa), very praised by its users. LispWorks also has iOS and Android runtimes. Example software built with CAPI include ScoreCloud. It is possible to try it with the LispWorks free demo.
- Allegro CL's IDE and Common Graphics windowing system (proprietary): Allegro's IDE is a general environment for developing applications. It works in concert with a windowing system called Common Graphics. The IDE is available for Allegro CL's Microsoft Windows, on Linux platforms, Free BSD and on the Mac.
    - NEW! 🎉 since Allegro CL 10.1 (released in March of 2022), the IDE, and the Common Graphics GUI toolkit, runs in the browser. It is called CG/JS.
- CCL's built-in Cocoa interface, used to build applications such as Opusmodus.
- Clozure CL's built-in Objective-C bridge and CocoaInterface, a Cocoa interface for CCL. Build Cocoa user interface windows dynamically using Lisp code and bypass the typical Xcode processes.
    - the bridge is good at catching ObjC errors and turning them into Lisp errors, so one can have an iterative REPL-based development cycle for a macOS GUI application.
- McCLIM and Garnet are toolkit in 100% Common Lisp. McClim even has a prototype running in the browser with the Broadway protocol and Garnet has an ongoing interface to Gtk.
- Alloy, another very new toolkit in 100% Common Lisp, used for example in the Kandria game.
- eql, eql5, eql5-android, embedded Qt4 and Qt5 Lisp, embedded in ECL, embeddable in Qt. Port of EQL5 to the Android platform.
- this demo using Java Swing from ABCL
- examples of using Gtk without C files with SBCL, as well as GTK-server.
- and, last but not least, Ceramic, to ship a cross-platform web app with Electron.

as well as the other ones listed on awesome-cl#gui and Cliki.

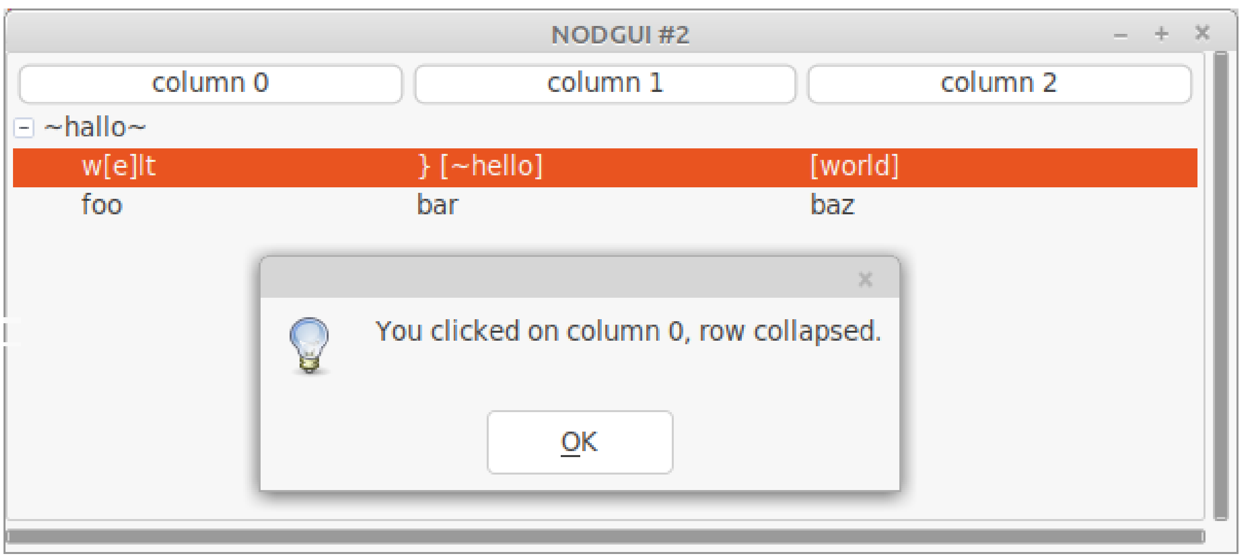
Tk (Ltk and nodgui)

Tk (or Tcl/Tk, where Tcl is the programming language) has the infamous reputation of having an outdated look. This is not (so) true anymore since its version 8 of 1997 (!). It is probably better than you think.

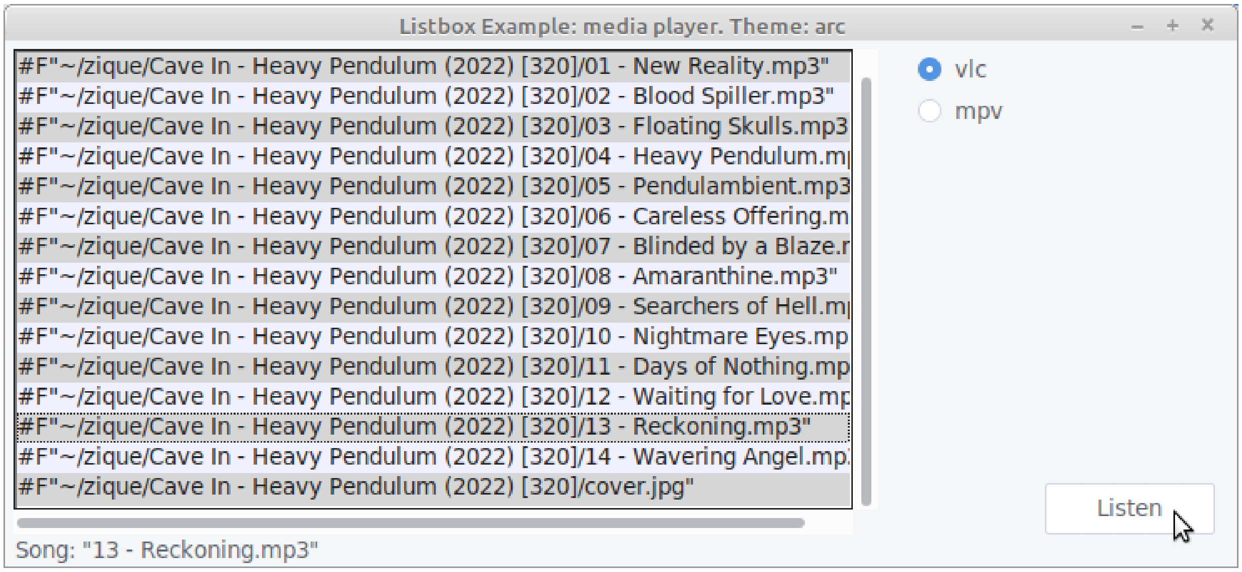
This is a simple GUI with nodgui's built-in theme (more on that below):



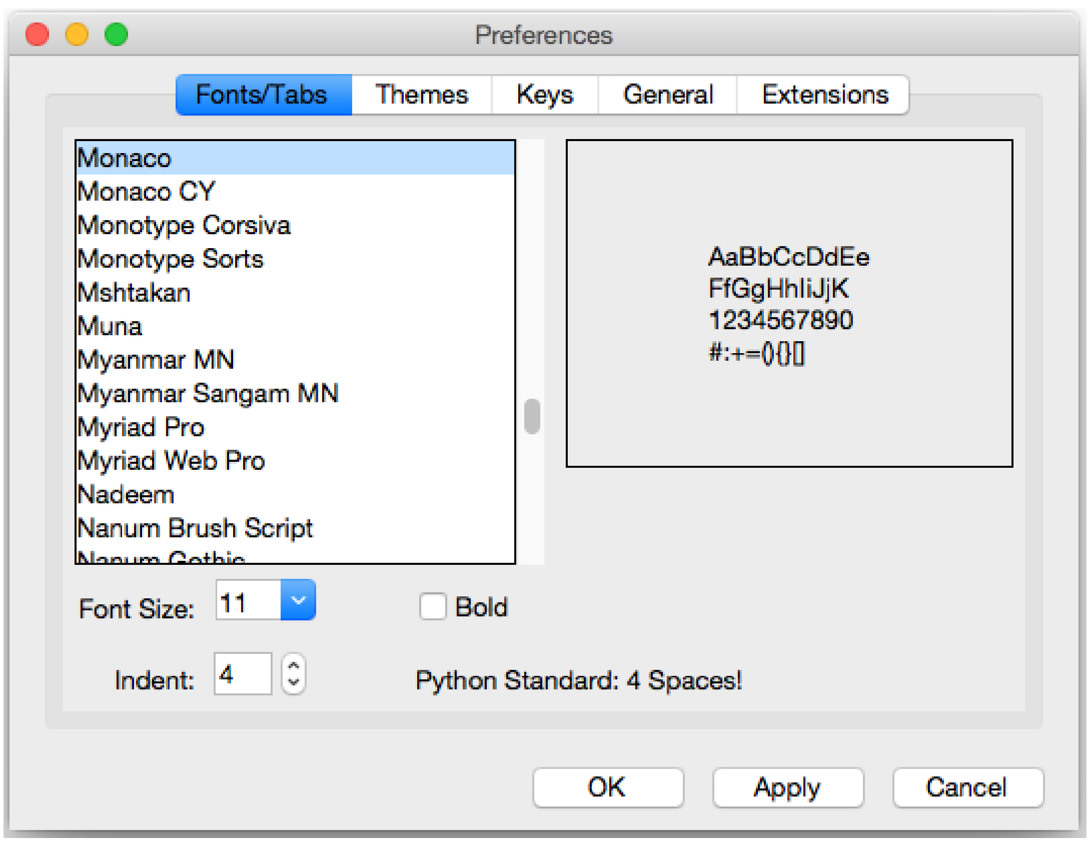
This is a treeview, with the same theme:



A toy mediaplayer, showing a tree list, checkboxes, buttons and labels, with the Arc theme:



This is a demo with a Macos theme:



In addition to those, we can use many of the ttkthemes, the Forest theme, and more. See this tcl/tk list.

But what is Tk good for? Tk doesn't have a great choice of widgets, but it has a useful canvas, and it has a couple of unique features: we can develop a graphical interface **fully interactively** and we can run the GUI **remotely** from the core app. It is also cross-platform.

So, Tk isn't native and doesn't have the most advanced features, but it is a used and proven GUI toolkit (and programming language) still used in the industry. It can be a great choice to quickly create simple GUIs, to leverage its ease of deployment, or when stability is required.

There are two Lisp bindings: Ltk and nodgui. Nodgui ("No Drama GUI") is a fork of Ltk, with added widgets (such as an auto-completion list widget), an asynchronous event loop and, what we really enjoy, the surprisingly nice-looking "Yaru" theme that comes with the library. It is also very easy to install and use any other theme of our choice, see below.

- **Tk is Written in**: Tcl
- **Portability**: cross-platform (Windows, macOS, Linux).
- **Widgets**: this is not the fort of Tk. It has a **small set** of default widgets, and misses important ones, for example a date picker. We can find some in extensions (such as in **Nodgui**), but they don't feel native, at all. The calendar is brought by a Tk extension and looks better.
- **Interactive development**: very much.
- **Graphical builder**: no
- **Other features**:
    - **remote execution**: the connection between Lisp and Tcl/Tk is done via a stream. It is thus possible to run the Lisp program on one computer, and to display the GUI on another one. The only thing required on the client computer is tcl/tk installed and the remote.tcl script. See Ltk-remote.
- **Bindings documentation**: short but complete. Nodgui too.
- **Bindings stability**: very stable
- **Bindings activity**: low for Ltk (mostly maintenance), active for nodgui (new features).

- **Licence**: Tcl/Tk is BSD-style, Ltk is LGPL.
- Example applications:
  - Fulci - a program to organise your movie collections.
  - Ltk small games - snake and tic-tac-toe.
  - cl-pkr - a cross-platform color picker.
  - cl-torrents - searching torrents on popular trackers. CLI, readline and a simple Tk GUI.
- More examples:
  - https://peterlane.netlify.app/ltk-examples/: LTk examples for the tkdocs tutorial.
  - LTk Plotchart - A wrapper around the tklib/plotchart library to work with LTk. This includes over 20 different chart types (xy-plots, gantt charts, 3d-bar charts etc…).

List of widgets

(please don't suppose the list is exhaustive)

```
Button Canvas Check-button Entry Frame Label Labelframe Listbox
Menu Menubutton Message
Paned-window
Radio-button Scale
Scrollbar Spinbox Text
Toplevel Widget Canvas

Ltk-megawidgets:
    progress
    history-entry
    menu-entry
```

nodgui adds:

```
treelist   tooltip   searchable-listbox   date-picker   calendar
autocomplete-listbox
password-entry progress-bar-star notify-window
dot-plot bar-chart equalizer-bar
swap-list
```

Qt4 (Qtools)

Do we need to present Qt and Qt4? Qt is huge and contains everything and the kitchen sink. Qt not only provides UI widgets, but numerous other layers (networking, D-BUS…).

Qt is free for open-source software, however you'll want to check the conditions to ship proprietary ones.

The Qtools bindings target Qt4. The Qt5 Lisp bindings are https://github.com/commonqt/commonqt5/ and not ready for prime time..

A companion library for Qtools, that you'll want to check out once you made your first Qtool application, is Qtools-ui, a collection of useful widgets and pre-made components. It comes with short demonstrations videos.

- **Framework written in**: C++
- **Framework Portability**: multi-platform, Android, embedded systems, WASM.

- **Bindings Portability**: Qtools runs on x86 desktop platforms on Windows, macOS and GNU/Linux.

- **Widgets choice**: large.

- **Graphical builder**: yes.

- **Other features**: Web browser, a lot more.

- **Bindings documentation**: lengthy explanations, a few examples. Prior Qt knowledge is required.
- **Bindings stability**: stable
- **Bindings activity**: active
- **Qt Licence**: both commercial and open source licences.
- Example applications:
  - https://github.com/Shinmera/qtools/tree/master/examples
  - https://github.com/Shirakumo/lionchat
  - https://github.com/shinmera/halftone - a simple image viewer

Gtk+3 (cl-cffi-gtk)

Gtk+3 is the primary library used to build GNOME applications. Its (currently most advanced) lisp bindings is cl-cffi-gtk. While primarily created for GNU/Linux, Gtk works fine under macOS and can now also be used on Windows.

- **Framework written in**: C

- **Portability**: GNU/Linux and macOS, also Windows.

- **Widgets choice**: large.

- **Graphical builder**: yes: Glade.

- **Other features**: web browser (WebKitGTK)

- **Bindings documentation**: very good: http://www.crategus.com/books/cl-gtk/gtk-tutorial.html
- **Bindings stability**: stable
- **Bindings activity**: low activity, active development.
- **Licence**: LGPL
- Example applications:
  - an Atmosphere Calculator, built with Glade.
- more documentation and examples:
  - Learn Common Lisp by Example: GTK GUI with SBCL

IUP (lispnik/IUP)

IUP is a cross-platform GUI toolkit actively developed at the PUC university of Rio de Janeiro, Brazil. It uses **native controls**: the Windows API for Windows, Gtk3 for GNU/Linux. At the time of writing, it has a Cocoa port in the works (as well as iOS, Android and WASM ones). A particularity of IUP is its **small API**.

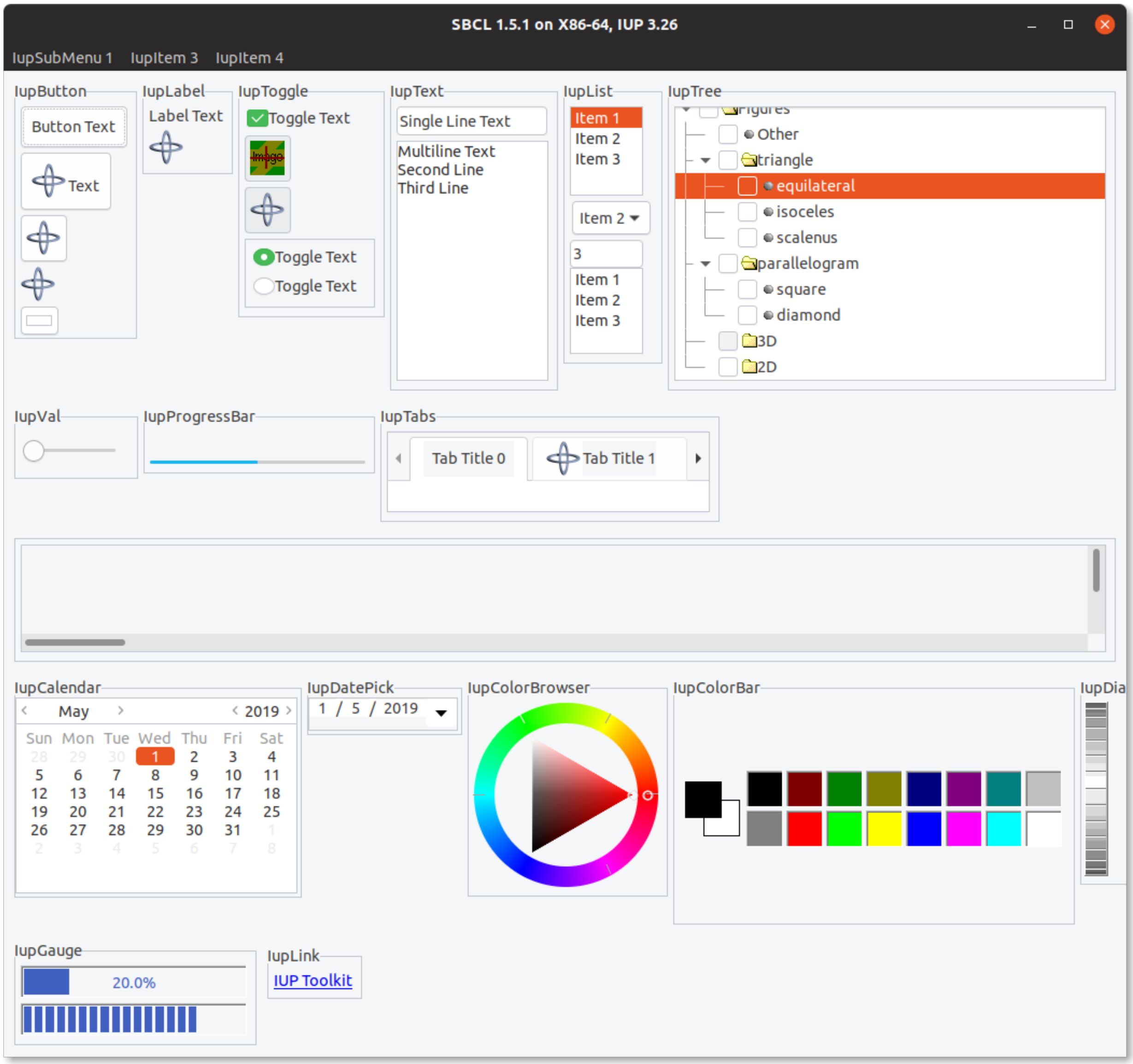
The Lisp bindings are lispnik/iup. They are nicely done in that they are automatically generated from the C sources. They can follow new IUP versions with a minimal work and the required steps are documented. All this gives us good guarantee over the bus factor.

IUP stands as a great solution in between Tk and Gtk or Qt.

- **Framework written in**: C (official API also in Lua and LED)
- **Portability**: Windows and Linux, work started for Cocoa, iOS, Android, WASM.
- **Widgets choice**: medium. Includes a web browser window (WebkitGTK on Linux, IE's WebBrowser on Windows).
- **Graphical builder**: yes: IupVisualLED
- **Other features**: OpenGL, Web browser (WebKitGTK on GNU/Linux), plotting, Scintilla text editor
- **Bindings documentation**: good examples and good readme, otherwise low.
- **Bindings stability**: alpha (but fully generated and working nicely).
- **Bindings activity**: low but steady, and reactive to new IUP versions.
- **Licence**: IUP and the bindings are MIT licenced.

List of widgets

```
Radio, Tabs, FlatTabs, ScrollBox, DetachBox,
Button, FlatButton, DropButton, Calendar, Canvas, Colorbar,
ColorBrowser, DatePick, Dial, Gauge, Label, FlatLabel,
FlatSeparator, Link, List, FlatList, ProgressBar, Spin, Text,
Toggle, Tree, Val,
listDialog, Alarm, Color, Message, Font, Scintilla, file-dialog…
Cells, Matrix, MatrixEx, MatrixList,
GLCanvas, Plot, MglPlot, OleControl, WebBrowser (WebKit/Gtk+)…
drag-and-drop
WebBrowser
```



Nuklear (Bodge-Nuklear)

[Nuklear](#) is a small [immediate-mode](#) GUI toolkit:

> [Nuklear](#) is a minimal-state, immediate-mode graphical user interface toolkit written in ANSI C and licensed under public domain. It was designed as a simple embeddable user interface for application and does not have any dependencies, a default render backend or OS window/input handling but instead provides a highly modular, library-based approach, with simple input state for input and draw commands describing primitive shapes as output. So instead of providing a layered

library that tries to abstract over a number of platform and render backends, it focuses only on the actual UI.

its Lisp binding is Bodge-Nuklear, and its higher level companions bodge-ui and bodge-ui-window.

Unlike traditional UI frameworks, Nuklear allows the developer to take over the rendering loop or the input management. This might require more setup, but it makes Nuklear particularly well suited for games, or for applications where you want to create new controls.

- **Framework written in**: ANSI C, single-header library.

- **Portability**: where C runs. Nuklear doesn't contain platform-specific code. No direct OS or window handling is done in Nuklear. Instead *all input state has to be provided by platform specific code*.

- **Widgets choice**: small.

- **Graphical builder**: no.

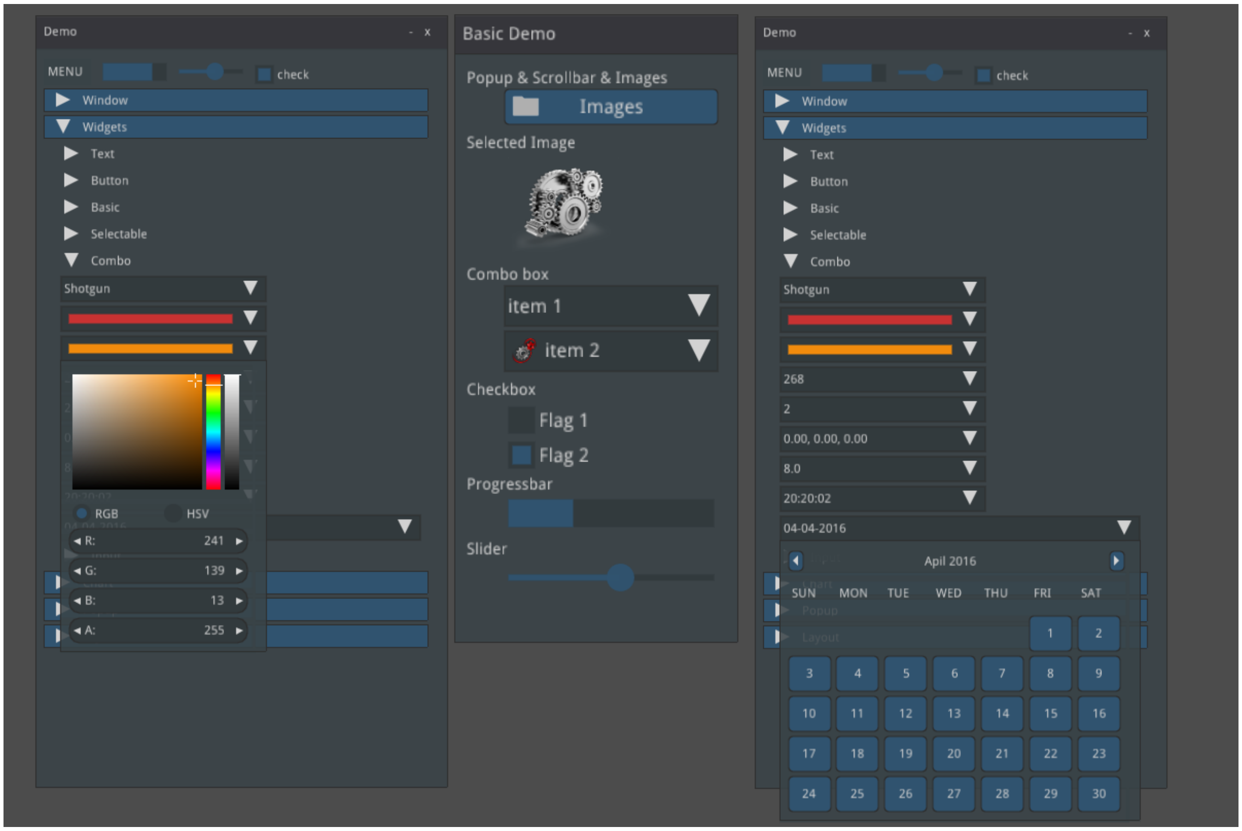
- **Other features**: fully skinnable and customisable.

- **Bindings stability**: stable
- **Bindings activity**: active
- **Licence**: MIT or Public Domain (unlicence).
- Example applications:
  - Trivial-gamekit
  - Obvius - a resurrected image processing library.
  - Notalone - an autumn 2017 Lisp Game Jam entry.

List of widgets

Non-exhaustive list:

```
buttons, progressbar, image selector, (collapsable) tree, list,
grid, range, slider, color picker,
date-picker
```



Getting started

Tk

Ltk is quick and easy to grasp.

```
(ql:quickload "ltk")
(in-package :ltk-user)
```

How to create widgets

All widgets are created with a regular `make-instance` and the widget name:

```
(make-instance 'button)
(make-instance 'treeview)
```

This makes Ltk explorable with the default symbol completion.

How to start the main loop

As with most bindings, the GUI-related code must be started inside a macro that handles the main loop, here `with-ltk`:

```
(with-ltk ()
  (let ((frame (make-instance 'frame)))
    …))
```

How to display widgets

After we created some widgets, we must place them on the layout. There are a few Tk systems for that, but the most recent one and the one we should start with is the `grid`. `grid` is a function that takes as arguments the widget, its column, its row, and a few optional parameters.

As with any Lisp code in a regular environment, the functions' signatures are indicated by the editor. It makes Ltk explorable.

Here's how to display a button:

```
(with-ltk ()
  (let ((button (make-instance 'button :text "hello")))
    (grid button 0 0)))
```

That's all there is to it.

Reacting to events

Many widgets have a `:command` argument that accept a lambda which is executed when the widget's event is started. In the case of a button, that will be on a click:

```
(make-instance 'button
  :text "Hello"
  :command (lambda ()
             (format t "clicked")))
```

Interactive development

When we start the Tk process in the background with `(start-wish)`, we can create widgets and place them on the grid interactively.

See [the documentation](#).

Once we're done, we can `(exit-wish)`.

Nodgui

To try the Nodgui demo, do:

```
(ql:quickload "nodgui")
(nodgui.demo:demo)
```

but hey, to load the demo with the better looking theme, do:

```
(nodgui.demo:demo :theme "yaru")
```

or

```
(setf nodgui:*default-theme* "yaru")
(nodgui.demo:demo)
```

Nodgui UI themes

To use the "yaru" theme that comes with nodgui, we can simply do:

```
(with-nodgui ()
  (use-theme "yaru")
  …)
```

or

```
(with-nodgui (:theme "yaru")
  …)
```

or

```
(setf nodgui:*default-theme* "yaru")
(with-nodgui ()
```

```
  …)
```

It is also possible to install and load another tcl theme. For example, clone the Forest ttk theme or the ttkthemes. Your project directory would look like this:

```
yourgui.asd
yourgui.lisp
ttkthemes/
```

Inside `ttkthemes/`, you will find themes under the `png/` directory (the other ones are currently not supported):

/ttkthemes/ttkthemes/png/arc/arc.tcl

You need to load the .tcl file with nodgui, and tell it to use this theme:

```
(with-nodgui ()
   (eval-tcl-file "/ttkthemes/ttkthemes/png/arc/arc.tcl")
   (use-theme "arc")
   … code here …)
```

and that's it. Your application now uses a new and decently looking GUI theme.

Qt4

```
(ql:quickload '(:qtools :qtcore :qtgui))

(defpackage #:qtools-test
  (:use #:cl+qt)
  (:export #:main))
(in-package :qtools-test)
(in-readtable :qtools)
```

We create our main widget that will contain the rest:

```
(define-widget main-window (QWidget)
  ())
```

We create an input field and a button inside this main widget:

```
(define-subwidget (main-window name) (q+:make-qlineedit main-win
  (setf (q+:placeholder-text name) "Your name please."))
```

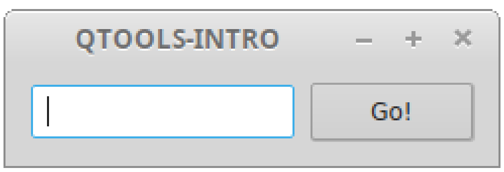
```
(define-subwidget (main-window go-button) (q+:make-qpushbutton "
```

We stack them horizontally:

```
(define-subwidget (main-window layout) (q+:make-qhboxlayout main
  (q+:add-widget layout name)
  (q+:add-widget layout go-button))
```

and we show them:

```
(with-main-window
  (window 'main-window))
```



That’s cool, but we don’t react to the click event yet.

Reacting to events

Reacting to events in Qt happens through signals and slots. **Slots** are functions that receive or “connect to” signals, and **signals** are event carriers.

Widgets already send their own signals: for example, a button sends a “pressed” event. So, most of the time, we only need to connect to them.

However, had we extra needs, we can create our own set of signals.

Built-in events

We want to connect our `go-button` to the `pressed` and `return-pressed` events and display a message box.

- we need to do this inside a `define-slot` function,
- where we establish the connection to those events,
- and where we create the message box. We grab the text of the `name` input field with `(q+:text name)`.

```
(define-slot (main-window go-button) ()
  (declare (connected go-button (pressed)))
  (declare (connected name (return-pressed)))
  (q+:qmessagebox-information main-window
                              "Greetings"  ;; title
                              (format NIL "Good day to you, ~a!"
```

And voilà. Run it with

```
(with-main-window (window 'main-window))
```

Custom events

We'll implement the same functionality as above, but for demonstration purposes we'll create our own signal named `name-set` to throw when the button is clicked.

We start by defining the signal, which happens inside the `main-window`, and which is of type `string`:

```
(define-signal (main-window name-set) (string))
```

We create a **first slot** to make our button react to the `pressed` and `return-pressed` events. But instead of creating the message box here, as above, we send the `name-set` signal, with the value of our input field..

```
(define-slot (main-window go-button) ()
  (declare (connected go-button (pressed)))
  (declare (connected name (return-pressed)))
  (signal! main-window (name-set string) (q+:text name)))
```

So far, nobody reacts to `name-set`. We create a **second slot** that connects to it, and displays our message. Here again, we precise the parameter type.

```
(define-slot (main-window name-set) ((new-name string))
  (declare (connected main-window (name-set string)))
  (q+:qmessagebox-information main-window "Greetings"
        (format NIL "Good day to you, ~a!" new-name)))
```

and run it:

```
(with-main-window (window 'main-window))
```

Building and deployment

It is possible to build a binary and bundle it together with all the necessary shared libraries.

Please read https://github.com/Shinmera/qtools#deployment.

You might also like this Travis CI script to build a self-contained binary for the three OSes.

Gtk3

The documentation is exceptionally good, including for beginners.

The library to quickload is `cl-cffi-gtk`. It is made of numerous ones, that we have to `:use` for our package.

```
(ql:quickload "cl-cffi-gtk")

(defpackage :gtk-tutorial
  (:use :gtk :gdk :gdk-pixbuf :gobject
   :glib :gio :pango :cairo :common-lisp))

(in-package :gtk-tutorial)
```

How to run the main loop

As with the other libraries, everything happens inside the main loop wrapper, here `with-main-loop`.

How to create a window

```
(make-instance 'gtk-window :type :toplevel :title "hello" ...).
```

How to create a widget

All widgets have a corresponding class. We can create them with `make-instance 'widget-class`, but we preferably use the constructors.

The constructors end with (or contain) “new”:

```
(gtk-label-new)
(gtk-button-new-with-label "Label")
```

How to create a layout

```
(let ((box (make-instance 'gtk-box :orientation :horizontal
                                   :spacing 6))) ...)
```

then pack a widget onto the box:

```
(gtk-box-pack-start box mybutton-1)
```

and add the box to the window:

```
(gtk-container-add window box)
```

and display them all:

```
(gtk-widget-show-all window)
```

Reacting to events

Use `g-signal-connect` + the concerned widget + the event name (as a string) + a lambda, that takes the widget as argument:

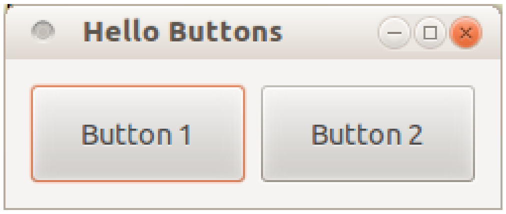
```
(g-signal-connect window "destroy"
  (lambda (widget)
    (declare (ignore widget))
    (leave-gtk-main)))
```

Or again:

```
(g-signal-connect button "clicked"
  (lambda (widget)
    (declare (ignore widget))
    (format t "Button was pressed.~%")))
```

Full example

```
(defun hello-world ()
  ;; in the docs, this is example-upgraded-hello-world-2.
  (within-main-loop
    (let ((window (make-instance 'gtk-window
                                 :type :toplevel
                                 :title "Hello Buttons"
                                 :default-width 250
                                 :default-height 75
                                 :border-width 12))
          (box (make-instance 'gtk-box
                              :orientation :horizontal
                              :spacing 6)))
      (g-signal-connect window "destroy"
                        (lambda (widget)
                          (declare (ignore widget))
                          (leave-gtk-main)))
      (let ((button (gtk-button-new-with-label "Button 1")))
        (g-signal-connect button "clicked"
                          (lambda (widget)
                            (declare (ignore widget))
                            (format t "Button 1 was pressed.~%")
        (gtk-box-pack-start box button))
      (let ((button (gtk-button-new-with-label "Button 2")))
        (g-signal-connect button "clicked"
                        (lambda (widget)
                          (declare (ignore widget))
                          (format t "Button 2 was pressed.~%")))
        (gtk-box-pack-start box button))
      (gtk-container-add window box)
      (gtk-widget-show-all window))))
```



IUP

Please check the installation instructions upstream. You may need one system dependency on GNU/Linux, and to modify an environment variable on Windows.

Finally, do:

```
(ql:quickload "iup")
```

We are not going to `:use` IUP (it is a bad practice generally after all).

```
(defpackage :test-iup
  (:use :cl))
(in-package :test-iup)
```

The following snippet creates a dialog frame to display a text label.

```
(defun hello ()
  (iup:with-iup ()
    (let* ((label (iup:label
                     :title
                     (format nil "Hello, World!~%IUP ~A~%~A ~A"
                       (iup:version)
                       (lisp-implementation-type)
                       (lisp-implementation-version))))
           (dialog (iup:dialog label :title "Hello, World!")))
      (iup:show dialog)
      (iup:main-loop))))
(hello)
```

Important note for SBCL: we currently must trap division-by-zero errors (see advancement on this issue). So, run snippets like so:

```lisp
(defun run-gui-function ()
  #-sbcl (gui-function)
  #+sbcl
  (sb-int:with-float-traps-masked
      (:divide-by-zero :invalid)
    (gui-function)))
```

How to run the main loop

As with all the bindings seen so far, widgets are shown inside a `with-iup` macro, and with a call to `iup:main-loop`.

How to create widgets

The constructor function is the name of the widget: `iup:label`, `iup:dialog`.

How to display a widget

Be sure to “show” it: `(iup:show dialog)`.

You can group widgets on `frames`, and stack them vertically or horizontally (with `vbox` or `hbox`, see the example below).

To allow a widget to be expanded on window resize, use `:expand :yes` (or `:horizontal` and `:vertical`).

Use also the `:alignement` properties.

How to get and set a widget’s attributes

Use `(iup:attribute widget attribute)` to get the attribute’s value, and use `setf` on it to set it.

Reacting to events

Most widgets take an `:action` parameter that takes a lambda function with one parameter (the handle).

```lisp
(iup:button :title "Test &1"
            :expand :yes
```

```
            :tip "Callback inline at control creation"
            :action (lambda (handle)
                      (iup:message "title" "button1's action cal
                      iup:+default+))
```

Below we create a label and put a button below it. We display a message dialog when we click on the button.

```
(defun click-button ()
  (iup:with-iup ()
    (let* ((label (iup:label :title
                                  (format nil "Hello, World!~%IUP ~A~%~A ~A"
                                      (iup:version)
                                      (lisp-implementation-type)
                                      (lisp-implementation-version))))
           (button (iup:button :title "Click me"
                               :expand :yes
                               :tip "yes, click me"
                               :action
                               (lambda (handle)
                                 (declare (ignorable handle))
                                 (iup:message "title"
                                              "button clicked")
                                 iup:+default+)))
           (vbox
            (iup:vbox (list label button)
                      :gap "10"
                      :margin "10x10"
                      :alignment :acenter))
           (dialog (iup:dialog vbox :title "Hello, World!")))
      (iup:show dialog)
      (iup:main-loop))))

#+sbcl
(sb-int:with-float-traps-masked
      (:divide-by-zero :invalid)
    (click-button))
```

Here's a similar example to make a counter of clicks. We use a label and its title to hold the count. The title is an integer.

```
(defun counter ()
  (iup:with-iup ()
    (let* ((counter (iup:label :title 0))
           (label (iup:label :title
                     (format nil "The button was clicked ~a time
                             (iup:attribute counter :title))))
           (button (iup:button :title "Click me"
                               :expand :yes
                               :tip "yes, click me"
                               :action (lambda (handle)
                                         (declare (ignorable hand
                                         (setf (iup:attribute cou
                                               (1+ (iup:attribute
                                         (setf (iup:attribute lal
                                               (format nil "The l
                                                       (iup:attr
                                         iup:+default+)))
           (vbox
            (iup:vbox (list label button)
                      :gap "10"
                      :margin "10x10"
                      :alignment :acenter))
           (dialog (iup:dialog vbox :title "Counter")))
      (iup:show dialog)
      (iup:main-loop))))

(defun run-counter ()
  #-sbcl
  (counter)
  #+sbcl
  (sb-int:with-float-traps-masked
      (:divide-by-zero :invalid)
    (counter)))
```

List widget example

Below we create three list widgets with simple and multiple selection, we set their default value (the pre-selected row) and we place them horizontally side by side.

```
(defun list-test ()
  (iup:with-iup ()
    (let*  ((list-1 (iup:list :tip "List 1"  ;; tooltip
                              ;; multiple selection
                              :multiple :yes
                              :expand :yes))
            (list-2 (iup:list :value 2   ;; default index of the
                              :tip "List 2" :expand :yes))
            (list-3 (iup:list :value 9 :tip "List 3" :expand :ye
            (frame (iup:frame
                    (iup:hbox
                     (progn
                       ;; populate the lists: display integers.
                       (loop for i from 1 upto 10
                          do (setf (iup:attribute list-1 i)
                                   (format nil "~A" i))
                          do (setf (iup:attribute list-2 i)
                                   (format nil "~A" (+ i 10)))
                          do (setf (iup:attribute list-3 i)
                                   (format nil "~A" (+ i 50))))
                       ;; hbox wants a list of widgets.
                       (list list-1 list-2 list-3)))
                    :title "IUP List"))
            (dialog (iup:dialog frame :menu "menu" :title "List 

      (iup:map dialog)
      (iup:show dialog)
      (iup:main-loop))))

(defun run-list-test ()
  #-sbcl (hello)
  #+sbcl
  (sb-int:with-float-traps-masked
      (:divide-by-zero :invalid)
    (list-test)))
```

Nuklear

Disclaimer: as per the author's words at the time of writing, bodge-ui is in early stages of development and not ready for general use yet. There are some quirks that need to be fixed, which might require some changes in the API.

`bodge-ui` is not in Quicklisp but in its own Quicklisp distribution. Let's install it:

```
(ql-dist:install-dist "http://bodge.borodust.org/dist/org.borodus
```

Uncomment and evaluate this line only if you want to enable the OpenGL 2 renderer:

```
;; (cl:pushnew :bodge-gl2 cl:*features*)
```

Quickload `bodge-ui-window`:

```
(ql:quickload "bodge-ui-window")
```

We can run the built-in example:

```
(ql:quickload "bodge-ui-window/examples")
(bodge-ui-window.example.basic:run)
```

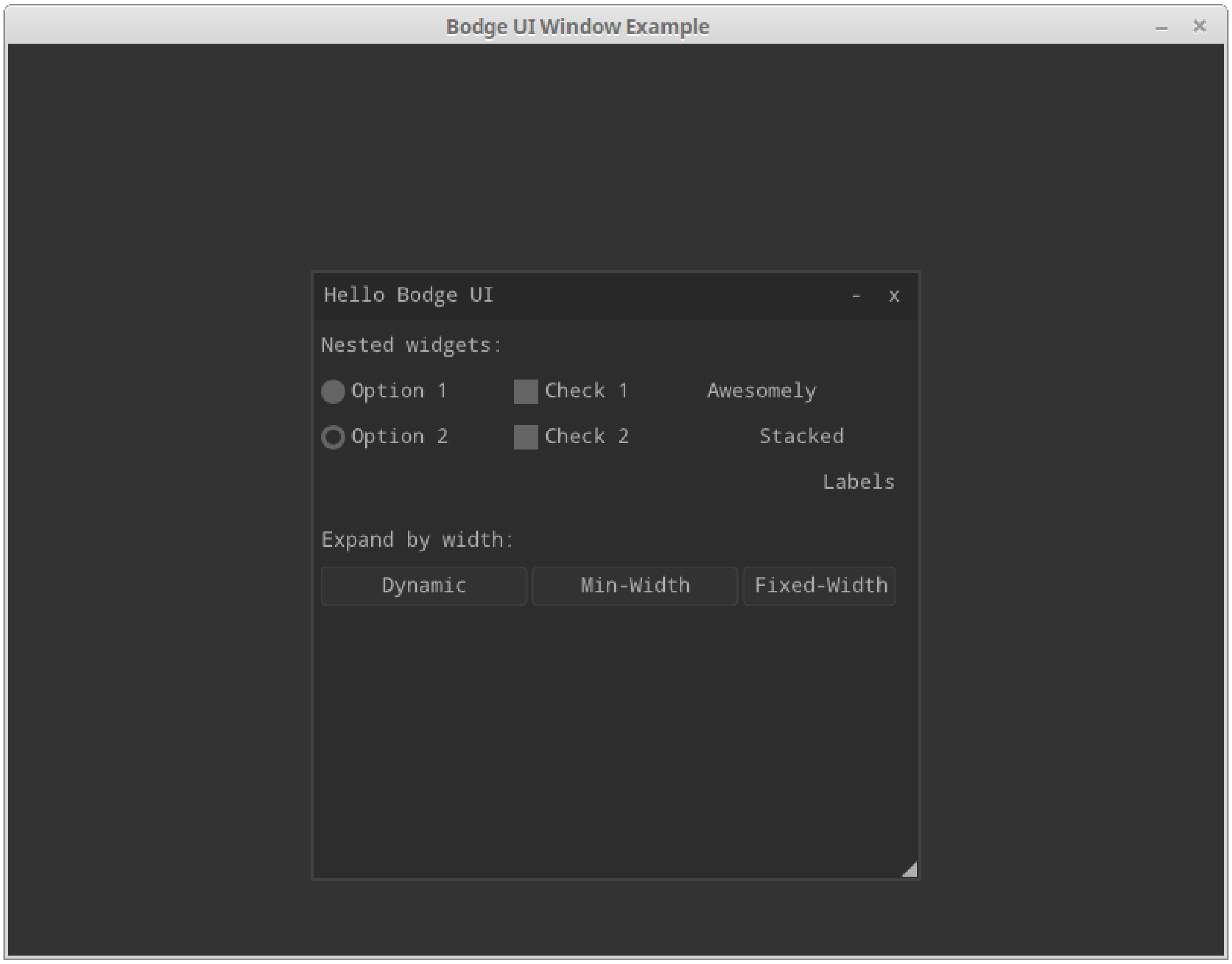
Now let's define a package to write a simple application.

```
(cl:defpackage :bodge-ui-window-test
  (:use :cl :bodge-ui :bodge-host))
(in-package :bodge-ui-window-test)

(defpanel (main-panel
           (:title "Hello Bodge UI")
           (:origin 200 50)
           (:width 400) (:height 400)
           (:options :movable :resizable
                     :minimizable :scrollable
                     :closable))
    (label :text "Nested widgets:")
  (horizontal-layout
   (radio-group
    (radio :label "Option 1")
    (radio :label "Option 2" :activated t))
   (vertical-layout
```

```lisp
    (check-box :label "Check 1" :width 100)
    (check-box :label "Check 2"))
   (vertical-layout
    (label :text "Awesomely" :align :left)
    (label :text "Stacked" :align :centered)
    (label :text "Labels" :align :right)))
  (label :text "Expand by width:")
  (horizontal-layout
   (button :label "Dynamic")
   (button :label "Min-Width" :width 80)
   (button :label "Fixed-Width" :expandable nil :width 100))
  (label :text "Expand by width:")
  (horizontal-layout
   (button :label "1.0" :expand-ratio 1.0)
   (button :label "0.75" :expand-ratio 0.75)
   (button :label "0.5" :expand-ratio 0.5))
  (label :text "Rest:")
  (button :label "Top-level Button"))

(defparameter *window-width* 800)
(defparameter *window-height* 600)

(defclass main-window (bodge-ui-window:ui-window) ()
  (:default-initargs
   :title "Bodge UI Window Example"
   :width *window-width*
   :height *window-height*
   :panels '(main-panel)
   :floating t
   :opengl-version #+bodge-gl2 '(2 1)
                   #+bodge-gl2 '(3 3)))



(defun run ()
  (bodge-host:open-window (make-instance 'main-window)))
```

and run it:

```lisp
(run)
```



To react to events, use the following signals:

```
:on-click
:on-hover
:on-leave
:on-change
:on-mouse-press
:on-mouse-release
```

They take as argument a function with one argument, the panel. But beware: they will be called on each rendering cycle when the widget is on the given state, so potentially a lot of times.

Interactive development

If you ran the example in the REPL, you couldn’t see what’s cool. Put the code in a lisp file and run it, so than you get the window. Now you can change the panel widgets and the layout, and your changes will be immediately applied while the application is running!

Conclusion

Have fun, and don’t hesitate to share your experience and your apps.

Web development

For web development as for any other task, one can leverage Common Lisp's advantages: the unmatched REPL that even helps to interact with a running web app, the exception handling system, performance, the ability to build a self-contained executable, stability, good threads story, strong typing, etc. We can, say, define a new route and try it right away, there is no need to restart any running server. We can change and compile *one function at a time* (the usual `C-c C-c` in Slime) and try it. The feedback is immediate. We can choose the degree of interactivity: the web server can catch exceptions and fire the interactive debugger, or print lisp backtraces on the browser, or display a 404 error page and print logs on standard output. The ability to build self-contained executables eases deployment tremendously (compared to, for example, npm-based apps), in that we just copy the executable to a server and run it.

And when we have deployed our app, we can still interact with it, allowing for hot reload, that even works when new dependencies have to be installed. If you are careful and don't want to use full live reload, you might still enjoy this capability to reload, for example, a user's configuration file.

We'll present here some established web frameworks and other common libraries to help you getting started in developing a web application. We do *not* aim to be exhaustive nor to replace the upstream documentation. Your feedback and contributions are appreciated.

Overview

Hunchentoot and Clack are two projects that you'll often hear about.

Hunchentoot is

> a web server and at the same time a toolkit for building dynamic websites. As a stand-alone web server, Hunchentoot is capable of HTTP/1.1 chunking (both directions), persistent connections (keep-

> alive), and SSL. It provides facilities like automatic session handling (with and without cookies), logging, customizable error handling, and easy access to GET and POST parameters sent by the client.

It is a software written by Edi Weitz ("Common Lisp Recipes", `cl-ppcre` and much more), it's used and proven solid. One can achieve a lot with it, but sometimes with more friction than with a traditional web framework. For example, dispatching a route by the HTTP method is a bit convoluted, one must write a function for the `:uri` parameter that does the check, when it is a built-in keyword in other frameworks like Caveman.

Clack is

> a web application environment for Common Lisp inspired by Python's WSGI and Ruby's Rack.

Also written by a prolific lisper (E. Fukamachi), it actually uses Hunchentoot by default as the server, but thanks to its pluggable architecture one can use another web server, like the asynchronous Woo, built on the libev event loop, maybe "the fastest web server written in any programming language".

We'll cite also Wookie, an asynchronous HTTP server, and its companion library cl-async, for general purpose, non-blocking programming in Common Lisp, built on libuv, the backend library in Node.js.

Clack being more recent and less documented, and Hunchentoot a de-facto standard, we'll concentrate on the latter for this recipe. Your contributions are of course welcome.

Web frameworks build upon web servers and can provide facilities for common activities in web development, like a templating system, access to a database, session management, or facilities to build a REST api.

Some web frameworks include:

- Caveman, by E. Fukamachi. It provides, out of the box, database management, a templating engine (Djula), a project skeleton generator,

a routing system à la Flask or Sinatra, deployment options (mod_lisp or FastCGI), support for Roswell on the command line, etc.
- Radiance, by Shinmera (Qtools, Portacle, lquery, …), is a web application environment, more general than usual web frameworks. It lets us write and tie websites and applications together, easing their deployment as a whole. It has thorough documentation, a tutorial, modules, pre-written applications such as an image board or a blogging platform, and more. For example websites, see https://shinmera.com/, reader.tymoon.eu and events.tymoon.eu.
- Snooze, by João Távora (Sly, Emacs' Yasnippet, Eglot, …), is "an URL router designed around REST web services". It is different because in Snooze, routes are just functions and HTTP conditions are just Lisp conditions.
- cl-rest-server is a library for writing REST web APIs. It features validation with schemas, annotations for logging, caching, permissions or authentication, documentation via OpenAPI (Swagger), etc.
- last but not least, Weblocks is a venerable Common Lisp web framework that permits to write ajax-based dynamic web applications without writing any JavaScript, nor writing some lisp that would transpile to JavaScript. It is seeing an extensive rewrite and update since 2017. We present it in more details below.

For a full list of libraries for the web, please see the awesome-cl list #network-and-internet and Cliki. If you are looking for a featureful static site generator, see Coleslaw.

Installation

Let's install the libraries we'll use:

```
(ql:quickload '("hunchentoot" "caveman2" "spinneret"
                "djula" "easy-routes"))
```

To try Weblocks, please see its documentation. The Weblocks in Quicklisp is not yet, as of writing, the one we are interested in.

We'll start by serving local files and we'll run more than one local server in the running image.

Simple webserver

Serve local files

Hunchentoot

Create and start a webserver like this:

```
(defvar *acceptor* (make-instance 'hunchentoot:easy-acceptor
                                  :port 4242))
(hunchentoot:start *acceptor*)
```

We create an instance of `easy-acceptor` on port 4242 and we start it. We can now access http://127.0.0.1:4242/. You should get a welcome screen with a link to the documentation and logs to the console.

By default, Hunchentoot serves the files from the `www/` directory in its source tree. Thus, if you go to the source of `easy-acceptor` (`M-.` in Slime), which is probably `~/quicklisp/dists/quicklisp/software/hunchentoot-v1.2.38/`, you'll find the `www/` directory. It contains:

- an `errors/` directory, with the error templates `404.html` and `500.html`,
- an `img/` directory,
- an `index.html` file.

To serve another directory, we give the option `:document-root` to `easy-acceptor`. We can also set the slot with its accessor:

```
(setf (hunchentoot:acceptor-document-root *acceptor*)
      #p"path/to/www")
```

Let's create our `index.html` first. Put this in a new `www/index.html` at the current directory (of the lisp repl):

```
<html>
  <head>
    <title>Hello!</title>
  </head>
  <body>
```

```
    <h1>Hello local server!</h1>
    <p>
    We just served our own files.
    </p>
  </body>
</html>
```

Let's start a new acceptor on a new port:

```
(defvar *my-acceptor* (make-instance 'hunchentoot:easy-acceptor
                                     :port 4444
                                     :document-root #p"www/"))
(hunchentoot:start *my-acceptor*)
```

go to http://127.0.0.1:4444/ and see the difference.

Note that we just created another *acceptor* on a different port on the same lisp image. This is already pretty cool.

Access your server from the internet

Hunchentoot

With Hunchentoot we have nothing to do, we can see the server from the internet right away.

If you evaluate this on your VPS:

(hunchentoot:start (make-instance 'hunchentoot:easy-acceptor :port 4242))

You can see it right away on your server's IP.

Stop it with `(hunchentoot:stop *)`.

Routing

Simple routes

Hunchentoot

To bind an existing function to a route, we create a “prefix dispatch” that we push onto the `*dispatch-table*` list:

```
(defun hello ()
   (format nil "Hello, it works!"))

(push
  (hunchentoot:create-prefix-dispatcher "/hello.html" #'hello)
  hunchentoot:*dispatch-table*)
```

To create a route with a regexp, we use `create-regex-dispatcher`, where the url-as-regexp can be a string, an s-expression or a cl-ppcre scanner.

If you didn’t yet, create an acceptor and start the server:

```
(defvar *server* (make-instance 'hunchentoot:easy-acceptor :port
(hunchentoot:start *server*)
```

and access it on http://localhost:4242/hello.html.

We can see logs on the REPL:

```
127.0.0.1 - [2018-10-27 23:50:09] "get / http/1.1" 200 393 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:58.0) Gecko/20100101 Firefox/58.0"
127.0.0.1 - [2018-10-27 23:50:10] "get /img/made-with-lisp-logo.jpg http/1.1" 200 12583 "http://localhost:4242/" "Mozilla/5.0 (X11; Linux x86_64; rv:58.0) Gecko/20100101 Firefox/58.0"
127.0.0.1 - [2018-10-27 23:50:10] "get /favicon.ico http/1.1" 200 1406 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:58.0) Gecko/20100101 Firefox/58.0"
127.0.0.1 - [2018-10-27 23:50:19] "get /hello.html http/1.1" 200 20 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:58.0) Gecko/20100101 Firefox/58.0"
```

[define-easy-handler]() allows to create a function and to bind it to an uri at once.

Its form follows

define-easy-handler (function-name :uri …) (lambda list parameters)

where <`uri`> can be a string or a function.

Example:

```
(hunchentoot:define-easy-handler (say-yo :uri "/yo") (name)
  (setf (hunchentoot:content-type*) "text/plain")
  (format nil "Hey~@[ ~A~]!" name))
```

Visit it at p://localhost:4242/yo and add parameters on the url: http://localhost:4242/yo?name=Alice.

Just a thought… we didn't explicitly ask Hunchentoot to add this route to our first acceptor of the port 4242. Let's try another acceptor (see previous section), on port 4444: http://localhost:4444/yo?name=Bob It works too ! In fact, `define-easy-handler` accepts an `acceptor-names` parameter:

> acceptor-names (which is evaluated) can be a list of symbols which means that the handler will only be returned by DISPATCH-EASY-HANDLERS in acceptors which have one of these names (see ACCEPTOR-NAME). acceptor-names can also be the symbol T which means that the handler will be returned by DISPATCH-EASY-HANDLERS in every acceptor.

So, `define-easy-handler` has the following signature:

define-easy-handler (function-name &key uri acceptor-names default-request-type) (lambda list parameters)

It also has a `default-parameter-type` which we'll use in a minute to get url parameters.

There are also keys to know for the lambda list. Please see the documentation.

Easy-routes (Hunchentoot)

easy-routes is a route handling extension on top of Hunchentoot. It provides:

- **dispatch** based on the HTTP method, such as GET or POST (which is otherwise cumbersome to do in Hunchentoot)
- **arguments extraction** from the url path
- **decorators** (functions to run before the route body, typically used to add a layer of authentication or changing the returned content type)
- **URL generation** from route names and given URL parameters
- visualization of routes
- and more

To use it, don't create a server with `hunchentoot:easy-acceptor` but with `easy-routes:easy-routes-acceptor`:

```
(setf *server* (make-instance 'easy-routes:easy-routes-acceptor)
```

Note: there is also `routes-acceptor`. The difference is that `easy-routes-acceptor` iterates over Hunchentoot's `*dispatch-table*` if no route is found by `easy-routes`. That allows us, for example, to serve static content the usual way with Hunchentoot.

Then define a route like this:

```
(easy-routes:defroute my-route-name ("/foo/:x" :method :get) (y &
    (format nil "x: ~a y: ~a z: ~a" x y z))
```

the route signature is made up of two parts:

("/foo/:x" :method :get) (y &get z)

Here, `:x` captures the path parameter and binds it to the `x` variable into the route body. `y` and `&get z` define URL parameters, and we can have `&post` parameters to extract from the HTTP request body.

These parameters can take an `:init-form` and `:parameter-type` options as in `define-easy-handler`.

Now, imagine that we are deeper in our web application logic, and we want to redirect our user to the route "/foo/3". Instead of hardcoding the URL, we can **generate the URL from its name**. Use `easy-routes:genurl` like this:

```
(easy-routes:genurl my-route-name :id 3)
;; => /foo/3

(easy-routes:genurl my-route-name :id 3 :y "yay")
;; => /foo/3?y=yay
```

Decorators are functions that are executed before the route body. They should call the `next` parameter function to continue executing the decoration chain and the route body finally. Examples:

```
(defun @auth (next)
  (let ((*user* (hunchentoot:session-value 'user)))
    (if (not *user*)
    (hunchentoot:redirect "/login")
    (funcall next))))

(defun @html (next)
  (setf (hunchentoot:content-type*) "text/html")
  (funcall next))

(defun @json (next)
  (setf (hunchentoot:content-type*) "application/json")
  (funcall next))
(defun @db (next)
  (postmodern:with-connection *db-spec*
    (funcall next)))
```

See `easy-routes`' readme for more.

Caveman

Caveman provides two ways to define a route: the `defroute` macro and the `@route` pythonic *annotation*:

```
(defroute "/welcome" (&key (|name| "Guest"))
  (format nil "Welcome, ~A" |name|))

@route GET "/welcome"
(lambda (&key (|name| "Guest"))
  (format nil "Welcome, ~A" |name|))
```

A route with an url parameter (note `:name` in the url):

```
(defroute "/hello/:name" (&key name)
  (format nil "Hello, ~A" name))
```

It is also possible to define "wildcards" parameters. It works with the `splat` key:

```
(defroute "/say/*/to/*" (&key splat)
  ; matches /say/hello/to/world
  (format nil "~A" splat))
;=> (hello world)
```

We must enable regexps with `:regexp t`:

```
(defroute ("/hello/([\\w]+)" :regexp t) (&key captures)
  (format nil "Hello, ~A!" (first captures)))
```

Accessing GET and POST parameters

Hunchentoot

First of all, note that we can access query parameters anytime with

```
(hunchentoot:parameter "my-param")
```

It acts on the default `*request*` object which is passed to all handlers.

There is also `get-parameter` and `post-parameter`.

Earlier we saw some key parameters to `define-easy-handler`. We now introduce `default-parameter-type`.

We defined the following handler:

```
(hunchentoot:define-easy-handler (say-yo :uri "/yo") (name)
  (setf (hunchentoot:content-type*) "text/plain")
  (format nil "Hey~@[ ~A~]!" name))
```

The variable `name` is a string by default. Let's check it out:

```
(hunchentoot:define-easy-handler (say-yo :uri "/yo") (name)
  (setf (hunchentoot:content-type*) "text/plain")
  (format nil "Hey~@[ ~A~] you are of type ~a" name (type-of name
```

Going to http://localhost:4242/yo?name=Alice returns

Hey Alice you are of type (SIMPLE-ARRAY CHARACTER (5))

To automatically bind it to another type, we use `default-parameter-type`. It can be one of those simple types:

- `'string` (default),
- `'integer`,
- `'character` (accepting strings of length 1 only, otherwise it is nil)
- or `'boolean`

or a compound list:

- `'(:list <type>)`
- `'(:array <type>)`
- `'(:hash-table <type>)`

where `<type>` is a simple type.

Accessing a JSON request body

Hunchentoot

To read a request body, use `hunchentoot:raw-post-data`, to which you can add `:force-text t` to always get a string (and not a vector of octets).

Then you can parse this string to JSON with the library of your choice (jzon, shasht…).

```
(easy-routes route-api-demo ("/api/:id/update" :method :post) ()
   (let ((json (ignore-errors
                 (jzon:parse (hunchentoot:raw-post-data :force-te
     (when json
        …)))
```

Error handling

In all frameworks, we can choose the level of interactivity. The web framework can return a 404 page and print output on the repl, it can catch errors and invoke the interactive lisp debugger, or it can show the lisp backtrace on the html page.

Hunchentoot

The global variables to set to choose the error handling behaviour are:

- `*catch-errors-p*`: set to `nil` if you want errors to be caught in the interactive debugger (for development only, of course):

```
(setf hunchentoot:*catch-errors-p* nil)
```

See also the generic function `maybe-invoke-debugger` if you want to fine-tune this behaviour. You might want to specialize it on specific condition classes (see below) for debugging purposes. The default method invokes the debugger if `*catch-errors-p*` is `nil`.

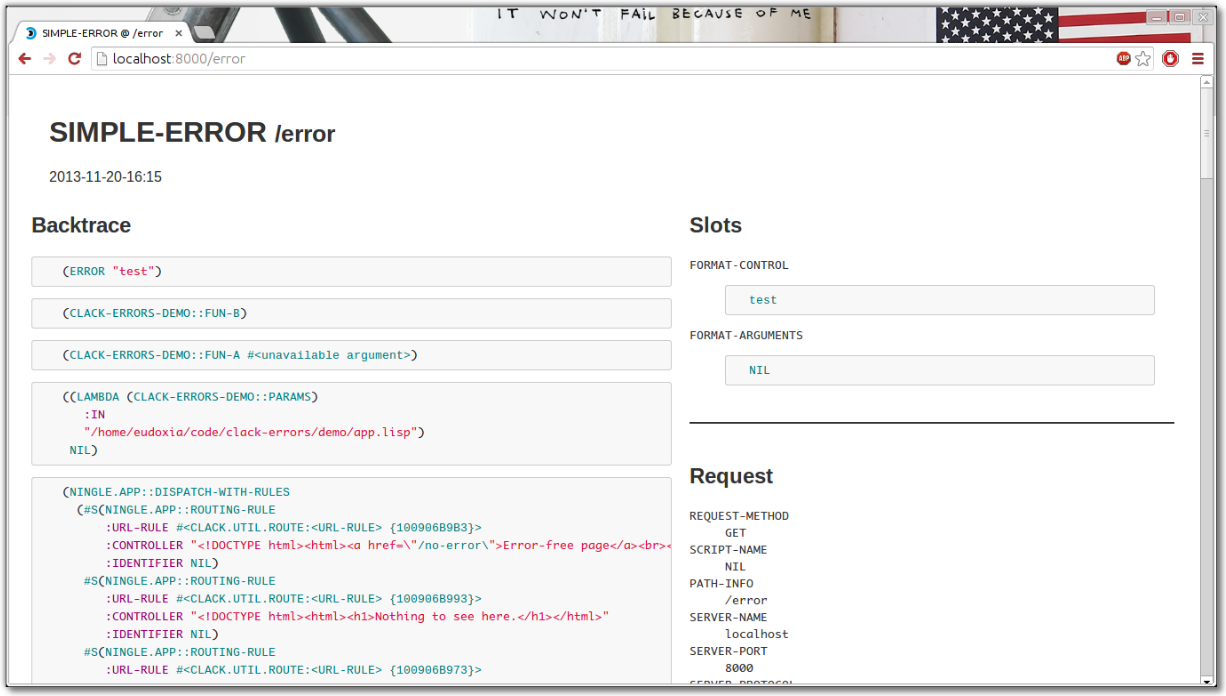
- `*show-lisp-errors-p*`: set to `t` if you want to see errors in HTML output in the browser.
- `*show-lisp-backtraces-p*`: set to `nil` if the errors shown in HTML output (when `*show-lisp-errors-p*` is `t`) should *not* contain backtrace information (defaults to `t`, shows the backtrace).

Hunchentoot defines condition classes. The superclass of all conditions is `hunchentoot-condition`. The superclass of errors is `hunchentoot-error` (itself a subclass of `hunchentoot-condition`).

See the documentation: https://edicl.github.io/hunchentoot/#conditions.

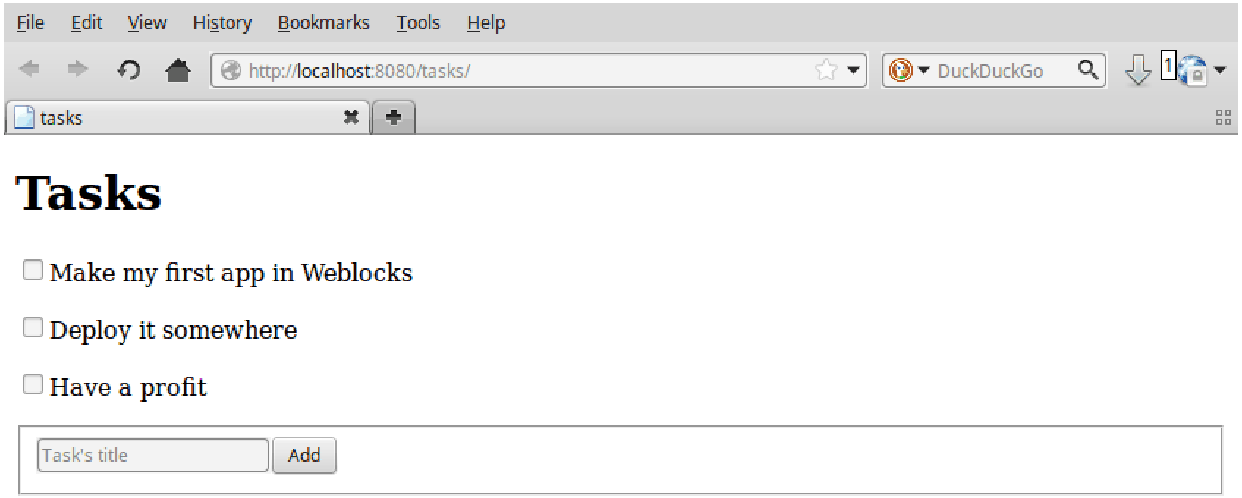
Clack

Clack users might make a good use of plugins, like the clack-errors middleware: https://github.com/CodyReichert/awesome-cl#clack-plugins.



Weblocks - solving the "JavaScript problem"©

[Weblocks](https://github.com/40ants/weblocks) is a widgets-based and server-based framework with a built-in ajax update mechanism. It allows to write dynamic web applications *without the need to write JavaScript or to write lisp code that would transpile to JavaScript*.



Weblocks is an old framework developed by Slava Akhmechet, Stephen Compall and Leslie Polzer. After nine calm years, it is seeing a very active update, refactoring and rewrite effort by Alexander Artemenko.

It was initially based on continuations (they were removed to date) and thus a lispy cousin of Smalltalk's [Seaside](). We can also relate it to Haskell's Haste, OCaml's Eliom, Elixir's Phoenix LiveView and others.

The [Ultralisp]() website is an example Weblocks website in production known in the CL community.

Weblock's unit of work is the *widget*. They look like a class definition:

```
(defwidget task ()
   ((title
     :initarg :title
     :accessor title)
    (done
     :initarg :done
     :initform nil
     :accessor done)))
```

Then all we have to do is to define the `render` method for this widget:

```
(defmethod render ((task task))
  "Render a task."
  (with-html
        (:span (if (done task)
                   (with-html
                         (:s (title task)))
                 (title task)))))
```

It uses the Spinneret template engine by default, but we can bind any other one of our choice.

To trigger an ajax event, we write lambdas in full Common Lisp:

```
...
(with-html
  (:p (:input :type "checkbox"
    :checked (done task)
    :onclick (make-js-action
              (lambda (&key &allow-other-keys)
                (toggle task))))
...
```

The function `make-js-action` creates a simple javascript function that calls the lisp one on the server, and automatically refreshes the HTML of the widgets that need it. In our example, it re-renders one task only.

Is it appealing ? Carry on this quickstart guide here: http://40ants.com/weblocks/quickstart.html.

Templates

Djula - HTML markup

Djula is a port of Python's Django template engine to Common Lisp. It has excellent documentation.

Caveman uses it by default, but otherwise it is not difficult to setup. We must declare where our templates are with something like

```
(djula:add-template-directory (asdf:system-relative-pathname "wel
```

and then we can declare and compile the ones we use, for example::

```
(defparameter +base.html+ (djula:compile-template* "base.html"))
(defparameter +welcome.html+ (djula:compile-template* "welcome.ht
```

A Djula template looks like this (forgive the antislash in `{\%`, this is a Jekyll limitation):

```
{\% extends "base.html" \%}
{\% block title %}Memberlist{\% endblock \%}
{\% block content \%}
  <ul>
  {\% for user in users \%}
    <li><a href="{{ user.url }}">{{ user.username }}</a></li>
  {\% endfor \%}
  </ul>
{\% endblock \%}
```

At last, to render the template, call `djula:render-template*` inside a route.

```
(easy-routes:defroute root ("/" :method :get) ()
  (djula:render-template* +welcome.html+ nil
                          :users (get-users)
```

Note that for efficiency Djula compiles the templates before rendering them.

It is, along with its companion access library, one of the most downloaded libraries of Quicklisp.

Djula filters

Filters allow to modify how a variable is displayed. Djula comes with a good set of built-in filters and they are well documented. They are not to be confused with tags.

They look like this: `{{ name | lower }}`, where `lower` is an existing filter, which renders the text into lowercase.

Filters sometimes take arguments. For example: `{{ value | add:2 }}` calls the `add` filter with arguments `value` and 2.

Moreover, it is very easy to define custom filters. All we have to do is to use the `def-filter` macro, which takes the variable as first argument, and which can take more optional arguments.

Its general form is:

```
(def-filter :myfilter-name (value arg) ;; arg is optional
   (body))
```

and it is used like this: `{{ value | myfilter-name }}`.

Here's how the `add` filter is defined:

```
(def-filter :add (it n)
  (+ it (parse-integer n)))
```

Once you have written a custom filter, you can use it right away throughout the application.

Filters are very handy to move non-trivial formatting or logic from the templates to the backend.

Spinneret - lispy templates

Spinneret is a "lispy" HTML5 generator. It looks like this:

```
(with-page (:title "Home page")
  (:header
   (:h1 "Home page"))
  (:section
   ("~A, here is *your* shopping list: " *user-name*)
   (:ol (dolist (item *shopping-list*)
          (:li (1+ (random 10)) item))))
  (:footer ("Last login: ~A" *last-login*)))
```

The author finds it is easier to compose the HTML in separate functions and macros than with the more famous cl-who. But it has more features under it sleeves:

- it warns on invalid tags and attributes
- it can automatically number headers, given their depth
- it pretty prints html per default, with control over line breaks
- it understands embedded markdown
- it can tell where in the document a generator function is (see `get-html-tag`)

Serve static assets

Hunchentoot

With Hunchentoot, use `create-folder-dispatcher-and-handler prefix directory`.

For example:

```
(push (hunchentoot:create-folder-dispatcher-and-handler
       "/static/" (merge-pathnames
                   "src/static" ; <-- starts without a /
                   (asdf:system-source-directory :myproject)))
      hunchentoot:*dispatch-table*)
```

Now our project's static files located under `/path/to/myproject/src/static/` are served with the `/static/` prefix:

```
<img src="/static/img/banner.jpg" />
```

Connecting to a database

Please see the databases section. The Mito ORM supports SQLite3, PostgreSQL, MySQL, it has migrations and db schema versioning, etc.

In Caveman, a database connection is alive during the Lisp session and is reused in each HTTP requests.

Checking a user is logged-in

A framework will provide a way to work with sessions. We'll create a little macro to wrap our routes to check if the user is logged in.

In Caveman, `*session*` is a hash table that represents the session's data. Here are our login and logout functions:

```
(defun login (user)
  "Log the user into the session"
  (setf (gethash :user *session*) user))

(defun logout ()
  "Log the user out of the session."
  (setf (gethash :user *session*) nil))
```

We define a simple predicate:

```
(defun logged-in-p ()
  (gethash :user cm:*session*))
```

and we define our `with-logged-in` macro:

```
(defmacro with-logged-in (&body body)
  `(if (logged-in-p)
       (progn ,@body)
       (render #p"login.html"
               '(:message "Please log-in to access this page."))
```

If the user isn't logged in, there will nothing in the session store, and we render the login page. When all is well, we execute the macro's body. We use it like this:

```
(defroute "/account/logout" ()
  "Show the log-out page, only if the user is logged in."
  (with-logged-in
    (logout)
    (render #p"logout.html")))

(defroute ("/account/review" :method :get) ()
  (with-logged-in
    (render #p"review.html"
            (list :review (get-review (gethash :user *session*))
```

and so on.

Encrypting passwords

With cl-pass

cl-pass is a password hashing and verification library. It is as simple to use as this:

```
(cl-pass:hash "test")
;; "PBKDF2$sha256:20000$5cf6ee792cdf05e1ba2b6325c41a5f10$19c7f2cc
(cl-pass:check-password "test" *)
;; t
(cl-pass:check-password "nope" **)
;; nil
```

You might also want to look at hermetic, a simple authentication system for Clack-based applications.

Manually (with Ironclad)

In this recipe we do the encryption and verification ourselves. We use the de-facto standard Ironclad cryptographic toolkit and the Babel charset encoding/decoding library.

The following snippet creates the password hash that should be stored in your database. Note that Ironclad expects a byte-vector, not a string.

```
(defun password-hash (password)
  (ironclad:pbkdf2-hash-password-to-combined-string
   (babel:string-to-octets password)))
```

`pbkdf2` is defined in RFC2898. It uses a pseudorandom function to derive a secure encryption key based on the password.

The following function checks if a user is active and verifies the entered password. It returns the user-id if active and verified and nil in all other cases even if an error occurs. Adapt it to your application.

```
(defun check-user-password (user password)
  (handler-case
      (let* ((data (my-get-user-data user))
             (hash (my-get-user-hash data))
             (active (my-get-user-active data)))
        (when (and active (ironclad:pbkdf2-check-password (babel
                                                            hash))
          (my-get-user-id data)))
    (condition () nil)))
```

And the following is an example on how to set the password on the database. Note that we use `(password-hash password)` to save the password. The rest is specific to the web framework and to the DB library.

```
(defun set-password (user password)
  (with-connection (db)
    (execute
     (make-statement :update :web_user
                     (set= :hash (password-hash password))
                     (make-clause :where
                                  (make-op := (if (integerp user)
                                                  :id_user
                                                  :email)
                                           user))))))
```

Credit: `/u/arvid` *on /r/learnlisp.*

Runnning and building

Running the application from source

To run our Lisp code from source, as a script, we can use the `--load` switch from our implementation.

We must ensure:

- to load the project's .asd system declaration (if any)

- to install the required dependencies (this demands we have installed Quicklisp previously)
- and to run our application's entry point.

We could use such commands:

```
;; run.lisp

(load "myproject.asd")

(ql:quickload "myproject")

(in-package :myproject)
(handler-case
    ;; The START function starts the web server.
    (myproject::start :port (ignore-errors
                              (parse-integer
                                (uiop:getenv "PROJECT_PORT"))))
  (error (c)
    (format *error-output* "~&An error occured: ~a~&" c)
    (uiop:quit 1)))
```

In addition we have allowed the user to set the application's port with an environment variable.

We can run the file like so:

sbcl –load run.lisp

After loading the project, the web server is started in the background. We are offered the usual Lisp REPL, from which we can interact with the running application.

We can also connect to the running application from our preferred editor, from home, and compile the changes in our editor to the running instance. See the following section #connecting-to-a-remote-lisp-image.

Building a self-contained executable

As for all Common Lisp applications, we can bundle our web app in one single executable, including the assets. It makes deployment very easy: copy it to your server and run it.

```
$ ./my-web-app
Hunchentoot server is started.
Listening on localhost:9003.
```

See this recipe on scripting#for-web-apps.

Continuous delivery with Travis CI or Gitlab CI

Please see the section on testing#continuous-integration.

Multi-platform delivery with Electron

Ceramic makes all the work for us.

It is as simple as this:

```
;; Load Ceramic and our app
(ql:quickload '(:ceramic :our-app))

;; Ensure Ceramic is set up
(ceramic:setup)
(ceramic:interactive)

;; Start our app (here based on the Lucerne framework)
(lucerne:start our-app.views:app :port 8000)

;; Open a browser window to it
(defvar window (ceramic:make-window :url "http://localhost:8000/"

;; start Ceramic
(ceramic:show-window window)
```

and we can ship this on Linux, Mac and Windows.

There is more:

> Ceramic applications are compiled down to native code, ensuring both performance and enabling you to deliver closed-source, commercial applications.

Thus, no need to minify our JS.

Deployment

Deploying manually

We can start our executable in a shell and send it to the background (`C-z bg`), or run it inside a `tmux` session. These are not the best but hey, it works©.

Systemd: Daemonizing, restarting in case of crashes, handling logs

This is actually a system-specific task. See how to do that on your system.

Most GNU/Linux distros now come with Systemd, so here's a little example.

Deploying an app with Systemd is as simple as writing a configuration file:

```
$ sudo emacs -nw /etc/systemd/system/my-app.service
[Unit]
Description=your lisp app on systemd example

[Service]
WorkingDirectory=/path/to/your/project/directory/
ExecStart=/usr/bin/make run  # or anything
Type=simple
Restart=on-failure

[Install]
WantedBy=network.target
```

Then we have a command to `start` it, only now:

sudo systemctl start my-app.service

and a command to install the service, to **start the app after a boot or reboot** (that's the "[Install]" part):

sudo systemctl enable my-app.service

Then we can check its `status`:

systemctl status my-app.service

and see our application's **logs** (we can write to stdout or stderr, and Systemd handles the logging):

journalctl -u my-app.service

(you can also use the `-f` option to see log updates in real time, and in that case augment the number of lines with `-n 50` or `--lines`).

Systemd handles crashes and **restarts the application**. That's the `Restart=on-failure` line.

Now keep in mind a couple things:

- we want our app to crash so that it can be re-started automatically: you'll want the `--disable-debugger` flag with SBCL.
- Systemd will, by default, run your app as root. If you rely on your Lisp to read your startup file (`~/.sbclrc`), especially to setup Quicklisp, you will need to use the `--userinit` flag, or to set the Systemd user with `User=xyz` in the `[service]` section. And if you use a startup file, be aware that the line `(user-homedir-pathname)` will not return the same result depending on the user, so the snippet might not find Quicklisp's setup.lisp file.

See more: https://www.freedesktop.org/software/systemd/man/systemd.service.html.

With Docker

There are several Docker images for Common Lisp. For example:

- clfoundation/sbcl includes the latest version of SBCL, many OS packages useful for CI purposes, and a script to install Quicklisp.
- 40ants/base-lisp-image is based on Ubuntu LTS and includes SBCL, CCL, Quicklisp, Qlot and Roswell.
- container-lisp/s2i-lisp is CentOs based and contains the source for building a Quicklisp based Common Lisp application as a reproducible docker image using OpenShift's source-to-image.

With Guix

GNU Guix is a transactional package manager, that can be installed on top of an existing OS, and a whole distro that supports declarative system configuration. It allows to ship self-contained tarballs, which also contain system dependencies. For an example, see the Nyxt browser.

Running behind Nginx

There is nothing CL-specific to run your Lisp web app behind Nginx. Here's an example to get you started.

We suppose you are running your Lisp app on a web server, with the IP address 1.2.3.4, on the port 8001. Nothing special here. We want to access our app with a real domain name (and eventuall benefit of other Nginx's advantages, such as rate limiting etc). We bought our domain name and we created a DNS record of type A that links the domain name to the server's IP address.

We must configure our server with Nginx to tell it that all connections coming from "your-domain-name.org", on port 80, are to be sent to the Lisp app running locally.

Create a new file: `/etc/nginx/sites-enabled/my-lisp-app.conf` and add this proxy directive:

```
server {
    listen www.your-domain-name.org:80;
    server_name your-domain-name.org www.your-domain-name.org;  ;
    location / {
```

```
        proxy_pass http://1.2.3.4:8001/;
    }

    # Optional: serve static files with nginx, not the Lisp app.
    location /files/ {
        proxy_pass http://1.2.3.4:8001/files/;
    }
}
```

Note that on the proxy_pass directive: `proxy_pass http://1.2.3.4:8001/;` we are using our server's public IP address. Oten, your Lisp webserver such as Hunchentoot directly listens on it. You might want, for security reasons, to run the Lisp app on localhost.

Reload nginx (send the "reload" signal):

$ nginx -s reload

and that's it: you can access your Lisp app from the outside through `http://www.your-domain-name.org`.

Deploying on Heroku and other services

See heroku-buildpack-common-lisp and the Awesome CL#deploy section for interface libraries for Kubernetes, OpenShift, AWS, etc.

Monitoring

See Prometheus.cl for a Grafana dashboard for SBCL and Hunchentoot metrics (memory, threads, requests per second,…).

Connecting to a remote Lisp image

This this section: debugging#remote-debugging.

Hot reload

This is an example from Quickutil. It is actually an automated version of the precedent section.

It has a Makefile target:

```
hot_deploy:
    $(call $(LISP), \
        (ql:quickload :quickutil-server) (ql:quickload :swank-cl
        (swank-client:with-slime-connection (conn "localhost" $(
            (swank-client:slime-eval (quote (handler-bind ((erro
                (ql:quickload :quickutil-utilities) (ql:quickloa
                (funcall (symbol-function (intern "STOP" :quicku
                (funcall (symbol-function (intern "START" :quick
        $($(LISP)-quit))
```

It has to be run on the server (a simple fabfile command can call this through ssh). Beforehand, a `fab update` has run `git pull` on the server, so new code is present but not running. It connects to the local swank server, loads the new code, stops and starts the app in a row.

See also

- Feather, a template for web application development, shows a functioning Hello World app with an HTML page, a JSON API, a passing test suite, a Postgres DB and DB migrations. Uses Qlot, Buildapp, SystemD for deployment.
- lisp-web-template-productlist, a simple project template with Hunchentoot, Easy-Routes, Djula and Bulma CSS.
- lisp-web-live-reload-example - a toy project to show how to interact with a running web app.

Credits

- https://lisp-journey.gitlab.io/web-dev/

Web Scraping

The set of tools to do web scraping in Common Lisp is pretty complete and pleasant. In this short tutorial we'll see how to make http requests, parse html, extract content and do asynchronous requests.

Our simple task will be to extract the list of links on the CL Cookbook's index page and check if they are reachable.

We'll use the following libraries:

- Dexador - an HTTP client (that aims at replacing the venerable Drakma),
- Plump - a markup parser, that works on malformed HTML,
- Lquery - a DOM manipulation library, to extract content from our Plump result,
- lparallel - a library for parallel programming (read more in the process section).

Before starting let's install those libraries with Quicklisp:

```
(ql:quickload '("dexador" "plump" "lquery" "lparallel"))
```

HTTP Requests

Easy things first. Install Dexador. Then we use the `get` function:

```
(defvar *url* "https://lispcookbook.github.io/cl-cookbook/")
(defvar *request* (dex:get *url*))
```

This returns a list of values: the whole page content, the return code (200), the response headers, the uri and the stream.

```
"<!DOCTYPE html>
 <html lang=\"en\">
  <head>
```

```
    <title>Home &ndash; the Common Lisp Cookbook</title>
    […]
    "
200
#<HASH-TABLE :TEST EQUAL :COUNT 19 {1008BF3043}>
#<QURI.URI.HTTP:URI-HTTPS https://lispcookbook.github.io/cl-cookbook/>
#<CL+SSL::SSL-STREAM for #<FD-STREAM for "socket 192.168.0.23:34897, peer: 151.101.120.133:443" {100781C133}>>
```

Remember, in Slime we can inspect the objects with a right-click on them.

Parsing and extracting content with CSS selectors

We'll use `lquery` to parse the html and extract the content.

- https://shinmera.github.io/lquery/

We first need to parse the html into an internal data structure. Use `(lquery:$ (initialize <html>))`:

```
(defvar *parsed-content* (lquery:$ (initialize *request*)))
;; => #<PLUMP-DOM:ROOT {1009EE5FE3}>
```

lquery uses [Plump](https://shinmera.github.io/plump/) internally.

Now we'll extract the links with CSS selectors.

Note: to find out what should be the CSS selector of the element I'm interested in, I right click on an element in the browser and I choose "Inspect element". This opens up the inspector of my browser's web dev tool and I can study the page structure.

So the links I want to extract are in a page with an `id` of value "content", and they are in regular list elements (`li`).

Let's try something:

```
(lquery:$ *parsed-content* "#content li")
;; => #(#<PLUMP-DOM:ELEMENT li {100B3263A3}> #<PLUMP-DOM:ELEMENT
;;  #<PLUMP-DOM:ELEMENT li {100B326423}> #<PLUMP-DOM:ELEMENT li
```

```
;;  #<PLUMP-DOM:ELEMENT li {100B3264A3}> #<PLUMP-DOM:ELEMENT li
;;  #<PLUMP-DOM:ELEMENT li {100B326523}> #<PLUMP-DOM:ELEMENT li
;;  #<PLUMP-DOM:ELEMENT li {100B3265A3}> #<PLUMP-DOM:ELEMENT li
;;  #<PLUMP-DOM:ELEMENT li {100B326623}> #<PLUMP-DOM:ELEMENT li
;;  […]
```

Wow it works ! We get here a vector of plump elements.

I'd like to easily check what those elements are. To see the entire html, we can end our lquery line with `(serialize)`:

```
(lquery:$  *parsed-content* "#content li" (serialize))
#("<li><a href=\"license.html\">License</a></li>"
  "<li><a href=\"getting-started.html\">Getting started</a></li
  "<li><a href=\"editor-support.html\">Editor support</a></li>"
  […]
```

And to see their *textual* content (the user-visible text inside the html), we can use `(text)` instead:

```
(lquery:$  *parsed-content* "#content" (text))
#("License" "Editor support" "Strings" "Dates and Times" "Hash T
  "Pattern Matching / Regular Expressions" "Functions" "Loop" "I
  "Files and Directories" "Packages" "Macros and Backquote"
  "CLOS (the Common Lisp Object System)" "Sockets" "Interfacing
  "Foreign Function Interfaces" "Threads" "Defining Systems"
  […]
  "Pascal Costanza's Highly Opinionated Guide to Lisp"
  "Loving Lisp - the Savy Programmer's Secret Weapon by Mark Wat
  "FranzInc, a company selling Common Lisp and Graph Database sc
```

All right, so we see we are manipulating what we want. Now to get their `href`, a quick look at lquery's doc and we'll use `(attr "some-name")`:

```
(lquery:$  *parsed-content* "#content li a" (attr :href))
;; => #("license.html" "editor-support.html" "strings.html" "dat
;;  "hashes.html" "pattern_matching.html" "functions.html" "loop
;;  "files.html" "packages.html" "macros.html"
;;  "/cl-cookbook/clos-tutorial/index.html" "os.html" "ffi.html'
;;  "process.html" "systems.html" "win32.html" "testing.html" "m
```

```
;;  […]
;;  "http://www.nicklevine.org/declarative/lectures/"
;;  "http://www.p-cos.net/lisp/guide.html" "https://leanpub.com/
;;  "https://franz.com/")
```

Note: using `(serialize)` after `attr` leads to an error.

Nice, we now have the list (well, a vector) of links of the page. We'll now write an async program to check and validate they are reachable.

External resources:

- CSS selectors

Async requests

In this example we'll take the list of url from above and we'll check if they are reachable. We want to do this asynchronously, but to see the benefits we'll first do it synchronously !

We need a bit of filtering first to exclude the email addresses (maybe that was doable in the CSS selector ?).

We put the vector of urls in a variable:

```
(defvar *urls* (lquery:$  *parsed-content* "#content li a" (attr
```

We remove the elements that start with "mailto:": (a quick look at the strings page will help)

```
(remove-if (lambda (it)
              (string= it "mailto:" :start1 0
                                    :end1 (length "mailto:")))
           *urls*)
;; => #("license.html" "editor-support.html" "strings.html" "dat
;;  […]
;;  "process.html" "systems.html" "win32.html" "testing.html" "m
;;  "license.html" "http://lisp-lang.org/"
;;  "https://github.com/CodyReichert/awesome-cl"
```

```
;;   "http://www.lispworks.com/documentation/HyperSpec/Front/inde
;;   […]
;;   "https://franz.com/")
```

Actually before writing the `remove-if` (which works on any sequence, including vectors) I tested with a `(map 'vector …)` to see that the results where indeed `nil` or `t`.

As a side note, there is a handy `starts-with` function in [cl-strings](), available in Quicklisp. So we could do:

```
(map 'vector (lambda (it)
                (cl-strings:starts-with it "mailto:"))
             *urls*)
```

it also has an option to ignore or respect the case.

While we're at it, we'll only consider links starting with "http", in order not to write too much stuff irrelevant to web scraping:

```
(remove-if-not (lambda (it)
                  (string= it "http" :start1 0 :end1 (length "htt
                *)
```

All right, we put this result in another variable:

```
(defvar *filtered-urls* *)
```

and now to the real work. For every url, we want to request it and check that its return code is 200. We have to ignore certain errors. Indeed, a request can timeout, be redirected (we don't want that) or return an error code.

To be in real conditions we'll add a link that times out in our list:

```
(setf (aref *filtered-urls* 0) "http://lisp.org")  ;; :/
```

We’ll take the simple approach to ignore errors and return `nil` in that case. If all goes well, we return the return code, that should be 200.

As we saw at the beginning, `dex:get` returns many values, including the return code. We’ll catch only this one with `nth-value` (instead of all of them with `multiple-value-bind`) and we’ll use `ignore-errors`, that returns nil in case of an error. We could also use `handler-case` and catch specific error types (see examples in dexador’s documentation) or (better yet ?) use `handler-bind` to catch any `condition`.

(*ignore-errors has the caveat that when there’s an error, we can not return the element it comes from. We’ll get to our ends though.*)

```
(map 'vector (lambda (it)
  (ignore-errors
    (nth-value 1 (dex:get it))))
  *filtered-urls*)
```

we get:

```
#(NIL 200 200 200 200 200 200 200 200 200 200 NIL 200 200 200
200 200 200 200
  200 200 200 200)
```

it works, but *it took a very long time*. How much time precisely ? with `(time …)`:

```
Evaluation took:
  21.554 seconds of real time
   0.188000 seconds of total run time (0.172000 user, 0.016000
system)
  0.87% CPU
  55,912,081,589 processor cycles
  9,279,664 bytes consed
```

21 seconds ! Obviously this synchronous method isn’t efficient. We wait 10 seconds for links that time out. It’s time to write and measure an async version.

After installing `lparallel` and looking at its documentation, we see that the parallel map pmap seems to be what we want. And it’s only a one word edit. Let’s try:

```
(time (lparallel:pmap 'vector
  (lambda (it)
    (ignore-errors
      (let ((status (nth-value 1 (dex:get it)))) status)))
  *filtered-urls*)
;;  Evaluation took:
;;  11.584 seconds of real time
;;  0.156000 seconds of total run time (0.136000 user, 0.020000
;;  1.35% CPU
;;  30,050,475,879 processor cycles
;;  7,241,616 bytes consed

;;
;;#(NIL 200 200 200 200 200 200 200 200 200 200 NIL 200 200 200
;;  200 200 200 200)
```

Bingo. It still takes more than 10 seconds because we wait 10 seconds for one request that times out. But otherwise it proceeds all the http requests in parallel and so it is much faster.

Shall we get the urls that aren’t reachable, remove them from our list and measure the execution time in the sync and async cases ?

What we do is: instead of returning only the return code, we check it is valid and we return the url:

```
... (if (and status (= 200 status)) it) ...
(defvar *valid-urls* *)
```

we get a vector of urls with a couple of `nil`s: indeed, I thought I would have only one unreachable url but I discovered another one. Hopefully I have pushed a fix before you try this tutorial.

But what are they ? We saw the status codes but not the urls :S We have a vector with all the urls and another with the valid ones. We’ll simply treat them as sets and compute their difference. This will show us the bad ones. We must transform our vectors to lists for that.

```
(set-difference (coerce *filtered-urls* 'list)
```

```
                 (coerce *valid-urls* 'list))
;; => ("http://lisp-lang.org/" "http://www.psg.com/~dlamkins/sl/
```

Gotcha !

BTW it takes 8.280 seconds of real time to me to check the list of valid urls synchronously, and 2.857 seconds async.

Have fun doing web scraping in CL !

More helpful libraries:

- we could use VCR, a store and replay utility to set up repeatable tests or to speed up a bit our experiments in the REPL.
- cl-async, carrier and others network, parallelism and concurrency libraries to see on the awesome-cl list, Cliki or Quickdocs.

WebSockets

The Common Lisp ecosystem boasts a few approaches to building WebSocket servers. First, there is the excellent Hunchensocket that is written as an extension to Hunchentoot, the classic web server for Common Lisp. I have used both and I find them to be wonderful.

Today, however, you will be using the equally excellent websocket-driver to build a WebSocket server with Clack. The Common Lisp web development community has expressed a slight preference for the Clack ecosystem because Clack provides a uniform interface to a variety of backends, including Hunchentoot. That is, with Clack, you can pick and choose the backend you prefer.

In what follows, you will build a simple chat server and connect to it from a web browser. The tutorial is written so that you can enter the code into your REPL as you go, but in case you miss something, the full code listing can be found at the end.

As a first step, you should load the needed libraries via quicklisp:

```
(ql:quickload '(clack websocket-driver alexandria))
```

The websocket-driver Concept

In websocket-driver, a WebSocket connection is an instance of the `ws` class, which exposes an event-driven API. You register event handlers by passing your WebSocket instance as the second argument to a method called `on`. For example, calling `(on :message my-websocket #'some-message-handler)` would invoke `some-message-handler` whenever a new message arrives.

The `websocket-driver` API provides handlers for the following events:

- `:open`: When a connection is opened. Expects a handler with zero arguments.
- `:message` When a message arrives. Expects a handler with one argument, the message received.
- `:close` When a connection closes. Expects a handler with two keyword args, a "code" and a "reason" for the dropped connection.
- `:error` When some kind of protocol level error occurs. Expects a handler with one argument, the error message.

For the purposes of your chat server, you will want to handle three cases: when a new user arrives to the channel, when a user sends a message to the channel, and when a user leaves.

Defining Handlers for Chat Server Logic

In this section you will define the functions that your event handlers will eventually call. These are helper functions that manage the chat server logic. You will define the WebSocket server in the next section.

First, when a user connects to the server, you need to give that user a nickname so that other users know whose chats belong to whom. You will also need a data structure to map individual WebSocket connections to nicknames:

```
;; make a hash table to map connections to nicknames
(defvar *connections* (make-hash-table))

;; and assign a random nickname to a user upon connection
(defun handle-new-connection (con)
  (setf (gethash con *connections*)
        (format nil "user-~a" (random 100000))))
```

Next, when a user sends a chat to the room, the rest of the room should be notified. The message that the server receives is prepended with the nickname of the user who sent it.

```
(defun broadcast-to-room (connection message)
  (let ((message (format nil "~a: ~a"
                         (gethash connection *connections*)
                         message)))
    (loop :for con :being :the :hash-key :of *connections* :do
          (websocket-driver:send con message))))
```

Finally, when a user leaves the channel, by closing the browser tab or navigating away, the room should be notified of that change, and the user's connection should be dropped from the `*connections*` table.

```
(defun handle-close-connection (connection)
  (let ((message (format nil " .... ~a has left."
                         (gethash connection *connections*))))
    (remhash connection *connections*)
    (loop :for con :being :the :hash-key :of *connections* :do
          (websocket-driver:send con message))))
```

Defining A Server

Using Clack, a server is started by passing a function to `clack:clackup`. You will define a function called `chat-server` that you will start by calling `(clack:clackup #'chat-server :port 12345)`.

A Clack server function accepts a single plist as its argument. That plist contains environment information about a request and is provided by the system. Your chat server will not make use of that environment, but if you want to learn more you can check out Clack's documentation.

When a browser connects to your server, a websocket will be instantiated and handlers will be defined on it for each of the the events you want to support. A WebSocket "handshake" will then be sent back to the browser, indicating that the connection has been made. Here's how it works:

```
(defun chat-server (env)
  (let ((ws (websocket-driver:make-server env)))

    (websocket-driver:on :open ws
                         (lambda () (handle-new-connection ws)))
```

```
    (websocket-driver:on :message ws
                         (lambda (msg)
                           (broadcast-to-room ws msg)))

    (websocket-driver:on :close ws
                         (lambda (&key code reason)
                           (declare (ignore code reason))
                           (handle-close-connection ws)))

    (lambda (responder)
      (declare (ignore responder))
      (websocket-driver:start-connection ws)))) ; send the hands
```

You may now start your server, running on port `12345`:

```
;; keep the handler around so that you can stop your server late

(defvar *chat-handler* (clack:clackup #'chat-server :port 12345)
```

A Quick HTML Chat Client

So now you need a way to talk to your server. Using Clack, define a simple application that serves a web page to display and send chats. First the web page:

```
(defvar *html*
  "<!doctype html>

<html lang=\"en\">
<head>
  <meta charset=\"utf-8\">
  <title>LISP-CHAT</title>
</head>

<body>
    <ul id=\"chat-echo-area\">
```

```
    </ul>
    <div style=\"position:fixed; bottom:0;\">
        <input id=\"chat-input\" placeholder=\"say something\" >
    </div>
    <script>
     window.onload = function () {
         const inputField = document.getElementById(\"chat-input

         function receivedMessage(msg) {
             let li = document.createElement(\"li\");
             li.textContent = msg.data;
             document.getElementById(\"chat-echo-area\").appendC
         }

         const ws = new WebSocket(\"ws://localhost:12345/chat\")
         ws.addEventListener('message', receivedMessage);

         inputField.addEventListener(\"keyup\", (evt) => {
             if (evt.key === \"Enter\") {
                 ws.send(evt.target.value);
                 evt.target.value = \"\";
             }
         });
     };

    </script>
</body>
</html>
")


(defun client-server (env)
    (declare (ignore env))


    `(200 (:content-type "text/html")
          (,*html*)))
```

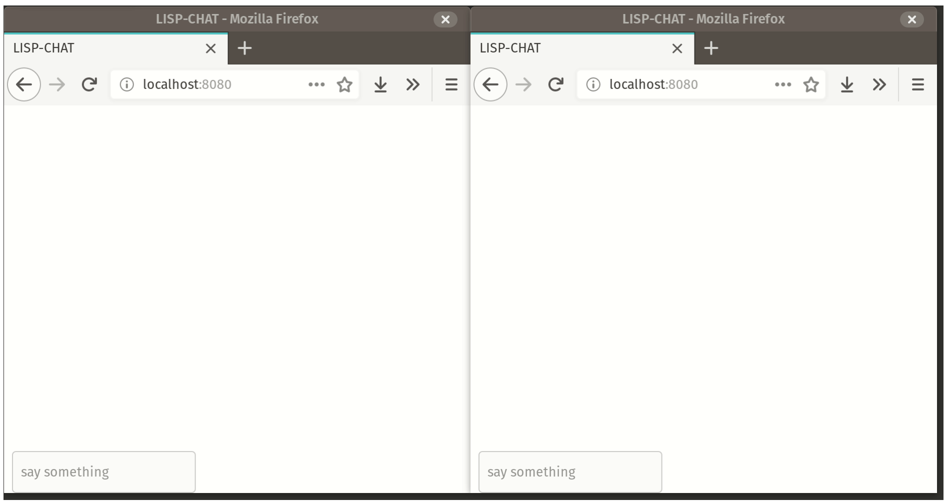
You might prefer to put the HTML into a file, as escaping quotes is kind of annoying. Keeping the page data in a `defvar` was simpler for the purposes of this tutorial.

You can see that the `client-server` function just serves the HTML content. Go ahead and start it, this time on port `8080`:

```
(defvar *client-handler* (clack:clackup #'client-server :port 80
```

Check it out!

Now open up two browser tabs and point them to `http://localhost:8080` and you should see your chat app!



All The Code

```
(ql:quickload '(clack websocket-driver alexandria))

(defvar *connections* (make-hash-table))

(defun handle-new-connection (con)
  (setf (gethash con *connections*)
        (format nil "user-~a" (random 100000))))

(defun broadcast-to-room (connection message)
  (let ((message (format nil "~a: ~a"

                         (gethash connection *connections*)
                         message)))
    (loop :for con :being :the :hash-key :of *connections* :do
```

```
            (websocket-driver:send con message))))

(defun handle-close-connection (connection)
  (let ((message (format nil " .... ~a has left."
                         (gethash connection *connections*))))
    (remhash connection *connections*)
    (loop :for con :being :the :hash-key :of *connections* :do
          (websocket-driver:send con message))))

(defun chat-server (env)
  (let ((ws (websocket-driver:make-server env)))
    (websocket-driver:on :open ws
                         (lambda () (handle-new-connection ws)))

    (websocket-driver:on :message ws
                         (lambda (msg)
                           (broadcast-to-room ws msg)))

    (websocket-driver:on :close ws
                         (lambda (&key code reason)
                           (declare (ignore code reason))
                           (handle-close-connection ws)))
    (lambda (responder)
      (declare (ignore responder))
      (websocket-driver:start-connection ws))))

(defvar *html*
  "<!doctype html>

<html lang=\"en\">
<head>
  <meta charset=\"utf-8\">
  <title>LISP-CHAT</title>
</head>

<body>
    <ul id=\"chat-echo-area\">
    </ul>

    <div style=\"position:fixed; bottom:0;\">
        <input id=\"chat-input\" placeholder=\"say something\" >
    </div>
```

```
    <script>
     window.onload = function () {
         const inputField = document.getElementById(\"chat-input

         function receivedMessage(msg) {
             let li = document.createElement(\"li\");
             li.textContent = msg.data;
             document.getElementById(\"chat-echo-area\").appendC
         }

         const ws = new WebSocket(\"ws://localhost:12345/\");
         ws.addEventListener('message', receivedMessage);

         inputField.addEventListener(\"keyup\", (evt) => {
             if (evt.key === \"Enter\") {
                 ws.send(evt.target.value);
                 evt.target.value = \"\";
             }
         });
     };

    </script>
</body>
</html>
")

(defun client-server (env)
  (declare (ignore env))
  `(200 (:content-type "text/html")
     (,*html*)))

(defvar *chat-handler* (clack:clackup #'chat-server :port 12345)
(defvar *client-handler* (clack:clackup #'client-server :port 80
```

APPENDIX: Contributors

Thank you to all contributors, as well as to the people reviewing pull requests whose name won't appear here.

The contributors on Github are:

- vindarel
- Paul Nathan
- nhabedi [1]
- Fernando Borretti
- bill_clementson
- chuchana
- Ben Dudson
- YUE Daian
- Pierre Neidhardt
- Rommel MARTINEZ
- digikar99
- nicklevine
- Dmitry Petrov
- otjura
- skeptomai
- alx-a
- jgart
- thegoofist
- Francis St-Amour
- Johan Widén
- emres
- jdcal
- Boutade
- airfoyle
- contrapunctus
- mvilleneuve
- Alex Ponomarev
- Alexander Artemenko

- Johan Sjölén
- Mariano Montone
- albertoriva
- Blue
- Daniel Keogh
- David Pflug
- David Sun
- Jason Legler
- Jiho Sung
- Kilian M. Haemmerle
- Matteo Landi
- Nikolaos Chatzikonstantinou
- Nisar Ahmad
- Nisen
- Vityok
- ctoid
- ozten
- reflektoin
- Ahmad Edrisy
- Alberto Ferreira
- Amol Dosanjh
- Andrew
- Andrew Hill
- André Alexandre Gomes
- Ankit Chandawala
- August Feng
- B1nj0y
- Bibek Panthi
- Bo Yao
- Brandon Hale
- Burhanuddin Baharuddin
- Coin Okay
- Colin Woodbury
- Daniel Uber
- Eric Timmons
- Giorgos Makris
- HiPhish

- Inc0n
- John Zhang
- Justin
- Kevin Layer
- Kevin Secretan
- LdBeth
- Matthew Kennedy
- Momozor
- NCM
- Noor
- Paul Donnelly
- Pavel Kulyov
- Phi-Long Nguyen
- R Primus
- Ralf Doering
- Salad Tea
- Victor Anyakin
- alaskasquirrel
- blackeuler
- contrapunctus-1
- convert-repo
- dangerdyke
- grobe0ba
- jthing
- mavis
- mwgkgk
- paul-donnelly
- various-and-sundry
- Štěpán Němec

(this list is sorted by number of commits)

And the contributors on the original SourceForge version are:

- Marco Antoniotti
- Zach Beane
- Pierpaolo Bernardi

- Christopher Brown
- Frederic Brunel
- Jeff Caldwell
- Bill Clementson
- Martin Cracauer
- Gerald Doussot
- Paul Foley
- Jörg-Cyril Höhle
- Nick Levine
- Austin King
- Lieven Marchand
- Drew McDermott
- Kalman Reti
- Alberto Riva
- Rudi Schlatte
- Emre Sevinç
- Paul Tarvydas
- Kenny Tilton
- Reini Urban
- Matthieu Villeneuve
- Edi Weitz

Finally, the credit for finally giving birth to the project probably goes to Edi Weitz who posted this message to comp.lang.lisp.

---

1. nhabedi is Edmund Weitz ;)↩